



TKN

Telecommunication
Networks Group

Technische Universität Berlin
Telecommunication Networks Group

UniFlex: Accelerating Networking Research and Experimentation through Software-Defined Wireless Networking

Piotr Gawłowicz, Anatolij Zubow, Mikolaj Chwalisz and
Adam Wolisz

{gawlowicz, zubow, chwalisz, wolisz}@tkn.tu-berlin.de

Berlin, November 2016

TKN Technical Report TKN 16-0003

TKN Technical Reports Series
Editor: Prof. Dr.-Ing. Adam Wolisz

Abstract

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).

We propose UniFlex, a framework enabling unified and flexible radio and network control. It provides an API enabling coordinated cross-layer control and management operation over multiple network nodes. The controller logic may be implemented either in a centralized or distributed manner. This allows to place time-sensitive control functions close to the controlled device (i.e., local control application), off-load more resource hungry network application to compute servers and make them work together to control entire network.

The UniFlex framework was prototypically implemented and provided to the research community as open-source. We evaluated the framework in a number of use-cases, what proved its usability.

Contents

1	Introduction	5
2	System Model	7
3	Requirements and Design Principles	8
4	Architecture Overview	10
4.1	Control Applications	10
4.2	Northbound Interface	11
4.3	Distributed Control Framework	12
4.4	Southbound Interface	12
4.5	UniFlex Framework UML Diagram	13
5	Implementation Details	15
5.1	Northbound Interface	15
5.2	Agent	17
5.3	Broker	17
5.4	Control Application	17
5.5	Southbound Interface	18
5.6	Deployment	19
5.7	Support of other Programming Languages	19
6	Integration with External Software Libraries	20
6.1	Python Scientific Packages	20
6.2	Node-RED Integration	20
6.3	Mininet Integration	20
7	Applications	22
7.1	Wireless Topology Monitor	22
7.2	Mobility Management	23
7.3	Interference Management	23
7.4	Device-to-Device	24
8	Evaluation	26
8.1	Basic Network Operation	26
8.2	Scalability	27
9	Related Work	28

10 Conclusions	29
11 Acknowledgments	29
12 Appendix A – Description of UniFlex APIs	33

Chapter 1

Introduction

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 the matter of fact the only widely accepted approach is the usage of SNMP or NETCONF 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 couple of attempts for wireless control protocols including LWAPP and CAPWAP, but those were designed with focus on 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 example 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 [1]. Here in recent years we have seen a boom of cross-layer design proposals for wireless networks [2, 3] where additional information from some layers are obtained and used to optimize operation of another layer.

So far control applications had to solve the challenges 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 individual 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 or distributed controllers [4].

Contribution: In this paper we propose **UniFlex**, a framework for **Unified** and **Flexible** radio and network control. The suggested 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, but support equally well also hierarchical control structure and logically centralized but physically distributed control. Network control applications can be either co-located with 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 application runs on server due to requiring high computing power). We believe that UniFlex will be an enabler for rapid prototyping of control applications for wireless network devices, management and control of operation of - possibly heterogeneous - nodes in wireless networks. The UniFlex prototype is provided as open-source to the community: <https://github.com/UniFlex>.

The rest of this report is organized as follows. In Sec. II the system model is introduced, while in Sec. III requirements for the wireless network control plane are defined. Sec. IV describes the architecture of envisioned UniFlex framework. In Sec. V and VI details of our prototypical implementation and integration with external software are given. Examples of wireless network control applications implemented using UniFlex are presented in Sec. VII. Performance of UniFlex prototype is evaluated in Sec. VIII. In Sec. IX related research is discussed and finally, Sec. X summarizes our main findings and concludes the paper.

Chapter 2

System Model

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

The **network** is a collection of **nodes** under common management and control domain (authority). A **node** is collection of equipment sharing a common platform and being run under a single instance of operating system. The types of nodes span from small constrained devices to powerful compute servers. A node is equipped with zero or more **devices**. A **device** is a piece of hardware fulfilling a dedicated functionality. Additionally, it may expose set of operations in Native Device Programming Interface (**NDPI**) to control its behaviour and parameters. There are different classes of sets of operations for different devices. For example, a **wireless network device** provides packet forwarding functions with usage of wireless transmission technology (e.g. 802.11, LTE) and exposes NDPI to control its parameters including transmission power, central frequency, bandwidth, etc.

Operation of a network is harmonized by single or multiple **controllers**. The controller logic may be implemented either as standalone or multiple cooperating **control applications** (aka network applications) that run in node(s). In particular, a control application may be located in the same node as network device that it is controlling.

We assume the existence of a common **Control Channel** enabling control application(s) to: i) access NDPI 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).

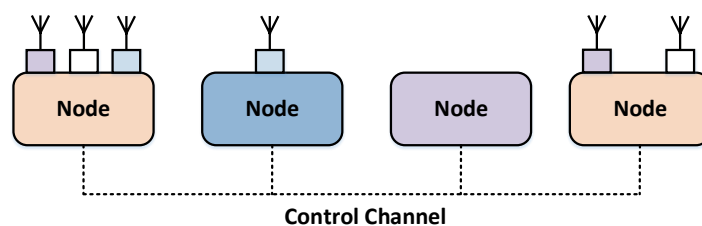


Figure 2.1: System model overview.

Chapter 3

Requirements and Design Principles

The main goal of our work is to facilitate and shorten time required for prototyping of novel control solutions in heterogeneous wireless networks. We argue that novel wireless applications may be realized when following functionality is provided:

- Coordinated **collection** of information from and **execution** of commands on different protocol layers (**cross-layer**), heterogeneous devices (**cross-technology**) and multiple nodes (**cross-node**) within 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 applications, i.e. placing time-sensitive task close to device and off-loading greedy tasks to powerful servers,
- Support for multiple levels of control for scalability reasons, i.e. local control applications handle frequent commands and events, while remote controllers handle rare events (Fig. 3.1),
- Support for detecting network changes in proactive and reactive control schemes in control applications,
- 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,
- 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 on 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 proactive approach, the network controller is polling the network entities, while in reactive approach execution of control program functions is triggered by events generated by the nodes in the network. It should be up to experimenter to define her 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 switch 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. A unification of the different Native Device Programming Interface (NDPI) would be desirable as it would allow controlling the devices of a HetNet in a unified way. Similar concept is nowadays developed for wired switches, e.g. Switch Abstraction Interface [5].

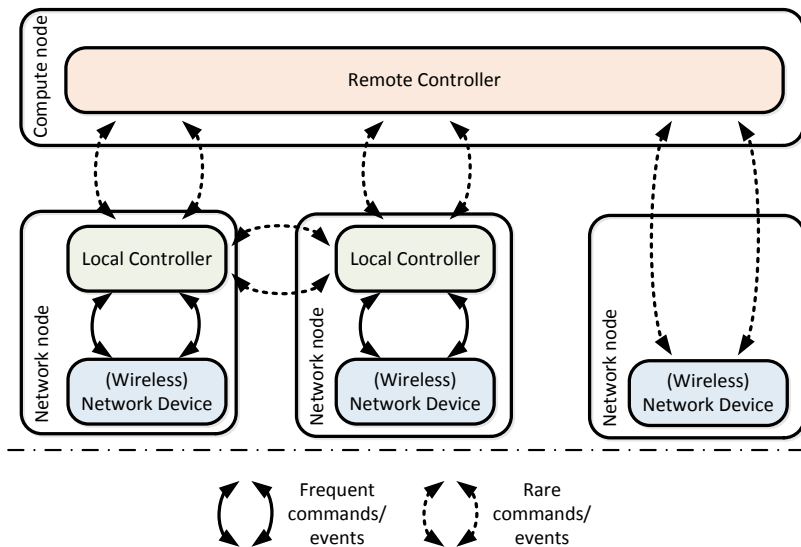


Figure 3.1: UniFlex’s levels of control. Local controllers handle frequent commands and events, while remote controllers handle rare events. The exchange of events between controllers is also rare.

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 application a lot. However, from the practical point of view a centralized controller would introduce a significant delay in the control plane, what can prevent time sensitive control logic to be implemented. Moreover, transporting all monitoring data from devices to a central controller may create too high load on the control plane. Sometimes pre-processing data locally at the node is feasible. In this way, controller may be partitioned into smaller applications where parts of them would run on the network nodes and others on the central compute node. Another advantage of such a split is the possibility of re-usage of control applications. For example, an averaging filter may be implemented once as control application and used as part of many other controllers. Fig. 3.1 shows the UniFlex’s levels of control. Local controllers handle frequent commands and events, while remote controllers handle rare events. The exchange of events between controllers is also rare.

In summary, the envisioned control framework may be considered as a wireless network operating system, that runs multiple control applications (network apps) communicating with each other and providing them with an interface for controlling all wireless network devices in coordinated way.

Chapter 4

Architecture Overview

Fig. 4.1 shows an overview of the architecture of the UniFlex framework. The framework is a distributed middleware platform running across multiple nodes that interconnects all control applications and wireless network devices. The network applications running on top of middleware perform control tasks over wireless devices by utilizing the provided API in Northbound Interface. The Southbound Interface is responsible for translating calls coming from control applications to NDPI. In the following subsections, we provide detailed description of the UniFlex framework.

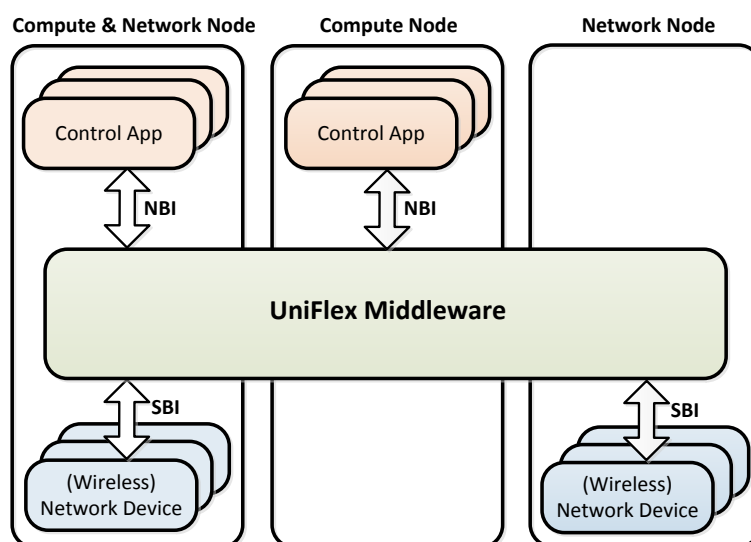


Figure 4.1: UniFlex's high-level architecture: Northbound Interface (NBI) and Southbound Interface (SBI).

4.1 Control Applications

A control application is an entity that implements particular network controller logic. In general, it collects information and measurements from network, make control decisions according to set policy and perform network reconfiguration.

Each control application is provided with a global view on all nodes in the network. Basically, control application is operating on proxy object of remote nodes. Each node proxy contains proxy objects referring to application and device modules present in a given remote node. By default, each

control application is able to communicate with all applications and control all device modules in entire wireless network, but some access policies may be applied.

Control applications may send as well as receive events generated by framework, devices and other applications. In order to be notified about specific event, an application has to subscribe to this event. Events originated from devices are usually used in reactive control approach, e.g. every time a network device is not able to receive a frame correctly, it sends a *FrameLostEvent*. Applications may use events to communicate with each other.

Events may be sent in three modes: i) unicast, ii) node-broadcast and iii) global-broadcast. Using unicast mode an application is able to send events to a particular recipient. In node-broadcast mode application sends events to a particular node and all applications running on that given node will receive it. Finally, using global-broadcast applications and devices are able to send events to all applications in the entire network. An event always contains an identifier of node and entity (application or device) that generated it. In addition, events of specific type may contain additional data, as measurement samples, etc.

4.2 Northbound Interface

Fig. 4.3 shows the Northbound Interface description as UML class diagram. Each network control application operates on proxy objects referring to nodes, applications and device modules. Those proxy objects provides an access to the UniFlex Northbound Interface. The NBI is the same for devices located in the same host machine as well as for remote devices. It is the framework that is responsible for delivering and executing function calls in proper device.

The Northbound Interface gives an application possibility to send, subscribe for and receive events. An application sends events using *send_event(event)* function and subscribe for events using *subscribe_for_events(eventType, callback)*. Once subscribed the framework will deliver events of proper type to application and fire bounded callback passing event as the argument. In order to send/receive events in unicast, node-broadcast and global-broadcast mode, an application has to execute *send_event()/subscribe_for_events()* function on application (device) proxy, node proxy and self-object, respectively.

In addition, a control application may execute Remote Procedure Calls (RPC) on device/protocol proxy object to call functions from its native programming interface. By default, all RPC calls are executed on the device/protocol proxy object are blocking execution of control application until function returns. For convenience, the `DeviceProxy/ProtocolProxy` 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 callback function for value returned from function call. Calling examples are given in Sec. 5.1.

In order to control a node, an application has to first obtain a `NodeProxy` object. This is achieved by subscribing to `NewNodeEvent` events. On discovery of a new node UniFlex notifies the application about this by sending event containing a node proxy. Thereafter, the application can retrieve the `DeviceProxy`, `ProtocolProxy` and/or `ApplicationProxy` objects from obtained `NodeProxy` and execute commands on them. Note that application is notified about all nodes in network including the one that it run on top of – a proper flag *Local* in `NodeProxy` is used to distinguish

between local and remote nodes. The framework tracks presence of all nodes in network and notifies applications about node lost events.

4.3 Distributed Control Framework

The UniFlex is a distributed middleware platform that inter-connects network devices and control applications. The framework takes care of node management including node discovery and monitoring connection to between all nodes. Whenever new node is discovered or connection with some node is lost, UniFlex notifies all applications about the changes by sending proper event.

While Control Application running on top of the framework are able to control wireless devices/protocols with simple API, it is the framework, that is responsible for delivering and executing function calls in proper device/protocol. Moreover, the middle-ware is also responsible for discovery of capabilities of each device/protocol and throwing exceptions if not supported function is to be called.

Using UniFlex it is possible to develop distributed network control applications, i.e. it is possible to split-up control logic into smaller cooperating control applications running on different nodes. In this way, UniFlex by design support three types of control programs: i) local – when controller is running in the same node as controlled device(s), ii) non-local – when controller is running on different node then controlled device(s) and iii) hybrid or hierarchical – when controller logic is split between 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 offload complex tasks to remote compute nodes (servers in cloud). Note that framework provides location transparency meaning that calling syntax is always the same for local as well as remote device.

4.4 Southbound Interface

The Southbound Interface works in two directions, i.e. control application 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 with RPC calls, while communication in out-direction is realized using events.

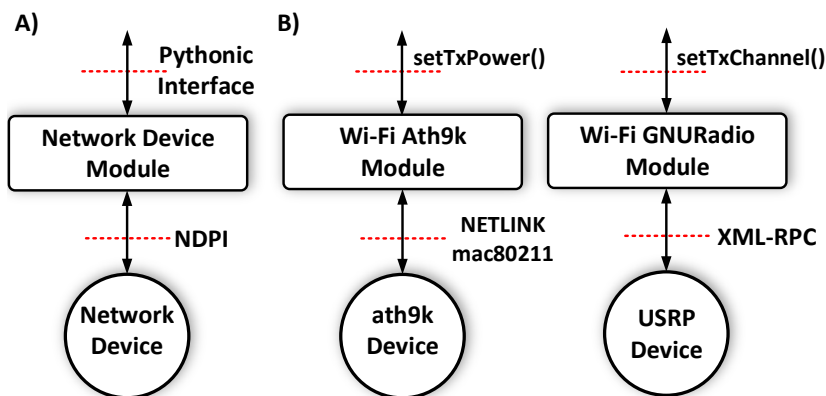


Figure 4.2: Device Module is an Python wrapper for Native Device Programming Interface.

The Southbound Interface is realized with help of Device and Protocol Modules. Device Module translates function calls from control applications into Native Device Programming Interface (NDPI) – Fig.4.2A. While Protocol Module translates function calls into Native Protocol Programming Interface (NPPI). In other words, Device/Protocol Module wraps different API and tools used to program device/protocol, and exposes them to framework in *Pythonic* way (i.e. as Python functions). In Fig.4.2B, we present two Device Modules as an example. As shows the Python functions are delivered to modules and translated to proper NDPI calls, i.e. NETLINK and XML/RPC respectively.

In many cases, while the semantics of features supported by different devices may be the same, their NDPI may differ greatly. We argue that this would be error prone and would prevent portability and re-usability of control application. We believe that those issues can be solved by a unified abstraction layer that hides specific NDPIs of different devices behind common one. We found Unified Programming Interface (UPI) defined in WiSHFUL project [6, 7] to be an appropriate option.

4.5 UniFlex Framework UML Diagram

The UML diagram in Fig. 4.3 presents UniFlex architecture in more detail. The Distributed Control Framework is consists of set of communicating Agents. The *Agent* is provides communication middleware for Control Applications, Device and Protocol Modules. Description of functions is given in Appendix A.

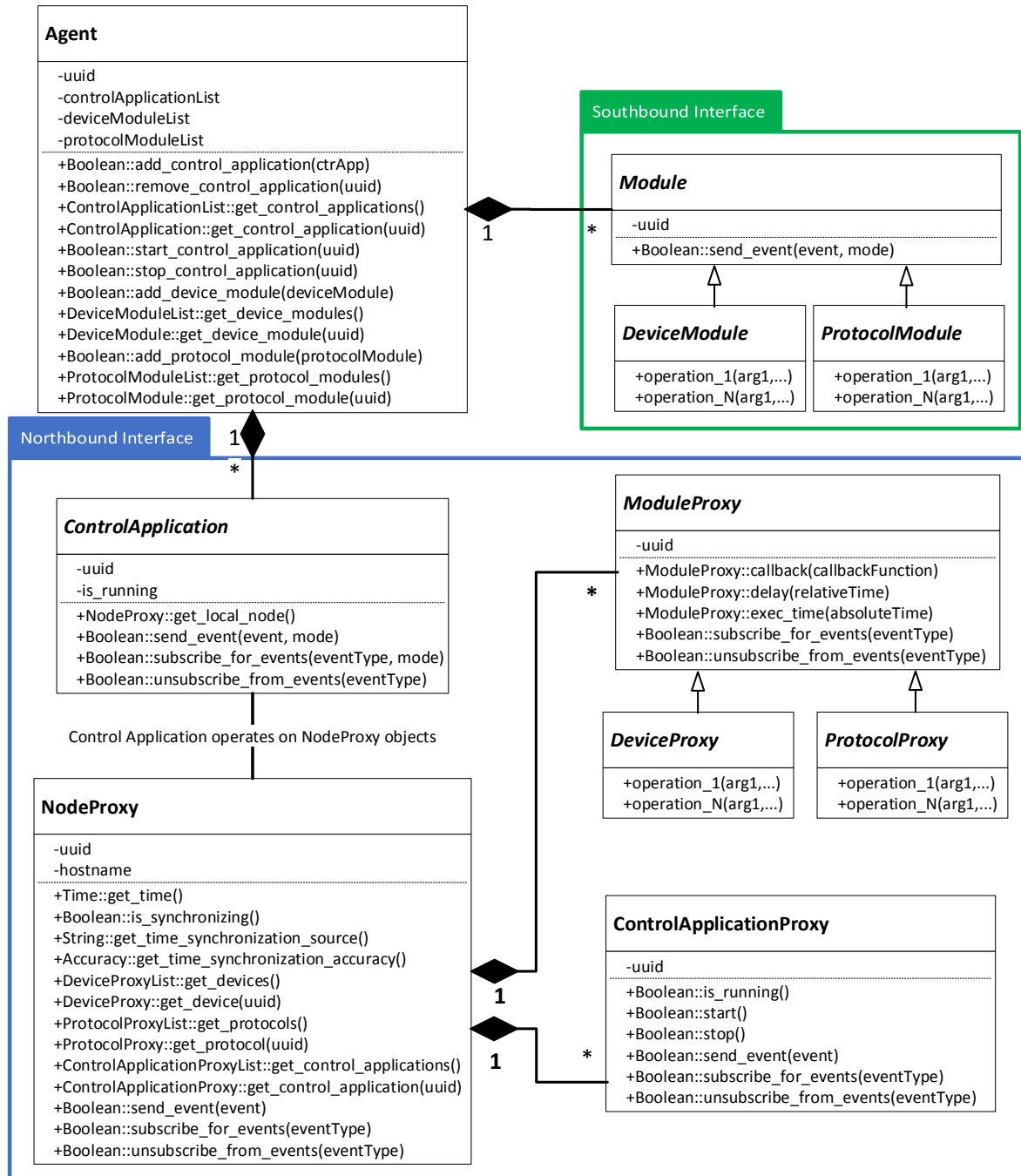


Figure 4.3: UniFlex framework (UML class diagram).

Chapter 5

Implementation Details

The UniFlex framework was prototypically implemented. Particular attention was paid to enabling code re-usability and support for different programming languages as well as possibility of usage of specialized external software libraries.

Our prototype is implemented in Python language, what makes it possible to run on multiple different host types (Linux, OpenWRT, Mac OS and Windows) and allows for rapid prototyping of control applications. 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.

An overview of framework implementation is presented in Fig.5.1. An Agent exposes *Northbound Interface* to Applications and connects to Device Modules using *Southbound Interface*. Agents communicate with each other over Broker. Each depicted entity will be described in the following subsection in more detail.

The framework is distributed as a single Python package containing implementation of Agent and Broker, and base classes for Applications and Device Modules. In other words, we provide core functionality of distributed control plane that is able to run on variety of nodes including constrained devices, and leave implementation of control applications and device modules to researchers.

5.1 Northbound Interface

Sending and subscription of events is implemented around PUB and SUB sockets available in *ØMQ* library [8] communication library.

The Remote-Procedure-Call mechanism was implemented on top of unicast event mechanism. For this purpose, we introduced two events, namely: `CommandEvent` and `ResponseEvent` that are handled internally by Agent. The `CommandEvent` event contains Calling Context, i.e. information *WHAT* (operation), *WHERE* (node and device), *HOW* (blocking, delayed, etc.) and optionally *WHEN* (point in time) is to be executed, while the `ResponseEvent` contains return value of function call. Creation of Calling Context and sending of `CommandEvent` is hidden behind execution of RPC call on device proxy objects. To facilitate building of Calling Context, we introduce following functions: *delay()*, *exec_time()* and *callback()* that may be executed on device proxy object. They are optional and may be chained together. The examples of the supported calling semantics are presented in Listing 5.1.

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

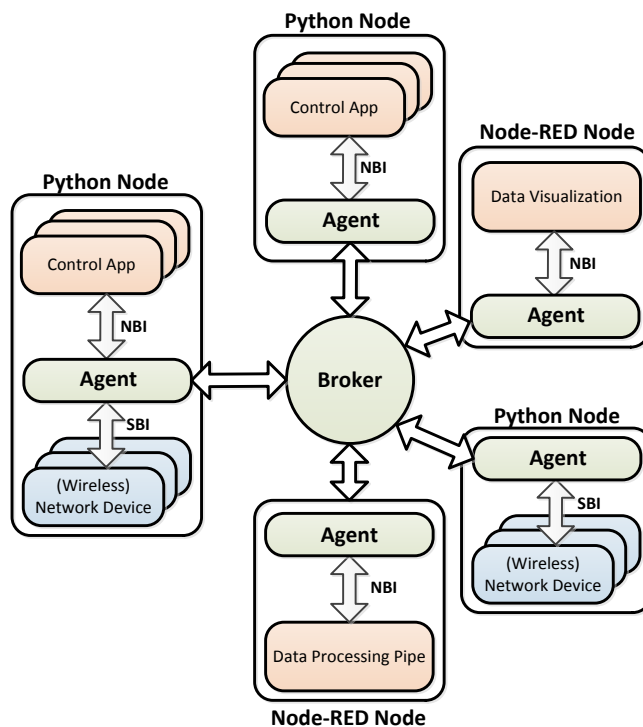


Figure 5.1: Overview of UniFlex framework implementation.

Listing 5.1: Calling examples.

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

#get device proxy from node proxy
device = node.get_device(0)

#execution of blocking call
result = device.radio.get_tx_power()

#execution of non-blocking call
device.callback(my_get_power_cb).radio.get_tx_power()

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

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


5.2 Agent

The Agent is an entity that runs Applications and Device Modules, and connects them to distributed control plane. It is mainly responsible for transferring events between local and remote (over Broker) applications and device modules. Moreover, it implements RPC mechanism as described in previous subsection.

Agent provides up-to-date information about all nodes in network to all its application. For this purpose we implemented discovery and heartbeat (based on hello messages) mechanisms. Every time an Agent discovers new node sending hello messages over Broker, it sends node unicast *NodeInformationRequest* and based on received *NodeInformationResponse* builds *NodeProxy* object. Then, the Agent notifies all its applications using *NewNodeEvent*. From this point, the Agent monitors presence of node by means of refreshing hello timer. If it expires, Agent sends *NodeLostEvent* to all its application.

Each application and device module is started within its own thread what increase performance, while preventing race-conditions.

5.3 Broker

While Agent is responsible for transporting events between Applications and Device Modules in single Node, the Broker takes care for delivering events between Nodes. For optimization, events of specific type are delivered only to those nodes that subscribed for them (i.e. listen on topic that is event's name). All Agents together with Broker constitute distributed UniFlex control framework middle-ware.

The Broker is wrapping and switching events between *XPUB* and *XSUB* sockets available in *ØMQ* library [8] library. The ZMQ itself implements mechanisms for topic agreement and message routing.

By default Pickle library is used for (de)serialization of events. This solution is very convenient, because developer do not have to care about writing parsing and serializing functions, which is wearisome and error prone task. On the other hand, it allows only for inter-Python communication and has lower performance comparing to fast serialization libraries. For inter-language communication and efficiency, we provide a possibility to define parse and serialization functions. If provided, they are used instead of Pickle. Those functions may be implemented in any format, but it has to be consistent among all nodes. Every time an agent sends/receives an event, it checks whether those functions are provided. If so, it uses them, otherwise Pickle functions are called.

5.4 Control Application

A Control Application is an entity that implements the entire or part of the network control logic. It can be as simple as a signal filter or as complicated as a mobility management unit. Application has to inherit from *ControlApplication* base class provided in UniFlex framework package.

While it is possible to subscribe at run-time to events using *subscribe_for_event()* as described in Sec. 4.2, we expect that mostly permanent (i.e. lasting for life-cycle of an application) subscriptions will be used. For this purpose, we provide *on_event* decorator that binds event notification with desired function. For example in Listing 5.2, the application subscribes for *NewNodeEvent* events and gets notification whenever a new node is discovered. Each event contains a proxy object to source

node and entity (application or device module), that an application may use to perform RPC calls and to send events.

Listing 5.2: Example of blocking RPC call

```
@on_event(NewNodeEvent)
def add_node(self, event):
    node = event.node
    device = node.get_device(0)
    txPower = device.radio.get_tx_power()
```

5.5 Southbound Interface

The Southbound Interface is realized with help of Device Modules that translate function calls from control applications into Native Device Programming Interface (NDPI) and Protocol Modules that translate them into Native Protocol Programming Interface (NPPI). We provide a `DeviceModule` and `ProtocolModule` base classes, which have interface to Agent, receive commands from control applications and execute proper function implementation according to requested commands. In order to be used within the UniFlex framework, a device's NDPI has to be wrapped with Device Module that inherits from base class. The same applies to protocol, i.e. its NPPI has to be wrapped with class inheriting from Protocol Module class. An example of function implementation is presented in Listing 5.3. Here, a `wifi_set_channel` function takes *channel* as an argument and use 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 an unified abstraction layer. In example, the function is hidden behind proper operation from UPI definition. Note that the Device and Protocol Modules are Python objects and they keep state between consecutive function calls. Finally, as the Agent takes care about parsing of function arguments and serialization of return values, developer of Device/Protocol Module do not have to care about it.

The UniFlex framework creates a `DeviceProxy` object for each Device Module and `ProtocolProxy` object for each Protocol Module present in controlled node and passes them to all control applications running on top. Such a proxy object contains all functions of device/protocol module and corresponding functions are bound together, i.e. calling function on proxy object translates in execution of function in device/protocol module object. This way, we achieve location-transparency.

We manage to make Device and Protocol Modules very *thin* and without dependencies to UniFlex core framework, so once implemented it may be used as part of the framework, but also as a standalone application.

Listing 5.3: Example implementation of a device module function

```
@bind_function(upi.radio.set_channel)
def wifi_set_channel(self, channel):
    self.channel = channel
    # set channel in wireless interface using NETLINK
    # .....
    return reponse
```

As an example, we provide two device modules for different implementation of the same wireless technology, i.e. IEEE 802.11. The `WiFiAth9kDeviceModule` translates Python function calls into NDPI of COTS Atheros 9500 wireless card (mostly into NETLINK calls), while `WiFiGnuRadioDeviceModule` maps them into parameter exposed by GNU-Radio engine. We have modified the 802.11 implementation [9] in order to expose internal parameters of Wi-Fi protocol stack for control purpose.

5.6 Deployment

We adopted *yaml* format for preparation and storing configuration for Agent. Such configuration file is loaded by Agent on it start and contains information about where to connect (Broker's URIs) and network applications as well as device modules to be started. The example configuration file is show on Listing 5.4. In order to load control application or 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* object attribute.

Listing 5.4: Example of Agent configuration file

```
agent:
  name: 'LocalWiFiController'
  info: 'Distributed WiFi controller - local part'
  sub: 'tcp://192.168.1.1:8990' #Broker X PUB
  pub: 'tcp://192.168.1.1:8989' #Broker X SUB

# List of control apps
applications:
  local_wifi_controller:
    file : local_wifi_controller.py
    class_name : MyLocalWiFiController

# List of device modules
modules:
  wifi_ath9k:
    module : module_wifi_ath9k
    class_name : WifiModule
    device : 'phy0'
  wifi_gnuradio:
    module : module_wifi_gnuradio
    class_name : WifiModule
    device : 'uhd0'
    kwargs: {UsrpServer='http://10.0.0.1:8080' }
```

5.7 Support of other Programming Languages

Since the UniFlex prototype is implemented around ZMQ library, that is available for most of programming languages, support of other programming languages like C/C++ is possible. As a proof, we provide basic support for Node-RED (see Sect. 6.2) and the Android OS.

Chapter 6

Integration with External Software Libraries

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

6.1 Python Scientific Packages

As UniFlex control applications are written using Python programming language, network developers can easily import any Python module within her network application. There exists number of scientific libraries for Python language including tools for data mining (SciPy), data processing (NumPy), machine learning (Tensorflow, PyBrain), etc.

6.2 Node-RED Integration

Flow graphs are a great abstraction model with sufficient computational power to be able to program complex control behavior. Node-RED¹ is a tool used by Internet-of-Things (IoT) community for wiring together hardware devices, APIs and online services in new and interesting ways. The UniFlex framework provides the possibility to connect to Node-RED. In a Node-RED flow graph, it is possible to subscribe to any event and also generate events, which are afterwards are processed by UniFlex control application(s).

Fig. 6.1 shows an example Node-RED flowgraph illustrating how UniFlex events can be consumed and produced in Node-RED. It subscribes to *SpectralScanSampleEvent*, calculates a moving average and publishes the result as an *AverageSpectralScanSampleEvent* event, which can afterwards be processed by another UniFlex control application.

With usage of ZMQ implementation for node.js, we provide two new Node-RED nodes: i) UniFlex Subscriber Node that connects to Broker and receives events of pre-configured type; and ii) UniFlex Publisher Node that sends events to Broker.

6.3 Mininet Integration

In order to offer the application developer an easy way to test own network control programs, before deploying them in a real testbed, our framework can be executed in Mininet [10], a container-based

¹<http://nodered.org/>

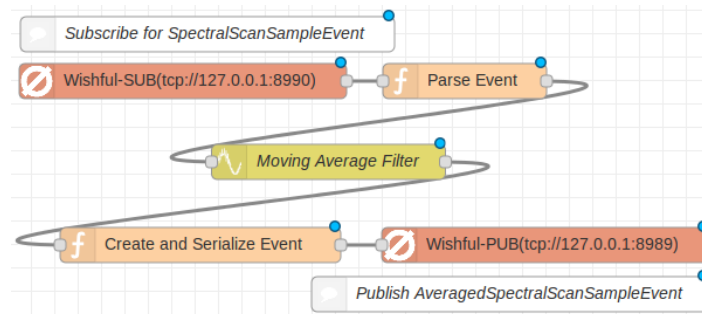


Figure 6.1: Example Node-RED flowgraph consuming and producing UniFlex events.

emulation which is able to emulate large network topologies on a single computer. Specifically, we use Mininet-WiFi [11], [12] 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 [13].

Chapter 7

Applications

In this section we present selected network apps we implemented using UniFlex framework.

7.1 Wireless Topology Monitor

Knowledge of wireless topology is of great importance when performing experiments within wireless testbeds as it provides the network programmer with a full view of the network. Usually, optimization of wireless networks is touching configuration of neighboring nodes (channel assignment, load balancing, station handover, D2D, etc).

We implemented topology monitoring as a control application in UniFlex. It consists of local application running on each wireless node that collects and aggregates information about wireless neighbors and sends it up to *Wireless Topology Monitor* application residing in central compute node. For example, in case of 802.11 infrastructure networks the local app running on each AP would periodically report the set of associated client stations.

In general the information about all wireless neighbors, a so-called hearing map, is desired, which can be obtained by the local app by means of packet sniffing over a monitor interface, i.e. from all received frames the packet headers are processed to get information about transmitter address as well as additional information like signal strength and bitrate. For efficiency reasons the local apps do not send the sniffed frames directly to the *Wireless Topology Monitor* app, instead only aggregated information like transmitter address, average signal strength and bitrate are send via events to central *Wireless Topology Monitor* app.

Based on the received information from the local network apps the *Wireless Topology Monitor* app is able to create the wireless topology graph. Note, that even within a single technology a topology may be defined in multiple ways. It may represent so called hearing map, where an edge is between nodes in communication range, or a connectivity map. While a hearing map may be used to discover handover or D2D opportunities, a connectivity map may be used to control link transmission parameters.

The *Topology Monitor* app may be parameterized to generate topology graph of required type(s). Moreover, its interfaces are technology independent, i.e. being given properly structured and described input data, it is able to generate topology graph.

The nodes in the topology graph contain unique identifiers (UUID) of UniFlex-enabled nodes, or transmitter MAC address for other nodes. Edges represent connections between nodes and their weights take value of pre-configured parameter, like bitrate, signal strength, etc. The topology graph contains also meta-data describing it. In multi-channel environments the topology graph is a set of graphs, one for each radio channel.

Any other network app can subscribe to events generated by *Topology Monitor* app, e.g. notification about topology changes. Note, the network nodes in topology graph can be directly accessed as UniFlex provides helper functions mapping node identifiers to node reference objects (if known).

7.2 Mobility Management

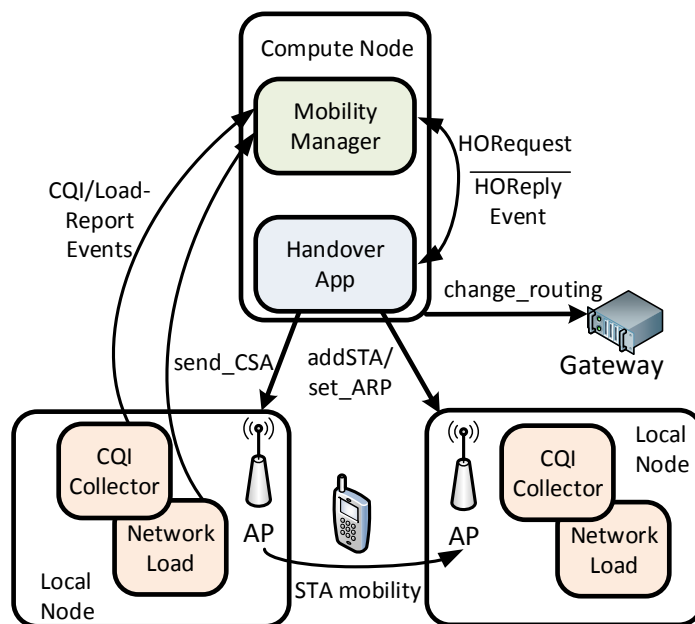


Figure 7.1: Mobility management for enterprise 802.11 networks.

Novel applications, e.g. mobile HD video, and devices, e.g. smartphones and tablets, require much better mobility support and higher QoS/QoE. Therefore, in [14, 15] we presented BIGAP, a seamless handover scheme for high performance enterprise IEEE 802.11 networks. We implemented the mobility management function of BIGAP in UniFlex. Fig. 7.1 shows the hierarchical controller architecture consisting of a central network app and multiple local network apps. On each AP two local control applications are running for collection of information, i.e. quality of the active wireless links as well as potential links to client stations in communication range which are currently being served by another co-located AP (*CQIReportEvent*). This data as well as information about the current network load at each AP (*LoadReportEvent*) is reported as event to the central mobility manager network app which decides on handover by sending out a *HORequest*. The latter is processed by the *Handover App* which performs the actual handover operation as described in [14].

7.3 Interference Management

Another known problem experienced in 802.11 networks is performance degradation due to co-channel interference because of hidden nodes. The impact can be mitigated by preventing overlapping

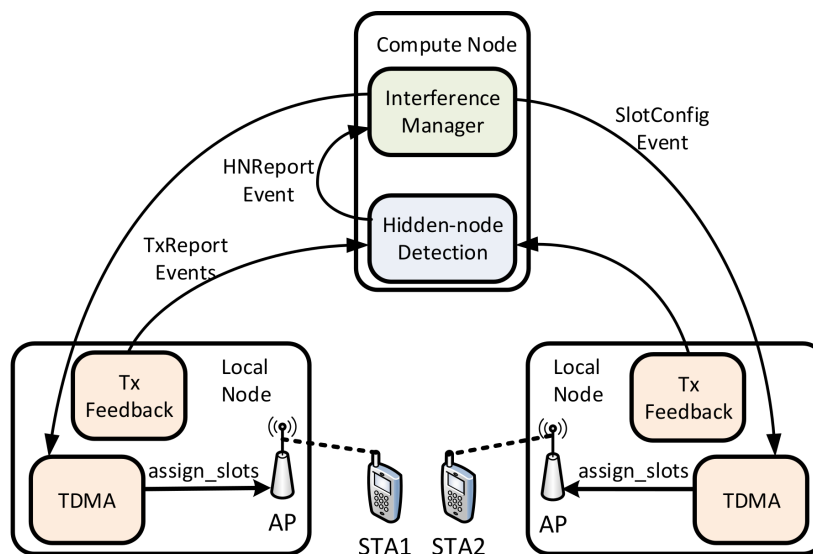


Figure 7.2: Interference management through airtime management in 802.11 networks.

transmissions (in time) of affected nodes, e.g. APs, by efficient airtime management through interference avoidance techniques at the MAC layer. We implemented interference management in UniFlex using hMAC [16]. In particular the following two features have been implemented: *i)* detection of wireless links suffering from hidden nodes and *ii)* execution of airtime management in which two wireless links suffering from the hidden node problem are getting exclusive time slots assigned.

Fig. 7.2 illustrates the developed hierarchical controller architecture. Here we have two control applications running on each AP locally due to timing constraints and efficiency reasons. The *TxFeeback* control application provides transmission feedback information like number of ARQ retries to the central *Hidden-node detection* control application which is using this information for discovery of hidden-nodes. Each pair of wireless links suffering from hidden-node is reported using *HNRReportEvent* and consumed by another central control application, *Interference Manager*, which in turn decides on the time slot configuration to be used. The actual assignment of time slots to nodes is performed by the local *TDMA* scheduler.

7.4 Device-to-Device

The efficiency of wireless infrastructure networks can be significantly improved by enabling direct device-to-device (D2D) communication between stations (Fig. 7.3). We implemented D2D functionality for 802.11 network as a central network app whose functionality can be used by other control programs. The interaction takes place by exchange of events, namely, *EnableD2D* and *D2DLink* event respectively. On receiving an *EnableD2D* event the D2D network app performs remote API function calls on the network devices of the corresponding two client stations. On one station is enables soft AP mode whereas on the other station connects to that AP. Moreover, on both clients IP connectivity is configured using the appropriate API function calls. Note, all the implementation details are hidden from the high level control design, thus simplify a controller implementation.

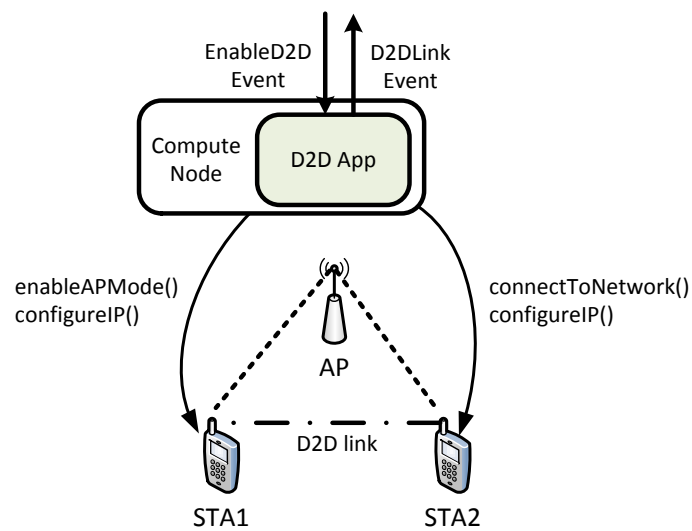


Figure 7.3: UniFlex network app enabling direct device-to-device connectivity in IEEE 802.11 networks.

Chapter 8

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.

8.1 Basic Network Operation

Observing and modifying the network state by means of executing API functions is a basic building block of UniFlex operations, its performance is of great importance on the overall system's performance. We identified latency for network state monitoring and API 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 application 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 8.1 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.

<i>Latency</i>	<i>Median</i>	<i>99 %ile</i>
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

Table 8.1: Latency for executing single blocking local API function.

Further, Table 8.2 shows the results when executing the same function remotely. Note that the network overhead for the execution of this API call is around 1600 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 used for real-world control applications. However, when using slow ARM SoCs the latency is $11 - 25 \times$ larger as compared to i7-4790 which might be insufficient. However, we argue that the UniFlex agent can be easily implemented in a low-level programming language like C.

<i>Latency</i>	<i>Median</i>	<i>99 %ile</i>
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

Table 8.2: Latency for executing single blocking remote API function.

8.2 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 [17] 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 Fig. 8.1. It takes less than 25 ms to execute a non-blocking API call on all 87 network nodes. Note, that the latency per API call decreases with the number of nodes, i.e. 2.37 ms vs. 0.24 ms 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.

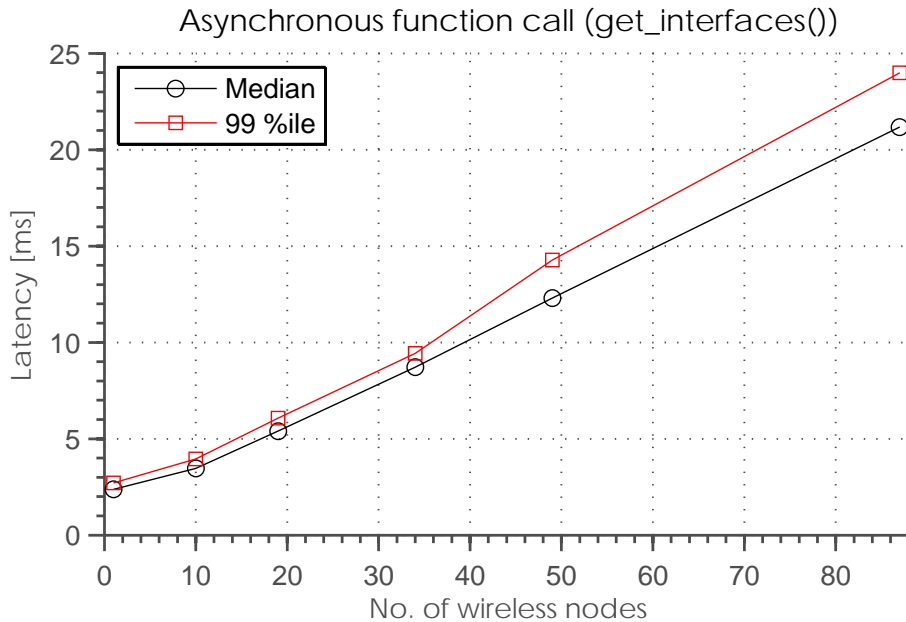


Figure 8.1: Latency for executing single non-blocking API function call on a set of nodes.

Chapter 9

Related Work

Related work falls into three categories:

Cross-layer Control: CRAWLER [18, 19] is experimentation architecture for centralized network monitoring and cross-layer coordination over different devices. ClickWatch [20] aims for simplification of experimentation of wireless cross-layer solutions implemented using the Click Modular Router [21]. Both frameworks aim to facilitate experimentation and offer possibility to control all nodes in the network from a single centralized controller. In contrast, UniFlex is more flexible as it allows distributing controller logic over multiple nodes so that time sensitive control logic can be executed directly on the network node to be controlled.

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). Some of them, like ONOS [22] and ONIX [23] 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 [4] and Kandoo [24] provide the possibility for offloading of control applications 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. In contrast UniFlex is not restricted to two levels of controllers as it allows direct communication between any controller applications. Beehive [25][26] provides interesting features like automatic distribution of network applications over network nodes. While having similar concepts, Beehive does not differentiate between control applications and device modules which are of great importance when targeting the control of heterogeneous wireless networks.

CoAP [27] proposes a vendor neutral centralized framework for configuration, coordination and management of residential 802.11 APs using an open API implemented over OpenFlow [28]. In contrast to UniFlex the CoAP API is restricted to control of 802.11 networks. Moreover, only centralized control programs are possible. OpenRF [3] 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 [29] SDN architecture for centralized spectrum brokerage in residential infrastructure Cognitive Radio networks was proposed.

General Distributed Control Platforms: ROS [30] 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 UniFlex, we are trying to achieve harmonization of multiple devices. Moreover, we also provide time scheduled execution of operations on multiple devices.

Chapter 10

Conclusions

This paper introduces UniFlex, a framework that uses SDN concepts to simplify prototyping of novel wireless networking solutions requiring cross-layer control coordinated among multiple heterogeneous wireless network nodes. It provides a rich API for management and control of operation of network devices. Control applications representing controller logic can be implemented in a local, central or distributed manner. This allows to place time-sensitive control functions close to controlled device, off-load resource hungry application to compute servers and make them work together to control the entire network.

Chapter 11

Acknowledgments

This work has been supported by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 645274 (WiSHFUL project).

Bibliography

- [1] J. Mvulla, E.-C. Park, M. Adnan, and J.-H. Son, "Analysis of asymmetric hidden node problem in IEEE 802.11 ax heterogeneous WLANs," in *Information and Communication Technology Convergence (ICTC), 2015 International Conference on*. IEEE, 2015, pp. 539–544.
- [2] V. Pejovic and E. M. Belding, "Whiterate: A context-aware approach to wireless rate adaptation," *IEEE Transactions on Mobile Computing*, vol. 13, no. 4, pp. 921–934, 2014.
- [3] 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.
- [4] S. Zhang, Y. Shen, M. Herlich, K. Nguyen, Y. Ji, and S. Yamada, "Ryuo: Using high level northbound API for control messages in software defined network," in *Network Operations and Management Symposium (APNOMS), 2015 17th Asia-Pacific*, Aug 2015, pp. 115–120.
- [5] "Switch Abstraction Interface (SAI) as part of Open Compute project." [Online]. Available: <https://github.com/opencomputeproject/OCP-Networking-Project-Community-Contributions>
- [6] P. Ruckebusch, S. Giannoulis, E. De Poorter, I. Moerman, I. Tinnirello, D. Garlisi, P. Gallo, N. Kaminski, L. DaSilva, P. Gawlowicz *et al.*, "A unified radio control architecture for prototyping adaptive wireless protocols," in *Networks and Communications (EuCNC), 2016 European Conference on*. IEEE, 2016, pp. 58–63.
- [7] C. Fortuna, P. Ruckebusch, C. Van Praet, I. Moerman, N. Kaminski, L. DaSilva, I. Tinirello, G. Bianchi, F. Gringoli, A. Zubow *et al.*, "Wireless software and hardware platforms for flexible and unified radio and network control," in *European Conference on Networks and Communications (Eu-CNC)*, 2015.
- [8] iMatix Corporation, "ZMQ - Code Connected," <http://zeromq.org/>, January 2014, accessed: 2015-08-04.
- [9] B. Bloessl, M. Segata, C. Sommer, and F. Dressler, "Towards an Open Source IEEE 802.11p Stack: A Full SDR-based Transceiver in GNURadio," in *5th IEEE Vehicular Networking Conference (VNC 2013)*. Boston, MA: IEEE, December 2013, pp. 143–149.
- [10] "Mininet." [Online]. Available: <http://mininet.org/>
- [11] "Mininet-WiFi." [Online]. Available: <https://github.com/intrig-unicamp/mininet-wifi>
- [12] 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.

- [13] “Linux wireless - mac80211.” [Online]. Available: <https://wireless.wiki.kernel.org/en/developers/documentation/mac80211>
- [14] A. Zubow, S. Zehl, and A. Wolisz, “BIG AP – Seamless Handover in High Performance Enterprise IEEE 802.11 Networks,” in *Network Operations and Management Symposium (NOMS), 2016 IEEE*, April 2016.
- [15] S. Zehl, A. Zubow, and A. Wolisz, “Bigap - a seamless handover scheme for high performance enterprise ieee 802.11 networks,” in *NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium*, April 2016, pp. 1015–1016.
- [16] —, “hMAC: Enabling Hybrid TDMA/CSMA on IEEE 802.11 Hardware,” Telecommunication Networks Group, Technische Universität Berlin, Tech. Rep. TKN-16-004, November 2016.
- [17] D. Raychaudhuri, I. Seskar, M. Ott, S. Ganu, K. Ramachandran, H. Kremo, R. Siracusa, H. Liu, and M. Singh, “Overview of the ORBIT radio grid testbed for evaluation of next-generation wireless network protocols,” in *IEEE Wireless Communications and Networking Conference, 2005*, vol. 3. IEEE, 2005, pp. 1664–1669.
- [18] I. Aktaş, O. Punñal, 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.
- [19] 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.
- [20] 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.
- [21] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, “The click modular router,” *ACM Transactions on Computer Systems (TOCS)*, vol. 18, no. 3, pp. 263–297, 2000.
- [22] 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.
- [23] 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. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1924943.1924968>
- [24] S. Hassas Yeganeh and Y. Ganjali, “Kandoo: a framework for efficient and scalable offloading of control applications,” in *Proceedings of the first workshop on Hot topics in software defined networks*. ACM, 2012, pp. 19–24.

- [25] 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. [Online]. Available: <http://doi.acm.org/10.1145/2670518.2673864>
- [26] —, “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. [Online]. Available: <http://doi.acm.org/10.1145/2890955.2890958>
- [27] 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.
- [28] 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.
- [29] 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.
- [30] 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.

Chapter 12

Appendix A – Description of UniFlex APIs

Table 12.1: Agent API

Function	Description
Boolean::add_control_application(<i>controlAppObj</i>)	Add ControlApplication object to Agent. Returns <i>True</i> if succeeded; otherwise <i>False</i>
Boolean::remove_control_application(<i>uuid</i>)	Remove ControlApplication object from Agent by its UUID. Returns <i>True</i> if succeeded; otherwise <i>False</i>
ControlApplicationList::get_control_applications()	Get all Control Application of Agent. Returns list of ControlApplication objects.
ControlApplication::get_control_application(<i>uuid</i>)	Get ControlApplication object by its UUID. Returns Control Application object.
Boolean::start_control_application(<i>uuid</i>)	Start Control Application by its UUID. Returns <i>True</i> if succeeded; otherwise <i>False</i>
Boolean::stop_control_application(<i>uuid</i>)	Stop Control Application by its UUID. Returns <i>True</i> if succeeded; otherwise <i>False</i>
Boolean::add_device_module(<i>deviceModuleObj</i>)	Add DeviceModule object to Agent. Returns <i>True</i> if succeeded; otherwise <i>False</i>
DeviceModuleList::get_device_modules()	Get all Device Modules installed in Agent. Returns list of DeviceModule objects.
DeviceModule::get_device_module(<i>uuid</i>)	Get DeviceModule object by its UUID. Returns DeviceModule object.
Boolean::add_protocol_module(<i>protModuleObj</i>)	Add ProtocolModule object to Agent. Returns <i>True</i> if succeeded; otherwise <i>False</i>
ProtocolModuleList::get_protocol_modules()	Get all Protocol Modules installed in Agent. Returns list of ProtocolModule objects.
ProtocolModule::get_protocol_module(<i>uuid</i>)	Get ProtocolModule object by its UUID. Returns ProtocolModule object.

Table 12.2: Module API

Function	Description
Boolean::send_event(<i>event, mode</i>)	Sent event using one of two modes: <i>node-broadcast</i> and <i>global-broadcast</i> . Returns <i>True</i> if succeeded; otherwise <i>False</i>

Table 12.3: Device Module API

Function	Description
operation_1(<i>arg1,...</i>)	Python wrapper for operation exposed by device.
operation_N(<i>arg1,...</i>)	Python wrapper for operation exposed by device.

Table 12.4: Protocol Module API

Function	Description
operation_1(<i>arg1,...</i>)	Python wrapper for operation exposed by protocol.
operation_N(<i>arg1,...</i>)	Python wrapper for operation exposed by protocol.

Table 12.5: Control Application API

Function	Description
NodeProxy::get_local_node()	Get NodeProxy object for local node , i.e. the one that runs Application. Returns NodeProxy object.
Boolean::send_event(<i>event, mode</i>)	Sent event using one of two modes: <i>node-broadcast</i> or <i>global-broadcast</i> . Returns <i>True</i> if succeeded; otherwise <i>False</i>
Boolean::subscribe_for_events(<i>eventType, callback, mode</i>)	Subscribe for events of specific type using one of two modes: - <i>node-broadcast</i> — subscribe for events of specific type generated on local node - <i>global-broadcast</i> – subscribe for events of specific type generated at any node in network The <i>callback</i> function will be called on reception of event. Note: If event type is not specified, application subscribes for events of all types. Returns <i>True</i> if succeeded; otherwise <i>False</i>
Boolean::unsubscribe_from_events(<i>eventType</i>)	Unsubscribe from event from specific type. If event type is not given, unsubscribe from all event types. Returns <i>True</i> if succeeded; otherwise <i>False</i>

Table 12.6: NodeProxy API

Function	Description
Time::get_time()	Get time of remote node. Returns UNIX time of remote node.
Boolean::is_synchronizing()	Check if remote node is synchronizing with some time server. Returns <i>True</i> if remote node runs time synchronization process; <i>False</i> otherwise
String::get_time_synchronization_source()	Get time synchronization source of remote node. Note: we need to check if remote node synchronizes with the same source as application's local node. Returns name of synchronization source.
Accuracy::get_time_synchronization_accuracy()	Get time synchronization accuracy. Returns time synchronization accuracy in milliseconds.
DeviceProxyList::get_devices()	Get proxy objects for all device modules installed in remote node. Returns list of Device Module Proxy objects.
DeviceProxy::get_device(<i>uuid</i>)	Get Device Module proxy object by its UUID. Returns DeviceModuleProxy object.
ProtocolProxyList::get_protocols()	Get proxy objects for all protocol modules installed in remote node. Returns list of Protocol Module Proxy objects.
ProtocolProxy::get_protocol(<i>uuid</i>)	Get Protocol Module proxy object by its UUID. Returns ProtocolModuleProxy object.
ControlApplicationProxyList::get_control_applications()	Get proxy objects for all control applications installed in remote node. Returns list of ControlApplicationProxy objects.
ControlApplicationProxy::get_control_application(<i>uuid</i>)	Get Control Application proxy object by its UUID. Returns ControlApplicationProxy object.
Boolean::send_event(<i>event</i>)	Send event to remote node in node-broadcast mode, i.e. event is delivered to node and broadcasted to all subscribed Control Applications. Returns <i>True</i> if succeeded; otherwise <i>False</i>
Boolean::subscribe_for_events(<i>eventType</i> , <i>callback</i>)	Subscribe for events of given type generated in remote node. If event type is not given, subscribe for all events generated in remote node. The <i>callback</i> function will be called on reception of event. Returns <i>True</i> if succeeded; otherwise <i>False</i>
Boolean::unsubscribe_from_events(<i>eventType</i>)	Unsubscribe from events of given type generated in remote node. If event type is not given unsubscribe from all events generated in remote node. Returns <i>True</i> if succeeded; otherwise <i>False</i>

Table 12.7: ControlApplicationProxy API

Function	Description
Boolean::is_running()	Check if remote control application is running. Returns <i>True</i> if remote app is running; otherwise <i>False</i>
Boolean::start()	Start remote control application. Returns <i>True</i> if succeeded; otherwise <i>False</i>
Boolean::stop()	Stop remote control application. Returns <i>True</i> if succeeded; otherwise <i>False</i>
Boolean::send_event(<i>event</i>)	Send event to remote application in unicast mode. Node: event will be delivered only if remote application subscribe for it. Returns <i>True</i> if succeeded; otherwise <i>False</i>
Boolean:: subscribe_for_events(<i>eventType</i> , <i>callback</i>)	Subscribe for events of given type generated in remote control application. If event type is not given, subscribe for all events generated in remote control application. The <i>callback</i> function will be called on reception of event. Returns <i>True</i> if succeeded; otherwise <i>False</i>
Boolean::unsubscribe_from_events(<i>eventType</i>)	Unsubscribe from events of given type generated in remote control application. If event type is not given unsubscribe from all events generated in remote control application. Returns <i>True</i> if succeeded; otherwise <i>False</i>

Table 12.8: ModuleProxy API

Function	Description
ModuleProxy::callback(<i>callbackFunction</i>)	<p>Execute operation of device in non-blocking mode and register callback function that will be called upon reception of return value from operation.</p> <p>If <i>callbackFunction</i> is not defined operation will be executed in non-blocking mode.</p> <p>Returns the same ModuleProxy object -> function changing.</p> <p><i>Example:</i> device.callback(<i>myCallback</i>).set_channel(11).</p>
ModuleProxy::delay(<i>relativeTime</i>)	<p>Delay execution of operation by given amount of time. It will result in non-blocking call. Use callback function to register callback.</p> <p>Returns the same ModuleProxy object -> function changing.</p> <p><i>Example:</i> device.delay(5s).set_channel(11).</p>
ModuleProxy::exec_time(<i>absoluteTime</i>)	<p>Schedule execution of operation in remote device module. It will result in non-blocking call. Use callback function to register callback. Absolute time is UNIX time.</p> <p>Returns the same ModuleProxy object -> function changing.</p> <p><i>Example:</i> device.exec_time(<i>execTime</i>).set_channel(11).</p>
Boolean:: subscribe_for_events(<i>eventType, callback</i>)	<p>Subscribe for events of given type generated in remote device module. If event type is not given, subscribe for all events generated in remote device module. The <i>callback</i> function will be called on reception of event.</p> <p>Returns <i>True</i> if succeeded; otherwise <i>False</i></p>
Boolean:: unsubscribe_from_events(<i>eventType</i>)	<p>Unsubscribe from events of given type generated in remote device module. If event type is not given unsubscribe from all events generated in remote device module.</p> <p>Returns <i>True</i> if succeeded; otherwise <i>False</i></p>

Table 12.9: DeviceProxy API

Function	Description
<code>operation_1(arg1,...)</code>	Python wrapper for remote operation exposed by device.
<code>operation_N(arg1,...)</code>	Python wrapper for remote operation exposed by device.

Table 12.10: ProtocolProxy API

Function	Description
<code>operation_1(arg1,...)</code>	Python wrapper for remote operation exposed by device.
<code>operation_N(arg1,...)</code>	Python wrapper for remote operation exposed by device.
