Programming Software–Defined Wireless Networks

Roberto Riggio*, Karina Mabell Gomez,*, Tinku Rasheed*,
Julius Schulz–Zander†, Slawomir Kuklinski‡, Mahesh K. Marina§
*CREATE-NET, Trento, Italy; Email: rriggio@create-net.org, kgomez@create-net.org, trasheed@create-net.org
†TU–Berlin, Berlin, Germany; Email: julius@inet.tu-berlin.de
‡Orange Polska, Warsaw, Poland; Email: slawomir.kuklinski@orange.com
§The University of Edinburgh, Edinburgh, UK; Email: mahesh@ed.ac.uk

***Abstract*—Programming a mobile network requires to account for multiple complex operations, such as allocating radio resources and monitoring interference. Nevertheless, the current Software–Defined Networking ecosystem provides little support for mobile networks in term of radio data–plane abstractions, controllers, and programming primitives. Starting from the consideration that WiFi is becoming an integral part of the 5G architecture, we present a set of programming abstractions modeling three fundamental aspects of a WiFi network, namely state management of wireless clients, resource provisioning, and network state collection. The proposed abstractions hide away the implementation details of the underlying wireless technology providing programmers with expressive tools to control the state of the network. We also describe a proof–of–concept implementation of a Software–Defined Radio Access Network controller for WiFi networks and a Python–based Software Development Kit leveraging the proposed abstractions. The resulting platform can be effectively leveraged in order to implement typical control tasks such as mobility management and traffic engineering as well as applications and services such as multicast video delivery and/or dynamic content caching.**

I. Introduction

Mobile networks are currently at a cross road. The dramatic adoption by end–users of smart–phones and tablets is fueling an increasing demand for mobile data access. Operators are coping with this trend by: (i) deploying denser cellular infrastructures; (ii) increasing the available bandwidth; and (iii) exploring massive MIMO solutions. WiFi is also being leveraged in order to relieve the cellular network from at least part of the burden generated by modern data–hungry mobile applications and services; in this sense, WiFi based access networks can be seen as an integral part of the next-generation mobile RANs. Notice that, the raw capacity of the wireless channel is not the actual bottleneck: LTE can deliver up to 300 Mb/s, the recent 802.11ac WiFi amendment can provide up to 1.5 Gb/s, and LTE–Advanced will exceed 600 Mb/s for a single user[1]. The real pitfalls lie in the increased interference between cells and in the signaling. The latter issue is particularly relevant in LTE and LTE–Advanced networks that, being designed with sparse networks in mind, heavily rely on inter base stations (BSes) signaling for their operations.

Likewise, the transition from an *homogeneous* Radio Access Network (RAN) composed of BSes that belong to the same type and power class to an *heterogeneous* RAN where low–power nodes (pico–, femto–cell) overlap in coverage with the macro–cell is raising several resource management challenges. This ever increasing network complexity is driving operators toward a virtualization of network functionality which calls for a paradigm shift from a hardware–based approach where a certain function, e.g. Mobility Management is implemented by a dedicated box, to a software–based approach where the same functions are provided by high volume servers and switches.

However, albeit several attempts at applying similar concepts to cellular [3], [4], [5] and to WiFi [6], [7] RANs can already be found, few or none programming abstractions specifically tailored for wireless networks in general and for mobile RANs in particular have been proposed. In a true SDN philosophy, programmers shall be exposed with just enough information about the state of the network to implement the task at hand and shall be able to focus on defining the expected behavior of the network rather than dealing with technology–dependent implementation details. The latter requirement is fundamental if features and services are to be ported seamlessly across different link–layer technologies.

In this work we take a step in this direction by focusing on the programming abstractions needed for managing WiFi networks. The proposed abstractions tackle wireless client state management, resource provisioning, and network state collection. In the rest of the paper we will show how these abstractions completely cover the three fundamental elements that compose a network control loop, namely: collecting the network state, specifying the network behavior, and updating the network configuration. We realized the abstractions in a proof–of–concept Software–Defined RAN (SD–RAN) Controller and in a Python–based Software Development Kit (SDK). The platform allows developers to implement features such as mobility management and traffic engineering as *Network App* running on top of the *SD–RAN Controller*. Moreover, the Software–Defined principles on which the platform is based allow also for advanced network virtualization functionality paving the way to virtual operator scenarios and end–to–end service delivery.

Although our discussion will focus mainly on WiFi due to the fact that the proof–of–concept currently supports only this technology, we will show how this is just a particular case and that the proposed abstractions can actually serve the requirements of current and future cellular technologies.

[1]Rates for LTE/LTE–A are relative to class 5 UEs with 4 MIMO streams [1], while for WiFi we refer to 802.11ac stations supporting 80 MHz channels and 4 MIMO streams [2].

The next section introduces the three programming abstractions proposed in this work together with the rationale behind their design. The *SD–RAN Controller* implementation details together with an overview of the main SDK features are provided in Sec. III. Section IV reports on the scalability of the proposed abstractions as implemented in our SDK. Finally, we discuss the related work in Sec. V and then we draw our conclusions in Sec. VI.

II. Wireless Networks Abstractions

Programming mobile RANs requires identifying how network resources are exposed (and represented) to software modules written by developers and how software modules can affect the network state. Although, OpenFlow [8] provides a practical forwarding abstraction for flow–switched networks, it has been argued that programming current networks using OpenFlow is equivalent to program applications in assembler, i.e. the interface is too low–level and exposes the programmers with too many implementation details. As a result in the last years we have witnessed a mushrooming of efforts aiming at providing developers with higher level interfaces toward their SDN [9], [10], [11], [12], [13], [14], [15].

The works above, however, aim at enabling programmability in wired networks and typically rely on OpenFlow as data–plane control API. Conversely, in this work we specifically set to investigate which kind of abstractions can be effectively used to implement control and coordination tasks in mobile RANs. As a matter of fact, a straightforward extension of current programming techniques would fail to capture the peculiarities of the radio environment. In particular, the flow abstraction on which OpenFlow relies does not account for: (i) the stochastic nature of wireless links (which are not equivalent to ports in Ethernet switches); (ii) the resource allocation granularity (the flow abstraction is too coarse for mobile networks); and (iii) the significant heterogeneity in the link and radio layer technologies (state management for network elements can differ significantly across currently deployed RAN technologies).

The proposed abstractions address the three fundamental elements that compose a network control loop, namely, collecting the current state of the network (*Network Status*), implementing control and management tasks (*Network Specification*) and disseminating the new configuration to the network elements (*Network Reconfiguration*). Let us analyze the requirements that each of the above elements impose on the abstractions:

- *Network Status*. The abstractions shall allow networks developers to gather the status of the network using high–level querying primitives. Such information shall include network statistics and topology changes. The network programmer shall not be exposed to the system–level details of polling network elements, aggregating statistics, and detecting network events.
- *Network Specification*. The abstractions shall allow developers to leverage the view of the network state that best suits the task at hand. Multiple software modules may be in charge of defining different aspects of the network such as fault management, load balancing, mobility management.
- *Network Reconfiguration*. The abstractions shall allow programmers to *describe* the desired behavior of the network leaving to the underlying run–time system the task of implementing it by configuring the individual network elements. Similarly, programmers shall not be exposed with technology–dependent implementation details such as state management.

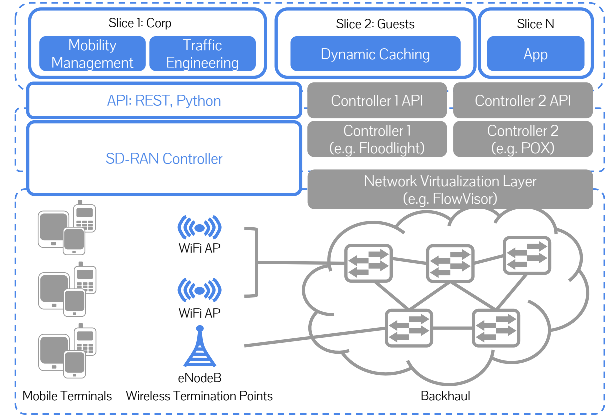


Fig. 1: The reference network architecture.

Figure 1 sketches the reference network architecture and introduces the terminology used throughout the paper. We name Wireless Termination Points (*WTP*s), the physical devices that form the RAN providing User Equipments (UEs) with wireless connectivity. *WTP*s basically coincide with Access Points (APs) in a WiFi network or eNodeBs (eNBs) in an LTE network. A secure channel connects the *WTP*s to a remote *SD–RAN Controller*. *Network App*s run on top of the *SD–RAN Controller* in their own slice of resources and exploit the programming primitives trough either a RESTful interface or a native Python API (bindings for other programming languages can be easily added). Notice that the slicing methodology, albeit supported by our platform, is not described in this work due to space constraints. Finally, although OpenFlow is a candidate technology, the abstractions proposed in our work do not rely on it and are effectively backhaul agnostic.

In the next subsections we will introduce three key abstractions for mobile RANs, namely the *Light Virtual Access Point* (*LVAP*) abstraction, the *Resource Pool* abstraction, and the *Interference Map* abstraction. Figure 2 depicts the relationship between *LVAP*, *Resource Pool*, and *Interference Map* using an UML class–diagram. Borrowing from the 3GPP terminology, a *Resource Block* represents the minimum chunk of wireless resources that can be assigned to a wireless client while the *LVAP* represents the state of a UE scheduled on a set of *Resource Block*s. Similarly, both the *LVAP*s and the *WTP*s support a set of *Resource Block*s, named *Resource Pool*. Finally, a relationship exists between each pair *LVAP*/*Resource Block* modeling the link quality between the two entities in

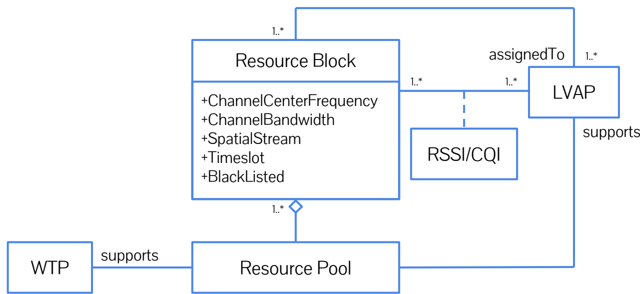


Fig. 2: The abstractions class–diagram.

terms of, for example, RSSI. This latter relationship constitute the *Interference Map*.

A. The Light Virtual Access Point

Different link layer technologies, or as a matter of fact even different releases of the same technology, can differ significantly in how UEs state is handled. For example authentication mechanisms changed significantly over the lifespan of the IEEE 802.11 family of standards. Similar consideration can be made with regards to QoE (802.11e) or fast handover (802.11r). Nevertheless exposing the programmer with the implementation details of the technology being used, such as association/authentication procedures, would severely limit the adoption of a certain *Network App*. On the contrary what we want to achieve is true *Network App* portability from one RAN to another, e.g. from a WiFi network to an heterogeneous WiFi/LTE deployment.

The LVAP abstraction builds on top of [6] which provides a high–level interface through which the state of a wireless terminal can be manipulated. The implementation of such an interface is required to handle all the technology–dependent details such as association, authentication, handover, and resource scheduling. In particular, a UE attempting to join the network, will trigger the creation of a new *LVAP*. Such *LVAP* has an ID that is specific to the newly associated UE (in a WiFi network the *LVAP* can be thought as a Virtual AP with its own BSSID). As a result each WTP (or better one or more *Resource Block*s supported by the WTP, see next section) will host as many *LVAP*s as the number of UEs currently communicating with it. Removing an *LVAP* from a *WTP* and instantiating it on another *WTP* effectively results in a handover.

B. The Resource Pool

Programming mobile RANs mandates for a way to expose the programmer with a consistent view of the network resources. The fundamentals of wireless communications basically calls for two main family of strategies to allocate resources in a wireless network: scheduled access and random access. In the former case, resources for a wireless link are allocated in the time, frequency, and space[2] domain. In the latter case, a common random access scheme for medium access is used by all participating wireless clients in order to reduce collisions. WiFi belongs to the latter family and exploits CSMA/CA as random access scheme.

The increasing demand for mobile data access is mandating the deployment of denser and heterogeneous RANs composed of macro–cells, small-cell (often overlapping in space with the macro–cell), and WiFi hotspots. The *Resource Pool* abstraction goes beyond the concept of cell and exposes the programmer with the collective resources in time, frequency, and space that are available in the whole RAN. The minimum allocation unit in the *Resource Pool* is the *Resource Block* and is identified by a frequency band, a time interval, a spatial stream, and the *WTP* at which it is available. The *Resource Pool* is exposed to the programmer through a set P where each *Resource Block* $b \in P$ is a 3–tuple $\langle f, t, s \rangle$, where f is the frequency band, t is the time slot, and s is the spatial stream. A frequency band is a 2–tuple $\langle c, b \rangle$ where c and b are, respectively, the center frequency and the bandwidth. Notice that the proposed model does not forbids the same *Resource Block* to be assigned to multiple *LVAP*s in that this could in general result in a valid resource allocation scheme if, for example, the *LVAP*s are sufficiently separated in space or if suitable ICIC (Inter–Cell Interference Coordination) schemes are employed.

For example, the *Resource Pool* made available by a legacy 802.11g AP tuned on channel 1 would be represented by the tuple $((1, 20), \infty, 1)$. Where 1 is the channel, 20 is the bandwidth (in MHz) and 1 is the single spatial stream supported. Similarly an 802.11n AP supporting 2 spatial streams and 40 MHz–wide channels would be represented by the tuple $((36, HT40), \infty, 1)$, $((36, HT40), \infty, 2)$. Notice that no time dimension is provided or, more precisely, the *Resource Block*s are allocated for all the time modeling the fact that in WiFi networks random access is used as the access scheme. Finally, *Resource Block*s can also be *blacklisted* preventing applications from using them. This could be, for example, the case of highly interfered blocks in an LTE network.

The same formulation is also exploited to model the resource requests coming from the UEs or more precisely from the *LVAP*s mapping those UEs. For example, an *LVAP* resource request could be represented by the following tuple $((1, 20), \infty, 1)$. This allows us to express resource allocation problems as an intersection between the *Resource Block*s available in the network and the *Resource Block*s supported by a client. Information on the link quality experience by the requesting *LVAP* on the matching *Resource Block*s can be used to further filter the set of candidate *Resource Block*s according to application–level parameters. A non–empty set of *Resource Block*s signifies that a valid solution for the resource allocation problem has been found. Conversely an empty set represents a case where a valid resource allocation could not be found.

Notice that, the final *Resource Block*s set could be composed of multiple *Resource Block*s possibly scheduled at different *WTP*s and on different frequency bands/timeslots. The support for such scenario depends on the actual implementation of the *LVAP* interface. For example, in a WiFi network, an *LVAP* mapping an 802.11n client supporting two spatial streams will

[2]Notice that, for spread spectrum–based technologies, such as UMTS, also the code space shall be considered.

Network (P_1)	*LVAP* (P_2)	Intersection
$W_1((6, HT20), \infty, 1)$ $W_1((6, HT20), \infty, 2)$ $W_1((6, 20), \infty, 1)$ $W_2((1, HT20), \infty, 1)$ $W_2((1, HT20), \infty, 2)$ $W_3((1, 20), \infty, 1)$	$L_1((1, HT20), \infty, 1)$ $L_1((1, HT20), \infty, 2)$ $L_1((1, 20), \infty, 1)$	$W_2((1, HT20), \infty, 1)$ $W_2((1, HT20), \infty, 2)$ $W_3((1, 20), \infty, 1)$

TABLE I: Network and LVAP *Resource Pool*s.

accept up to two *Resource Block*s from the same *WTP* and on the same channel. Conversely, in the more general case of an LTE–A based network, an *LVAP* will accept multiple *Resource Block*s possibly scheduled at different BSes and on different frequencies modeling the technique known as Cooperative Multi–Point, or CoMP [16]. Thus the *Resource Pool* entails advantages for handling the interference, reducing energy consumption and performing effective scheduling and load–balancing. Moreover providing the network developer with a global view of the network resources simplifies the implementation of tasks such as interference mitigation and topology control as well as energy saving or cell zooming.

A simple resource allocation scenario for a WiFi network is summarized in Table I where P_1 is the network *Resource Pool* and P_2 is the *Resource Pool* supported by an *LVAP*. In this case it is easy to seen that the intersection $P_1 \cap P_2$ produces a non–empty set composed of three *Resource Block*s scheduled at two different *WTP*s (W_1, W_2). The limitations of the actual link layer technology, i.e. WiFi, does not allow the control logic to schedule the *LVAP* on both *WTP*s. As a result the final resource allocation decision shall be taken according to the channel quality experienced by the *LVAP* on the matching *Resource Block*s and/or on the specific application–level requirements.

Notice that operations such as *Resource Block* scheduling as well as rate adaption cannot be decoupled from the *WTP* and moved to the remote *SD–RAN Controller* in that they are characterized by strict latency requirements. As a result we envision such operations to be left within the scope of the single *WTP*s leaving the *SD–RAN Controller* in charge of slowly changing resource management operations.

C. The Interference Map

Links in a mobile RAN are highly volatile, fading and multi–path can severely affect the channel SNIR and must be taken into consideration, together with the requirements coming from the application layer in order to define a proper resource allocation scheme. For example, disaster recovery scenarios may prefer to allocate a minimum amount of resources to all the clients even if this results in a poor utilization of the medium. On the other hand commercial deployments aims at the best network utilization even if this could result in poor performance for the users at the edges of the cell.

The *Interference Map* abstraction provides network programmers with a full view of the network state in terms of channel quality between *Resource Block*s and *LVAP*s. The *Interference Map* essentially consist in a (typically) sparse matrix R where each entry $R(m, l)$ is the channel quality between the *Resource Block* m and the *LVAP* l. In WiFI networks this information can be achieved trough passive measurements on the received traffic as well as by leveraging dedicated hardware such as the spectrum scanners commonly found in recent WiFi chipsets. Conversely in an LTE/LTE–A network, UEs can be asked to perform active channel quality measurements on a set of *Resource Block*s using standard indicators, such as the CQI in the downlink and the Sounding Reference Signals (SRSs) in the uplink.

As matter of fact, from the perspective of serving UEs, it is not important to which infrastructural *WTP* they are attached but what communication QoS they can obtain which, at the physical layer, essentially translates into bandwidth and SNIR. The *Interference Map* allows the control logic to *reason* about the channel quality experienced by the various *LVAP*s and to assign resources accordingly.

III. Implementation Details

To verify the flexibility of the proposed abstractions, we designed and implemented: (i) an SD–RAN controller, (ii) a programmable WiFi data–plane, and (iii) a Python–based SDK. This section briefly summarizes each component. More information on the software platform can be found online[3]. The platform has been successfully used to replicate the technological demonstrations presented in [17], [18].

A. SD–RAN Controller

The SD–RAN controller implementation leverages the Tornado Web Server as the web framework [19]. The main reason for choosing Tornado is its non–blocking network I/O which allows to continue serving incoming requests while the others are being processed. Here follows some of the main features of the SD–RAN controller.

- Slicing: Multiple logical virtual networks, or *Pool*s, can be instantiated on top of the *SD–RAN Controller*. Each *Pool* is characterized by its own SSID and a set of *WTP*s. Each *Network App* can be instantiated within one or multiple *Pool*s and can only affect the state of *LVAP*s that are associated to that *Pool*. Similarly, users can *opt–in* a certain *Pool* by associating to its SSID.

- Soft State: The only persistent information stored at the controller are the UEs' authentication method (currently only ACLs are supported) and the list of currently defined *Pool*s. *LVAP*'s state is kept within the network in a distributed fashion and is synchronized when the *WTP* connects to the *SD–RAN Controller*. As a result the *SD–RAN Controller* can be hot–swapped with another instance without affecting the active clients. Moreover, the network itself can still function in its last known state even if the controller becomes unavailable.

- Modular Architecture: With the exception of the logging subsystem, every other task supported by the controller is implemented as a *plug–in* (i.e., a Python module) that can be loaded at runtime. Examples of such *plug–in* are the modules implementing the data–path control protocol, the RESTful web interface and the mobility/load–balancing applications.

[3] Available at: http://empower.create-net.org

B. Wireless Termination Points

Each *WTP* consists of two components: one OpenvSwitch [20] instance managing the communication over the wired backhaul; and one Click modular router [21] instance implementing a WiFi AP's data–path. Click is a framework for writing multi–purpose packet processing engines and is being used to implement just the *WTP*s/UEs frame exchange while all the decision logic is implemented at the SD–RAN controller. Click is also used to implement the packet counters and the RSSI triggers described in the next subsection. Communications between Click and the SD–RAN controller take place over a persistent TCP connection. The Click instance can run over standard WiFi devices, in particular our deployment exploited a mix of PCEngines ALIX (x86) and Gateworks Cambria (ARM) embedded platforms all running the OpenWRT operating system (Chaos Calmer r42609).

C. Software Development Kit

A Python–based SDK mapping the abstractions introduced in Sec. II to Python constructs is also made available to application developers (see Table II). In this section we shall briefly summarize, using some practical examples, some of the SDK's most interesting features. Notice that since the SDK is specifically tailored for the WiFI technology, *Resource Block*s are identified by just the 2–tuple $\langle f, b \rangle$.

The *LVAP* is exposed to the programmer through a Python object mapping the properties to functions. These properties are: (i) the *Resource Block*(s) on which the *LVAP* is currently scheduled, (ii) the list of *Resource Block*s supported by the *LVAP*, and (iii) the counters tracking the incoming/outgoing traffic. Such an interface allows programmers to fetch the *Resource Block*(s) a certain LVAP is currently scheduled at, by accessing the corresponding field of an *LVAP* object. Similarly, performing a handover is as simple as assigning an *LVAP* a new list of *Resource Block*s to the same field.

- Resource Management: The following Python routine assigns an *LVAP* to a random *Resource Block* whose RSSI to the *LVAP* is greater than or equal to -65 dB:

```
def handover(lvap, wtps):
    """ Handover the LVAP to a WTP with
    an RSSI higher that -65dB. """

    # Initialize the Resource Pool
    pool = ResourcePool()

    # Update the Resource Pool with all
    # the available Resourse Blocks
    for wtp in wtps:
        pool = pool | wtp.supports

    # Select matching Resource Blocks
    matches = pool & lvap.supports

    #  Filter Resource Blocks by RSSI
    valid = [block for block in matches
             if block.rssi[lvap] >= -65]

    # Perform the handover
    new_block = valid.pop() if valid else None
    lvap.assigned_to = new_block
```

The method above accepts as input two parameters, an *LVAP* object (`lvap`) and a list of *WTP* objects (`wtps`). The method initializes the network *Resource Pool* with the *Resource Block*s available at every *WTP*. Then, an intersection between the network *Resource Pool* and the *LVAP Resource Pool* is computed. The resulting set is then traversed in order to filter–out the *Resource Block* whose RSSI to the *LVAP* is below a certain threshold (-65dB in this example). Finally, one random *Resource Block* matching the above condition is assigned to the *LVAP*. Notice that, for the sake of simplicity, error handling is omitted in this example. For example, if the valid *Resource Block* set is empty it is up to the application to decide to either lower the RSSI threshold or to handover the *LVAP* to best available *WTP* regardless of the link quality. Finally, it is worth stressing that, the one reported above is just an example aiming at showing the basic capabilities of the SDK and that the programmer has the flexibility to seek for an *optimal* configuration considering criteria like throughput, fairness, reliability etc.

- Querying: The packets counters allow programmers to track the traffic exchanged by a certain *LVAP* and to use binning in order to aggregate such information by frame length (useful in wireless network due to the fact the short packets incur in an heavier transmission overhead). For example:

```
C = packets_count(bins=[512, 1472, 8192],
                  lvap='11:22:33:44:55:66',
                  every=5000,
                  ssid='Guests')
```

The statement above instructs the controller to track the packets transmitted and received by a certain *LVAP* and to aggregate the information into the specified bins. The WTP currently hosting the LVAP is polled every 5000ms. It is also possible to issue a single query by specifying -1 as polling period. The counters' current state can be accessed with:

```
>>>C.tx_samples
[60, 10, 0]
```

Meaning that the *LVAP* transmitted 60, 10, and 0 packets smaller or equal to respectively 500, 1460, and 8192 bytes. Similarly, `bytes_count` allows to track the bytes transmitted and received by the *LVAP*.

-Interference Tracking: The *Interference Map* allows the developers to schedule *LVAP*s in the *Resource Block*s that best satisfy their traffic requirements. Moreover, *Network App* can react to changes in the network conditions by setting callbacks that are triggered when the measured RSSI for a given set of *LVAP*s verifies a specified condition. For example:

```
T = rssi(lvaps='11:22:33:44:55:66',
         relation='LT',
         value=-70,
         ssid='Guests')
T.callback = self.callback
```

The callback above is triggered the first time the RSSI of the specified *LVAP* goes above -70 dB at any *WTP* in network. After the trigger has fired the first time and as long as the RSSI remains greater than -70 dB, the callback method is not called again by the same *WTP*, however the same callback

Abstractions	*Python Construct*	*Example*
Resource Block	ResourceBlock (object)	`block = ResourceBlock(lvap, 1, 'L20', False)` `block = ResourceBlock(wtp, 1, 'L20', False)`
Resource Pool	ResourcePool (extends `set`)	`wtp.supports = ResourcePool()` `wtp.supports.add(block)`
Union (∪)	|	`pool = wtp1.supports | wtp2.supports`
Intersection (∩)	&	`matches = lvap.supports & pool`
Filtering (∀)	List Comprehensions	`valid = [block for block in matches if block.rssi[lvap] >= -65]`
Assignment (←)	=	`lvap.assigned_to = valid`

TABLE II: Programming abstractions in the Python–based SDK.

can be triggered by multiple *WTP*s. In order to detect RSSIs that are going below the -70 dB threshold another trigger must be created. *LVAP*s are matched by logical *AND*, thus specifying `FF:FF:FF:FF:FF:FF` as *LVAP* will trigger the callback when the RSSI of any *LVAP* at any *WTP* is higher than -70 dBm.

D. Use Case: Mobility Management

In this section we describe a simple *Mobility Manager* which leverages the SDK's RSSI tracking primitives in order to detect when the link quality experienced by a UE is deteriorating and thus a handover must be triggered. Moreover, the *Mobility Manager* periodically checks if a better handover opportunity exists even if the channel quality experience by the *LVAP*s is still acceptable. The complete *Mobility Manager* code is reported below.

```
class MobilityManager(BasePoolWorker):

  def __init__(self, pool, period):

    BasePoolWorker.__init__(self, pool, period)

    # Register an RSSI trigger for all LVAPs
    self.rssi = self.rssi(lvaps='ff:ff:ff:ff:ff:ff',
                          relation='LT',
                          value=-70,
                          callback=self.low_rssi)

  def low_rssi(self, lvap, wtp, trigger, rssi):
    """ Callback on low RSSi. """
    handover(lvap, self.wtps())

  def loop(self):
    """ Periodic job. """
    for lvap in self.lvaps():
      handover(lvap, self.wtps())


def launch(pool, period=None):
    """ Initialize the module. """

    return MobilityManager(pool, period)
```

The *Network App* above is essentially a Python module that can be loaded/unloaded at run–time without affecting the network operations. Each module is required to implement a `launch` method called when the *Network App* is loaded in order to perform initialization tasks. Notice that an optional `shutdown` method, called when a module is unloaded, can also be defined in order to perform clean–ups tasks. The parameters accepted by the `launch` method depend on the actual *Network App*. In this case just two parameters must be specified, namely the *Pool* on which the module shall operate and the control loop interval.

As it can be seen, the *Mobility Manager* consists of a single Python class instantiated during the module initialization. The class implements just two methods: (i) the *loop* method which periodically checks if an *LVAP* shall be handed–over to another *WTP*; and (ii) the *callback* method which is invoked when the RSSI between a pair of *LVAP*/*WTP* is going below a certain threshold (-70dB in this example). In such a case the *handover* routine is invoked. Finally, an RSSI trigger matching all *LVAP*s in the network in created bt the *class constructor*.

Notice that, in order to reduce the verbosity, error handling, logging code, and modules import statements have been omitted. Moreover, this *Mobility Manager* does not take into account the actual traffic generated by the UEs and as result could overload some *WTP*s. A more refined implementation would also implement load balancing functionality spreading the traffic across the network.

IV. Evaluation

In this section we will show how the proposed programming abstractions can significantly improve network performance in an enterprise WLAN. Moreover we will provide a preliminary assessment of the platform overall scalability. It is worth noticing that, however, the main goal of this work is to investigate the fundamental requirements and trade–offs in programming a Software–Defined RAN and that, as a consequence, the current implementation is not intended to deliver scale nor optimal performance. The evaluation setup is composed of a single wireless client (a DELL D630 laptop) and two *WTP*s based on the PCEngines ALIX embedded platform and running the latest version of the OpenWRT operating system (Chaos Calmer r42609). Performance tests are conducted with the wireless NICs operating in 11a mode and tuned on a channel not shared with any other WiFi network. Iperf [22] is used in order to generate synthetic traffic.

A. Resource Management

In order to assess the scalability and the effectiveness of the *LVAP* abstraction we implemented a simple *Network App* which periodically handover the wireless client between the two *WTP*s. The *WTP*s are 5m apart and the wireless client is

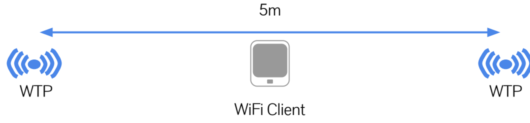


Fig. 3: Network setup for the resource management scenario.

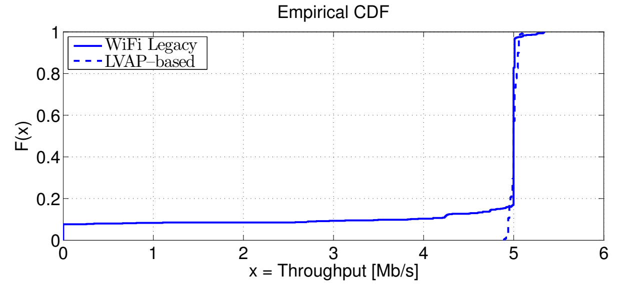
equidistant from the two *WTP*s. The network setup is sketched in Fig. 3. Notice that, in addition to the *SD–RAN Controller* presented in this work, also an OpenFlow controller, namely Floodlight, has been used in order to pre–configure the *WTP* forwarding tables before each handover.

As baseline scenario we consider a standard WiFi network where client mobility is emulated by progressively reducing the transmission power of the serving AP. We remind the reader that, in a standard WiFi network, handovers are triggered by the wireless clients and the network has no way of controlling wireless clients' mobility. As a result, a WiFi client that sees a progressively decreasing RSSI, will, at some point, handover to another AP (if available). Such process however is not deterministic and, as we will see, can take a variable amount of time.

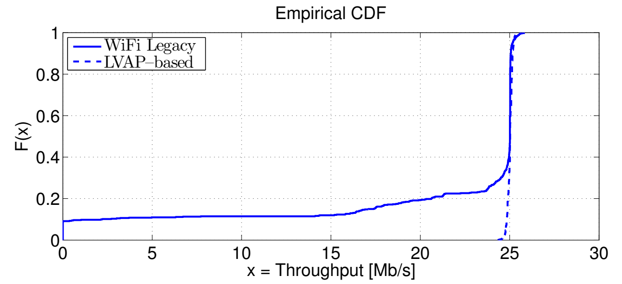
The traffic is generated at the wireless client toward a fixed node which shares the backhaul with the two *WTP*s and consists in a single UDP flow with constant payload size (1472 bytes) and constant bitrate (5, or 25 Mb/s). Figure 4 shows the distribution of the goodput at the receiver's side when the client is performing a handover every 10s in both the baseline and the *LVAP*–based scenarios. The figure is the result of a 600s–long measurement with goodput samples taken every 1s. As it can be seen, in the baseline scenario almost 20% and 40% of the samples are below the target bitrate for, respectively, the 5 and the 25 Mb/s flows.

Figure 5 shows the instantaneous bitrate in both the baseline and the SDN scenario for 5 Mb/s flow. As it can be seen, the non–deterministic and uncoordinated nature of the standard WiFi handover can lead to a significant throughput degradation. In particular during our measurements we noticed that, when a client fails to re–associate with the new AP during an handover a full network scan followed by the DHCP exchange is triggered. This procedure can take up to 1-2 seconds to complete. Notice also that, being triggered by the client, this procedure can effectively lead to scenarios where the clients is bouncing between two APs. On the other hand, the *LVAP* abstractions allows for completely seamless handovers.

We also tested the *LVAP*–based handover performance with TCP traffic. In this case the client generates a saturated TCP flow toward the remote host. Figure 6 shows the throughput at the receiver's side for an increasing handover rate from 1 handover every two seconds up to 10 handovers per seconds. As it can be seen, the link capacity is not significantly affected by the handover rate. Results are the average of 10 runs. Each run was 600–seconds long. 95% confidence intervals are shown as error–bars.

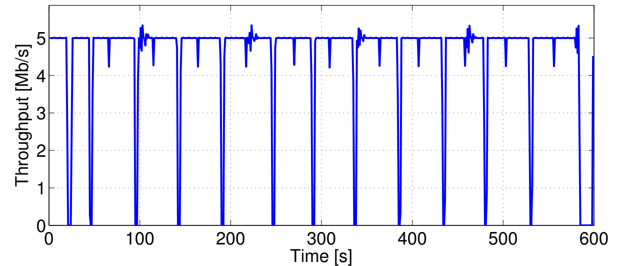


(a) 5 Mb/s flow.

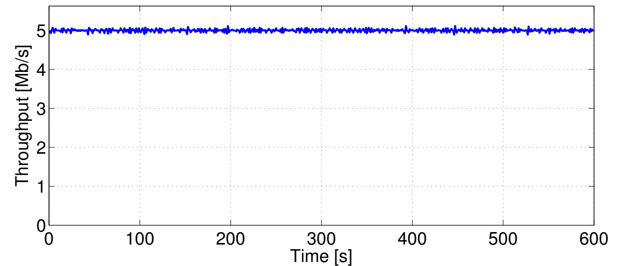


(b) 25 Mb/s flow.

Fig. 4: Distribution of the goodput at the receiver's side when the client is performing a handover every 10s.



(a) WiFi Legacy.



(b) *LVAP*–based handover.

Fig. 5: Instantaneous the goodput at the receiver's side when the client is performing a handover every 10s.

B. Querying

In order to demonstrate the scalability of the querying subsystem we implemented a sample application which periodically request an *LVAP* uplink and downlink statistics. The polling period is increased from 1 request every two seconds to 10 requests per second. Figure 7a shows the throughput attained by the wireless client under an increasing polling rate.

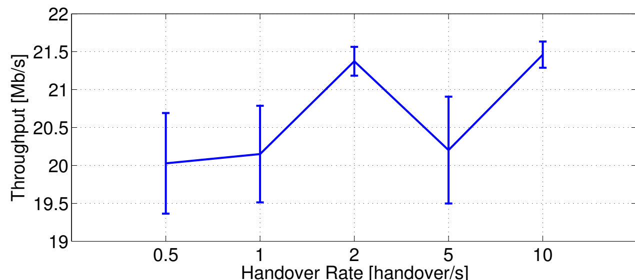
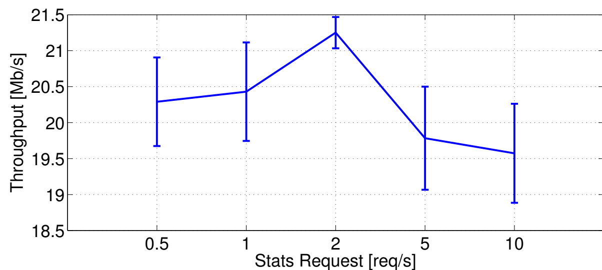
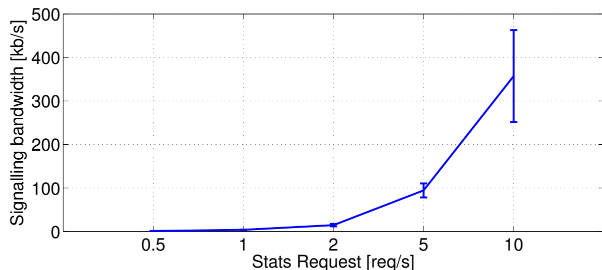


Fig. 6: Wireless client throughput Vs. an increasing handover rate using the *LVAP*–based handover. The client is generating a saturated TCP stream toward a remote host. The client throughput is not affected by the handover rate.



(a) Throughput.



(b) Signaling channel bandwidth.

Fig. 7: Scalability of the statistics subsystem. The client is generating a saturated TCP stream toward a remote host. The client throughput is not affected by the polling frequency (a) while the required signaling bandwidth scales with the square of the polling rate (b).

As it can be seen the client throughput is only marginally affected by statistics requests. This result is obtained through the use of shredded counters at the *WTP* and non–blocking I/O between *WTP*s and controller. Figure 7b, on the other hand, shows the average bandwidth used for the signaling channel under an increasing polling rate. Albeit the bandwidth increases with the square of the polling rate, the actual value does not exceed 500 kb/s even in the most extreme case of 10 requests per second. Notice that this traffic is typically carried over the wired link interconnecting the *WTP* with the *SD–RAN Controller*. As a result even a 1 Gb/s Ethernet connection could easily accommodate up to 2000 *LVAP*s.

C. Interference Tracking

Tracking a client RSSI across the network in real–time can be significantly expensive [23]. Moreover, if very generic triggers are defined at the application layer, i.e. triggers that match any address in the the network, the lookup time per fame would scale as c^2n, where c is the number of wireless clients per *WTP* and n is the number of *WTP*s in the network (c queries to check for each of the cn clients in the network). Since performing this matching over the frame fast–path is not realistic, *WTP*s keep track of the RSSI of last frame received from each neighbor. An exponentiation moving average is used in order to take into account old measurements. This neighbors table is traversed periodically in order to check for matches over the defined triggers. Only if a match is found a trigger to the controller is generated.

V. Related Works

In this section, the most relevant works that investigate SDN applied to wireless networks are summarized.

- Wireless LAN: In [24], Murty et al. present a software architecture to address the problem of extensibility in wireless LANs, by defining a set of APIs for clients and APs to be managed by a centralized controller. The controller can control the network's behavior based on a global network view and enact a rich set of policies. Odin [6] is an SDN framework for controlling and managing enterprise WLANs. Odin allows network applications and services to be deployed as *Networks Apps* on top of a centralized controller.

- Mobile Networks: In [25], [26] the authors argue that SDN can simplify the design and management of mobile networks, while enabling new services. This work is extended in [4], where SoftCell, a scalable architecture that supports fine–grained policies for mobile devices in cellular core networks, is presented. Cloud–RAN (C–RAN) [5] aims at making deployment of 5G systems cheaper, faster and more flexible. C–RAN is composed by a system of distributed antennas connected using high bandwidth links to servers responsible for their baseband processing. A distributed hierarchical architecture for heterogeneous RANs based on OpenFlow is presented in [27]. Similarly, in [28] an OpenFlow–based control plane for LTE/EPC is presented. SoftRAN [3] proposes a fundamental refactoring of the cellular RAN by introducing a big base station abstraction mapping the resources of all BSes in a certain area.

Although the above works demonstrates continuing interest at addressing the complexity of future mobile networks using SDN principles, none of them put the focus on providing programmers with high–level interface to control the next–generation mobile RANs. Some of these works tackles the challenges from the architectural perspective [25], [26], other aims at a refactoring of the RAN/EPC [5], [27], [28]. Some attempts at providing a more high–level view of the network can be found [4], [6] however these works either address the challenges at the EPC or they focus only on wireless clients state management.

Closest by goals and principles to our work is SoftRAN [3] in which the authors introduce the big base station abstractions in order to address the challenges raised by densification of the cellular RAN. But they do not elaborate on the primitives to be exposed to the programmers, the focus of our work. Finally, several recent works have put their focus on applying high–level programming primitives and techniques to SDN [9], [10], [11], [12], [13], [14], [15]. Their target however is to enable programmability in wired networks typically relying on OpenFlow as data–plane control API. In contrast our aim in this work is focused on modeling the most critical aspects of a WiFi–based RAN, namely wireless client state management, resource allocation, and interference management and at exposing them to the network programmer through a set of technology agnostic programming primitives.

VI. Conclusions

Starting from the consideration that the current SD–RAN ecosystem in terms of controllers, data–plane abstractions, and programming interfaces lacks the tools to properly control and manage heterogeneous and dense mobile RANs, we proposed a set of programming primitives aiming at providing the developers with expressive tools to control the state of the network while hiding away the implementation details of the underlying technology.

The proposed abstractions catch the three fundamental elements that compose a network control loop, namely collecting the network status, specifying the desired behavior and disseminating the new configuration. Our architecture specifically accounts for the stochastic nature of the wireless links and for the significant heterogeneity in term of radio layer technologies that characterize modern RANs. This essentially translates into separating policies, i.e. how to select the optimal transmission rate, from mechanisms, i.e. the knobs to be turned in order to obtain the desired behavior, and to putting the former in the hands of the network programmers who are not necessarily network experts while leaving the latter to equipment vendors.

A proof–of–concept *SD–RAN Controller* targeting WiFi–based RANs and a Python–based SDK have also been developed to asses the flexibility of the proposed abstractions. A preliminary evaluation campaign has shown that the proposed abstractions can actually be implemented in scalable fashion. As future work we plan to validate the abstractions in a cellular scenario. Moreover we intend also to progress on the representation of the network and on investigating the limits boundaries between control and management in mobile RANs.

References

[1] S. Sesia, I. Toufik, and M. Baker, *LTE, The UMTS Long Term Evolution: From Theory to Practice*, ser. Wiley InterScience. Wiley, 2009.

[2] L. Verma, M. Fakharzadeh, and S. Choi, "WiFi on steroids: 802.11AC and 802.11AD," *Wireless Communications, IEEE*, vol. 20, no. 6, pp. 30–35, December 2013.

[3] A. Gudipati, D. Perry, L. E. Li, and S. Katti, "SoftRAN: software defined radio access network," in *Proc. of ACM HotSDN*, 2013.

[4] X. Jin, L. Li, L. Vanbever, and J. Rexford, "SoftCell: scalable and flexible cellular core network architecture," in *Proc. of ACM CoNEXT*, 2013.

[5] M. HadZialic, B. Dosenovic, M. Dzaferagic, and J. Musovic, "Cloud-RAN: innovative radio access network architecture," in *Proc. of IEEE ELMAR*, 2013.

[6] L. Suresh, J. Schulz-Zander, R. Merz, A. Feldmann, and T. Vazao, "Towards programmable enterprise WLANS with Odin," in *Proc. of ACM HotSDN*, 2012.

[7] G. Bianchi, P. Gallo, D. Garlisi, F. Giuliano, F. Gringoli, and I. Tinnirello, "MAClets: active MAC protocols over hard-coded devices," in *Proc. ACM CoNEXT '12*.

[8] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: enabling innovation in campus networks," *SIGCOMM Comput.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.

[9] T. L. Hinrichs, N. S. Gude, M. Casado, J. C. Mitchell, and S. Shenker, "Practical declarative network management," in *Proc. of ACM WREN*, 2009.

[10] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing software-defined networks," in *Proc. of USENIX NSDI*, 2013.

[11] A. Voellmy and P. Hudak, "Nettle: Taking the sting out of programming network routers," in *Proc. of ACM PADL*, 2011.

[12] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," *SIGPLAN Not.*, vol. 46, no. 9, pp. 279–291, Sep. 2011.

[13] A. Voellmy, H. Kim, and N. Feamster, "Procera: A language for high-level reactive network control," in *Proc. of ACM HotSDN*, 2012.

[14] A. K. Nayak, A. Reimers, N. Feamster, and R. Clark, "Resonance: Dynamic access control for enterprise networks," in *Proc. of ACM WREN*, 2009.

[15] C. Monsanto, N. Foster, R. Harrison, and D. Walker, "A compiler and run-time system for network programming languages," in *Proc. of ACM POPL*, 2012.

[16] D. Lee, H. Seo, B. Clerckx, E. Hardouin, D. Mazzarese, S. Nagata, and K. Sayana, "Coordinated multipoint transmission and reception in lte-advanced: deployment scenarios and operational challenges," *Communications Magazine, IEEE*, vol. 50, no. 2, pp. 148–155, 2012.

[17] P. L. Suresh, J. Schulz-Zander, R. Merz, and A. Feldmann, "Demo: programming enterprise WLANs with odin." in *Proc. of ACM SIGCOMM*, 2012.

[18] R. Riggio, C. Sengul, L. S. J. Schulz-Zander, and A. Feldmannx, "Thor: Energy programmable wifi networks," in *Proc. of IEEE INFOCOM*, 2013.

[19] "Tornado Web Server." [Online]. Available: http://www.tornadoweb.org/

[20] B. Pfaff, J. Pettit, K. Amidon, M. Casado, T. Koponen, and S. Shenker, "Extending networking into the virtualization layer." in *Prof. of ACM Hotnets*, 2009.

[21] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Trans. Comput. Syst.*, vol. 18, no. 3, pp. 263–297, Aug. 2000.

[22] "Iperf." [Online]. Available: http://iperf.sourceforge.net/

[23] S. Rayanchu, V. Shrivastava, S. Banerjee, and R. Chandra, "Fluid: Improving throughputs in enterprise wireless lans through flexible channelization," in *Proc. of ACM MobiCom*, 2011.

[24] R. Murty, J. Padhye, A. Wolman, and M. Welsh, "Dyson: An architecture for extensible wireless lans," in *Proc. USENIX ATC '10*.

[25] L. Li, Z. Mao, and J. Rexford, "Toward software-defined cellular networks," in *Proc. of EWSDN*, Oct 2012, pp. 7–12.

[26] H. Ali-Ahmad, C. Cicconetti, A. de la Oliva, M. Draxler, R. Gupta, V. Mancuso, L. Roullet, and V. Sciancalepore, "CROWD: an SDN approach for DenseNets," in *Proc. of EWSDN*, Oct 2013, pp. 25–31.

[27] G. Sun, G. Liu, H. Zhang, and W. Tan, "Architecture on mobility management in openflow-based radio access networks," in *Proc. of IEEE GHTCE*, 2013.

[28] S. Ben Hadj Said, M. Sama, K. Guillouard, L. Suciu, G. Simon, X. Lagrange, and J.-M. Bonnin, "New control plane in 3GPP LTE/EPC architecture for on-demand connectivity service," in *Proc. of IEEE CloudNet*, 2013.