Programming Abstractions for Software–Defined Wireless Networks

Programming Abstractions for Software-Defined Wireless Networks

Roberto Riggio, *Member, IEEE*, Mahesh K. Marina, *Senior Member, IEEE*, Julius Schulz-Zander, Slawomir Kuklinski, *Member, IEEE*, and Tinku Rasheed, *Member, IEEE*

***Abstract*—Software-Defined Networking (SDN) has received, in the last years, significant interest from the academic and the industrial communities alike. The decoupled control and data planes found in an SDN allows for logically centralized intelligence in the control plane and generalized network hardware in the data plane. Although the current SDN ecosystem provides a rich support for wired packet–switched networks, the same cannot be said for wireless networks where specific radio data-plane abstractions, controllers, and programming primitives are still yet to be established. In this work, we present a set of programming abstractions modeling the fundamental aspects of a wireless network, namely state management, resource provisioning, network monitoring, and network reconfiguration. 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 present a Software-Defined Radio Access Network Controller for Enterprise WLANs and a Python–based Software Development Kit implementing the proposed abstractions. Finally, we experimentally evaluate the usefulness, efficiency and flexibility of the platform over a real 802.11-based WLAN.**

***Index Terms*—Network management, programming abstractions, software-defined wireless networks, WLANs.**

I. Introduction

MANAGING modern wireless networks is an increasingly complex task. To cope with the dramatic increase in data traffic generated by modern mobile applications, operators are currently deploying denser and heterogeneous mobile networks. Network management platforms are then required to deal with a multitude of technologies characterized by significantly diverse protocols stacks and vendor-specific interfaces. This growing complexity calls for novel abstractions and tools to control and manage both the wired and the wireless parts of a network. Software-Defined Networking (SDN) has already delivered on such programmatic interfaces to the switching fabric. While several attempts have been made to apply similar concepts to the cellular [1]–[3] and to the WiFi [4], [5] networks, there is very little by way of programming abstractions specifically tailored for wireless networks. 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. Meeting the latter requirement is vital 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 definition of reusable high level programming abstractions for managing wireless networks. Said abstractions tackle state management, resource provisioning, network monitoring, and network reconfiguration. Moreover, we also introduce a proof-of-concept implementation of a *SD-RAN Controller* realizing the proposed primitives and a Python-based Software Development Kit (SDK) allowing programmers to create and deploy new applications and services as *Network App*s over it. Although our discussion will focus mainly on WiFi due to the fact that the proof-of-concept currently supports only this technology, we believe that the proposed abstractions can also be useful in meeting the requirements of current and future cellular technologies. This paper extends our previous works [6], [7] on programming abstractions for Enterprise WLANs by: (i) extending the APIs with additional interference modeling primitives; (ii) studying the usefulness of the primitives in realistic settings; and (iii) reporting on a number of *Network App*s implemented using the SDK.

The entire software stack, named *EmPOWER*, encompassing: a data-path implementation for 802.11-based networks, the reference *SD-RAN Controller*, and the Python-based SDK have been released under a permissive BSD-like license for academic use. These components along with documentation and tutorials are available on the official web-site.[1]

The next section introduces the 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, respectively, Section III and Section IV. Section V reports on the benefit of the proposed abstractions for seamless handovers and on the characterization of the signaling overhead of the system. A number of WLAN applications and services implemented using our SDK are presented in Section VI. Finally, we discuss the related work in Section VII and then we draw our conclusions in Section VIII.

Manuscript received February 8, 2015; revised March 23, 2015; accepted March 24, 2015. Date of publication March 30, 2015; date of current version June 12, 2015. The associate editor coordinating the review of this paper and approving it for publication was L. Granville.

R. Riggio and T. Rasheed are with CREATE-NET, 38123 Trento, Italy (e-mail: rriggio@create-net.org; trasheed@create-net.org).

M. K. Marina is with the School of Informatics, University of Edinburgh, Edinburgh EH8 9Y, U.K.

J. Schulz-Zander is with the Technische Universität Berlin (TU–Berlin), 10623 Berlin, Germany.

S. Kuklinski is with Orange Polska, 96-100 Warsaw, Poland.

Digital Object Identifier 10.1109/TNSM.2015.2417772

[1] http://empower.create-net.org/

II. Wireless SDN Abstractions

In this section we identify the common aspects concerning resource management in wireless access networks. We will do so by first acknowledging the difference between control and management and then by analyzing the requirements imposed by wireless networks.

Programming wireless networks 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 packet-switched networks, it has been argued that programming current networks using OpenFlow is equivalent to programming applications in assembler, i.e., the interface is too low-level and exposes the programmers with too many implementation details. As a result, we have witnessed a plethora of efforts in recent years aimed at providing developers with higher level interfaces to their SDN [9]–[15].

The works above, however, aim at enabling programmability in *wired* networks and typically rely on OpenFlow as the data-plane control API. In contrast, our aim is to investigate which kind of abstractions can be effectively used to implement control and coordination tasks in wireless networks. In 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 wireless networks); and (iii) the significant heterogeneity in the link and radio layer technologies (state management for network elements can differ significantly even across currently deployed technologies).

In this work we draw a clear line between network control and network management. The former (control) deals with fast timescale operations executed by the elements at the edges of the network, such as scheduling in LTE networks or transmission rate adaptation in WiFi networks. The latter (management) is in charge of checking whether the operating conditions for a certain policy are still met, and, if this is not the case, of reconfiguring or replacing the policy. For example a certain scheduling algorithm could be optimized for a uniform distribution of clients across the sectors of a mobile cell. However, if the clients distribution is not uniform, a different policy could be required [16].

A. Overview

The abstractions proposed in this work address four control aspects of wireless networks, namely: state management, resource allocation, network monitoring, and network reconfiguration. Let us analyze the requirements imposed by each of them on the control plane:

- *State management*. 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. Programmers shall not be exposed with technology-dependent details such as how to implement handover for a particular link-layer technology.
- *Resource allocation*. The abstractions shall allow developers to leverage a global view of the network resources. The programming abstractions shall be general enough to accommodate for resource allocation models ranging from random access to scheduled access.
- *Network monitoring*. The abstractions shall allow network 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 reconfiguration*. The programming abstractions shall go in the direction of separating fast timescale operations running at the edges of the network, i.e., near the air interface, from tasks running at the controller.

Fig. 1 sketches the reference network architecture and introduces the terminology used throughout the paper. We use the term Wireless Termination Points (*WTP*s) to refer to the physical devices that form the Radio Access Network (RAN) providing clients with wireless connectivity. *WTP*s basically coincide with Access Points (APs) in a WiFi network or eNodeBs (eNBs) in a LTE network. A secure channel connects the *WTP*s to the remote *SD-RAN Controller*. The *SD-RAN Controller* can run multiple virtual networks, or slices, on top of the same physical infrastructure. A slice is a virtual network with its own set of *WTP*s. *Network App*s run on top of the *SD-RAN Controller* in their own slice of resources and exploit the programming primitives through either a REST API or a native Python API (bindings for other languages can be added). The *SD-RAN Controller* ensures that a *Network App* is only presented a view of the network corresponding to its slice. Notice that, in this architecture, the term *Network App* is used to address any consumer of the *SD-RAN Controller* API. Examples includes, but are not limited to, OpenDaylight and Openstack. Finally, although OpenFlow is a candidate backhaul technology, the abstractions proposed in our work do not rely on it and are effectively backhaul agnostic.

In the next subsections we will introduce four key abstractions for wireless networks, namely the *Light Virtual Access Point* (*LVAP*) abstraction, the *Resource Pool* abstraction, the *Channel Quality and Interference Map* abstraction, and the *Port* abstraction. Fig. 2 depicts the relationship between *LVAP*, *Resource Pool*, *Channel Quality and Interference Map*, and *Port* using an UML class-diagram. Here, 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 wireless client scheduled on a set of *Resource Block*s. Both *LVAP*s and *WTP*s support a set of *Resource Block*s, named *Resource Pool*. The *Port* abstraction models the dynamic and reconfigurable characteristics of the link between *WTP* and *LVAP* on a set of *Resource Block*s. A relationship exists between *LVAP*s and *WTP*s and between *WTP*s modeling the link quality between the two entities. These relationships are captured by the *Channel Quality Maps*.

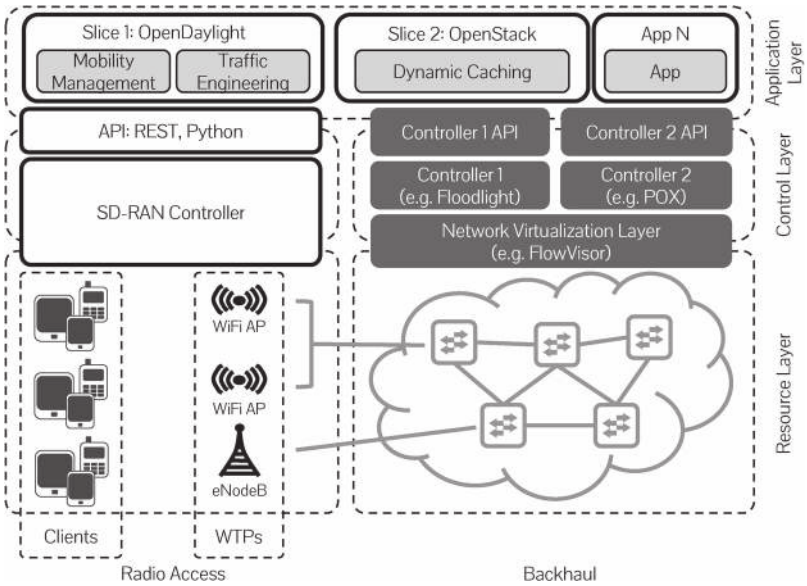

Fig. 1. The reference network architecture [6].

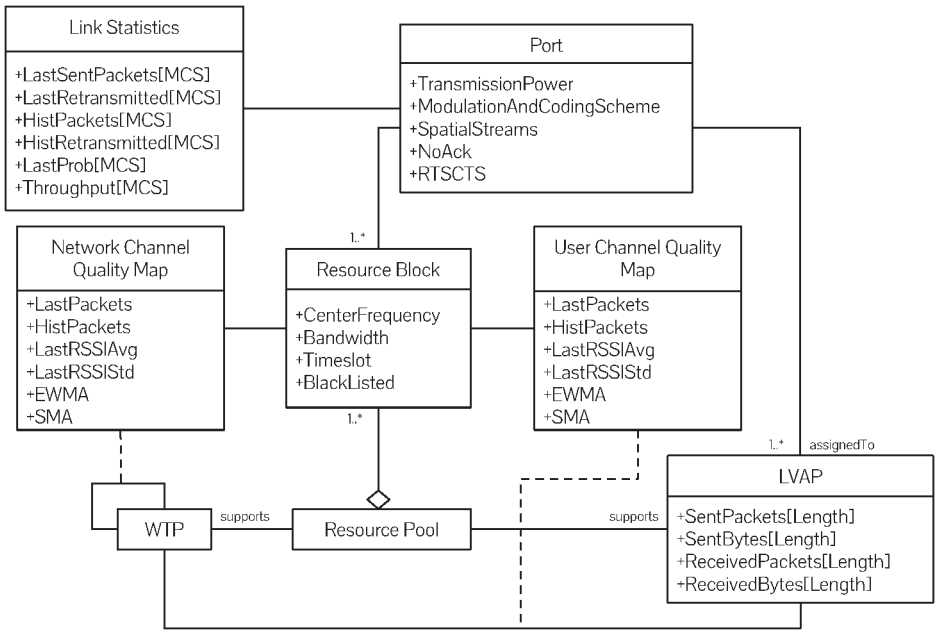

Fig. 2. Relationship between the different programming abstractions.

B. 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 a client's state is handled. For example, QoS and handover management changed significantly over the lifespan of the IEEE 802.11 family of standards. Nevertheless exposing the programmer with the implementation details of the technology being used would severely limit the adoption of a certain solution. 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 mixed WiFi/LTE deployment.

The *LVAP* abstraction [4], [17] provides a high-level interface for wireless clients state management. The implementation of such an interface handles all the technology-dependent details such as association, authentication, handover, and resource scheduling. A client attempting to join the network, will trigger the creation of a new *LVAP*. Specifically, in WiFi, an *LVAP* is a per-client virtual access point which simplifies network management and introduces seamless mobility support, i.e., the *LVAP* abstraction captures the complexities of the IEEE 802.11 protocol stack such as handling of client associations, authentication, and handovers. More specifically, every newly received probe request frame from a client at a *WTP* is forwarded to the controller which may trigger the generation of a probe response frame through the creation of an *LVAP* at the requesting *WTP*. The *LVAP* will thus become a potential candidate AP for the

client to perform an association. The controller can also decide whether the network has sufficient resources left to handle the new client and might suppress the generation of the *LVAP*. Similarly each *WTP* will host as many *LVAP*s as the number of wireless clients that are currently under its control. Such *LVAP* has an ID that is specific to the newly associated client (in a WiFi network the *LVAP* can be thought as a Virtual AP with its own BSSID). Removing a *LVAP* from a *WTP* and instantiating it on another *WTP* effectively results in a handover.

C. Resource Pool

Programming wireless networks mandates for a way to expose the programmer with a consistent view of the network resources. Broadly speaking, there are 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] domains. In the latter case, a common random access scheme for medium access is used by all participating wireless clients to reduce collisions. LTE belongs to the former family and uses OFDMA as (scheduled) medium access scheme while WiFi belongs to the latter family and exploits CSMA/CA as (random) medium access scheme.

The *Resource Pool* abstraction is tightly coupled with the *LVAP* abstraction and goes in the direction of abandoning the concept of cell and exposing the network programmer with the collective resources in time, frequency, and space that are available in the network. The minimum allocation unit in the *Resource Pool* is the *Resource Block* identified by a frequency band, a time interval, and the *WTP* at which it is available. The *Resource Pool* is exposed to the programmer through a set P where each *Resource Block* $i \in P$ is a 2-tuple $\langle f, t \rangle$, where f is the frequency band, and t is the time slot. 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 forbid 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 Inter-Cell Interference Coordination (ICIC) schemes are employed. Similarly, wireless networks using random access protocols, such as CSMA/CA, effectively schedule multiple transmissions on the same resources in frequency and time with the aid of suitable back-off and retransmission schemes to handle collisions.

For example, the *Resource Pool* made available by an 802.11n AP tuned on channel 36 and supporting 40 MHz-wide channels is represented by the tuple $((36, HT40), \infty)$. The prefix HT is used to indicate that this band supports the High Throughput MCS. *Resource Block*s can also be *blacklisted* preventing applications from using them. This could be the case of highly interfered blocks in an LTE network.

The same formulation is also exploited to model the resource requests coming from the clients or more precisely from the *LVAP*s mapping those clients. For example, a resource request could be represented by the following tuple $((1, 20), \infty)$. 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/requested by a client. Information on the link quality experienced 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 intersection set of *Resource Block*s signifies that a valid solution for the resource allocation problem has been found.

TABLE I
NETWORK AND LVAP RESOURCE POOLS

Network (P_N)	*LVAP* (P_L)	Intersection
$W_1((6, HT20), \infty)$	$L_1((36, HT20), \infty)$	$W_1((36, HT20), \infty)$
$W_1((36, HT20), \infty)$	$L_1((48, HT20), \infty)$	$W_2((36, HT20), \infty)$
$W_2((1, HT20), \infty)$	$L_1((54, HT20), \infty)$	
$W_2((36, HT20), \infty)$		
$W_3((11, 20), \infty)$		
$W_3((36, 20), \infty)$		

Notice that, the final 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 LTE network, an *LVAP* could accept multiple *Resource Block*s possibly scheduled at different eNBs and on different frequencies modeling the technique known as Cooperative Multi-Point, or CoMP [18]. This model also effectively decouples uplink and downlink allowing clients to be scheduled at different *WTP*s on the uplink and on the downlink directions (if the feature is supported by the link-layer technology).

An example of a simple resource allocation scenario for a WiFi WLAN is shown in Table I. Here P_N is the network *Resource Pool* and P_L is the *Resource Pool* supported by an *LVAP*. It is easy to see that the intersection $P_N \cap P_L$ produces a non-empty set composed of two *Resource Block*s scheduled at two different *WTP*s (W_1, W_2). Due to the fact that WiFi does not allow to schedule one *LVAP* on more than one *WTP*, the final resource allocation decision will be a single *Resource Block* selected using criteria such as the channel quality experienced by the *LVAP* on the matching *Resource Block*s and/or the specific application-level requirements.

D. Channel Quality and Interference Map

From the perspective of an *LVAP*, it is not important to which *WTP* it is attached but what communication QoS it can obtain. Such information translates, at the physical layer, into the transmission efficiency of the radio channel linking interference with parameters such as packet error rate. The *Channel Quality and Interference Map* allows the control logic to *reason* about the channel quality and interference experienced by clients and to assign resources accordingly.

The *Channel Quality Map* abstraction provides network programmers with a full view of the network state in terms of channel quality between *LVAP*s and *WTP*s over the available *Resource Block*s. Let $G = (V, E)$ be a directed graph, where $V = V_{WTP} \cup V_{LVAP}$ is the set of $v_1 = |V_{WTP}|$ *WTP*s and $v_2 = |V_{LVAP}|$ *LVAP*s in the network, and E is the set of edges

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

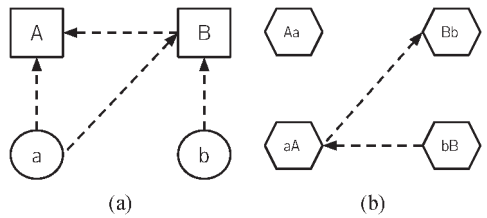

Fig. 3. Examples of *Channel Quality Map* and *Interference Map*. WTPs and $LVAP$s are represented as, respectively, squares and circles. (a) *Channel quality map*. (b) *Interference map*.

or links. An edge $e_{n,m,i} \in E$ with $n, m \in V$ exists if m is within the communication range of n over the *Resource Block* $i \in P$. A weight $q(e_{n,m,i})$ is assigned to each link $e_{n,m,i} \in E : q(e_{n,m,i}) \in \mathbb{N}^+$ represents the channel quality of the link between the two nodes.

A link interference (conflict) graph or *Interference Map* $G^I = (V^I, E^I)$ can be also constructed in such a way that we have a vertex $v^I \in V^I$ for each communication link in E. A directed edge $e^I_{n,m,i} \in E^I$ if the transmitter of the link $n \in V^I$ is within the interference range of the receiver of the link $m \in V^I$ over the *Resource Block* $i \in P$. A weight $q^I(e^I_{n,m,i})$ is assigned to each link $e^I_{n,m,i} \in E^I : q^I(e^I_{n,m,i}) \in \mathbb{N}^+$ represents the potential interference caused by a transmission on link n to link m.

An example of the *Channel Quality and Interference Map* for a WiFi network is sketched in Fig. 3. Here the dashed lines are the weighted directed edges in the *Channel Quality Map* for a given *Resource Block* (weights are not shown to improve readability). A partial *Interference Map* associated with this *Channel Quality Map* is reported in Fig. 3(b). Notice that, in this case, a conflict between two links exists if a transmitter is within the carrier sense range of another transmitter *or* if the transmission from one node interferes (collides) with the reception of a packet at another node. In the conflict graph a directed edge exist between the link $a \rightarrow A$ and the link $B \rightarrow b$ modeling the fact that transmission from node a cause the node B to defer its access to the medium. Similarly another edge exists between the link $b \rightarrow B$ and the link $a \rightarrow A$ because node b can cause interference at A.

The *Channel Quality Map* abstractions can be used to select the *Resource Block*s that can satisfy the requirements of an *LVAP* by intersecting the set of available *Resource Block*s in the network (P_N) with the requested *Resource Block*s (P_L). The set of available *Resource Block*s is obtained as: $P_N = W_1 \cup W_2 \cup \cdots \cup W_N$ The matching *Resource Block*s M are then given by: $M = P_N \cap P_L$ The list of *Resource Block*s M' that satisfy a certain interference level condition, such as the signal to interference plus noise ratio (SINR) between the *LVAP* n and the *WTP* m on the *Resource Block* i being greater than a certain threshold t, is given by: $M' = \{i \in M : SINR(e_{n,m,i}) > t\}$ where $SINR(e_{n,m,i})$ can be estimated via the edge weights in the *Channel Quality and Interference Map*. M' is the empty set if a valid resource allocation is not found. Allocating resources is simply a matter of assigning one or more *Resource Block*s from M' to the *LVAP*, which may result in a handover if the new *Resource Block*(s) are handled by a different *WTP*.

E. Port

Links in a wired network, e.g., a switched Ethernet LAN, are essentially deterministic and the status of a port in a switch is binary, i.e., active or not active. While some Ethernet switches can select the transmission rate (10, 100, 1000 Mb/s), this feature is aimed at reducing power consumption when the traffic load is low and not as a mechanism for coping with fluctuations in the channel quality. In contrast, links in a wireless network are stochastic and, as a result, the physical layer parameters that characterize the radio link between an *LVAP* and a *WTP*, such as transmission power, modulation and coding schemes, and MIMO configuration must be adapted according to the actual channel conditions.

Such level of adaptation requires real-time coordination between *LVAP*s and *WTP*s and can only be implemented near the air interface. The *Port* abstraction allows the *SD-RAN Controller* to reconfigure or replace a certain control policy if its optimal operating conditions are not met.

A port is defined by a 3-tuple $\langle p, m, a \rangle$ where p is the transmission power, m is the set of available Modulation and Coding Schemes (MCS), and a is the MIMO configuration (number of spatial streams). For example, in the case of an 802.11n network, assigning the port configuration: $l \xleftarrow{\langle 30,(0:7),1 \rangle} M'$ to a *LVAP* means that the *WTP* will use a fixed transmission power of 30 dBm, the set of MCS between 0 and 7, and single antenna configuration for its communication with the *LVAP*. The port abstraction allows fast timescale adaptations (MCS adaptation in this example) to be delegated to a local controller located near the *WTP* [19], [20] or to the *WTP* itself. In this work we assume the latter. Finally, since a *Port* specifies the configuration of the link between a *WTP* and a *LVAP*, a *WTP* will have as many *Port* configurations as the number of *LVAP*s it is currently managing.

III. *EmPOWER* Prototype Implementation

To demonstrate the usefulness of the proposed abstractions in real-world settings, we implemented: (i) a SD-RAN controller, (ii) a programmable WiFi data-plane, and (iii) a Python-based SDK. In this section we shall describe in detail the first two components while the features of the SDK are described in the next section. As the implementation of *SD-RAN Controller* in its current form only supports WiFi-based *WTP*s, *Resource Block*s are identified by the 2-tuple $\langle c, b \rangle$ (i.e.: no temporal dimension). A high level view of the *EmPOWER* system architecture is depicted in Fig. 4.

A. SD-RAN Controller

The SD-RAN controller implementation leverages the Tornado Web Server as the web framework [21]. 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. In the following, we elaborate on some of the main features of the *SD-RAN Controller*.

— **Slicing**: The *SD-RAN Controller* can accommodate multiple virtual networks or slices on top of the same physical

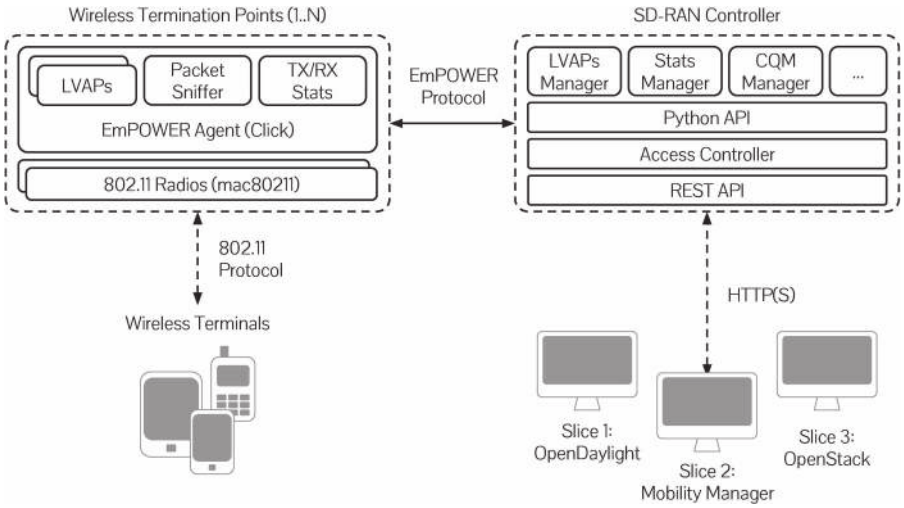

Fig. 4. The EmPOWER system architecture.

infrastructure. A network slice is a virtual network with a specific SSID and its own set of *WTP*s. Clients can *opt-in* a certain slice by associating to its SSID.

— **Soft State**: The only persistent information stored at the controller are the clients' authentication method (currently only ACLs are supported) and the list of currently defined slices. *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-ins* are the modules implementing the *LVAP* control protocol, the Statistics Manager, and the *Channel Quality Map* Manager.

B. Wireless Termination Points

Each *WTP* consists of two components: one OpenvSwitch [22] instance managing the communication over the wired backhaul; and one Click modular router [23] instance implementing the 802.11 data-path. Click is a framework for writing multi-purpose packet processing engines and is being used to implement just the *WTP*s/*LVAP* frame exchange while all the decision logic is implemented at the SD-RAN controller. Communications between Click and the SD-RAN controller take place over a persistent TCP connection.

The *WTP*s in our deployment are a mix of PCEngines ALIX (x86) and Gateworks Cambria (ARM) embedded platforms running the OpenWRT operating system. *WTP*s are equipped with two WiFi interfaces both leveraging a patched *ath9k* driver for their operations. Such patch includes the *LVAP* logic and the per-packet configuration of parameters such as transmission rate and power.

C. Light Virtual Access Point

The Click agent on the *WTP* implements a WiFi split-MAC architecture together with the *SD-RAN Controller*. The state associated with each *LVAP* hosted by the *EmPOWER* Agent is minimal (approximately 54 bytes).

ACK Generation: The agent needs to ensure that an ACK is generated for each unicast data frame that the client sends to its *LVAP*. However, due to their strict timing constraint ACK frame generation is handled in hardware by the WiFi cards. Specifically, Atheros WiFi cards implement this using a BSSID mask register which indicates the common bits of all the BSSIDs for the WiFi card.

Whenever the WiFi card receives a valid 802.11 frame, it matches the destination address against the BSSIDs it is hosting, i.e., check against the bits set in the BSSID mask. If it matches, an ACK frame is generated. However, a practical limitation exists with this approach. Consider the following two BSSIDs *02:00:00:00:00:02* and *02:00:00:00:00:01* which differ only in last two bits. In this case, the generated mask would be *ff:ff:ff:ff:ff:fc*. This leads to the hardware ignoring the last two bits of the destination address of an incoming frame to decide whether to generate an ACK frame. Consequently, a frame destined to *02:00:00:00:00:03* will also cause the hardware to generate an ACK, even though it is not hosting a BSSID with that value: a false positive.

Since *LVAP*s are per-client virtual APs, i.e., client gets one BSSID assigned, this needs to be handled carefully. One way to overcome this issue is to assign BSSIDs to client *LVAP*s such that the mask on the AP where the *LVAP* is being assigned retains as many set bits as possible and remains orthogonal to the masks of neighboring APs. This can be achieved in software by the controller. Limiting the number of *LVAP*s per NIC or spreading them over multiple NICs and APs will also alleviate the problem. Another approach is to suppress spurious ACKs by modifying the check that the hardware performs upon receiving a frame. Today's low-end Broadcom WiFi cards support custom firmware such as OpenFWWF [5] (which is not available for Atheros hardware). However, we conjecture that a program mable content-addressable memory for matching incoming

TABLE II
PYTHON CONSTRUCTS USED BY THE SDK

Operation	*Python Construct*	*Example*
Union ($\cup$)	|	`pool = wtp1.supports | wtp2.supports`
Intersection ($\cap$)	&	`matches = lvap.supports & pool`
Filtering ($\forall$)	List Comprehensions	`valid = [block for block in matches if block.rssi[lvap]['ewma_rssi'] >= -65]`
Assignment ($\leftarrow$)	=	`lvap.assigned_to = valid[0]`
Reconfiguration ($\xleftarrow{\langle p,m,a \rangle}$)	[]	`lvap.assigned_to[valid[0]].tx_power = 27`

frames in hardware enables possibilities beyond just selective ACK generation, with little increase in cost and performance impact. This is particularly important as 802.11ac adoption is increasing, which supports throughput on the order of 6.77 Gb/s. Recent work on software radios such as OpenRadio [24] will also aid in this direction.

AP Discovery: As per IEEE 802.11, clients can perform active scans by broadcasting probe request messages on all possible channels at the same time. An agent receiving such a probe request frame forwards it to the *SD-RAN Controller*. The latter then generates an *LVAP* with a BSSID unique to the client if not already done, and retrieves the list of SSIDs to announce (the union of SSIDs across all slices that the *WTP* belongs to). It then instructs the agent to generate a probe response for each of these SSIDs, through the client-specific BSSID. At this point the client can authenticate and associate to its *LVAP* as per IEEE 802.11 procedures. If a client does not associate to its *LVAP* within a configurable amount of time, the *LVAP* is removed from the agent. The agent process maintains a lookup table with the mappings of the client's MAC address to the *LVAP*s state. It then makes use of this per client state to prepare the right 802.11 frames.

IV. SOFTWARE DEVELOPMENT KIT

A Python-based SDK mapping the abstractions introduced in Section II to Python constructs is made available to developers (see Table II). In this section we shall briefly summarize some of the SDK's most interesting features. The comprehensive list of Python primitives can be found in Table III. Primitives can operate in *polling* or *trigger* mode. In the former mode (*polling*) the *SD-RAN Controller* periodically polls one or more *WTP*s for a specific information, e.g., the number of packets received by and *LVAP*. In the latter mode (*trigger*) a thread is created at one or more *WTP*s. Such thread is identified by a firing condition, e.g., the RSSI of one *LVAP* going below a certain threshold. When such condition is verified a message is generated by the *WTP*. A termination condition can also be specified. All primitives are non-blocking and, as such, they immediately return control to the calling *Network App*. An optional *callback* method can be provided specifying a method to be executed when the primitive returns a result. A Python dictionary, whose structure depends on the actual primitive, is passed as parameter to the callback. Moreover, all primitives require at least one parameter named *ssid* specifying the name of the slice on which they shall operate.

A. Light Virtual Access Point

The *LVAP* is exposed to the programmer through a Python object mapping properties to operations. 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 `assigned_to` property field of an *LVAP* object. Similarly, performing a handover is as simple as assigning a new list of *Resource Block*s to the same field. It is worth stressing that, each *Resource Block* contains implicitly the information of the *WTP* at which it is available.

The following Python routine schedules an *LVAP* on a random *Resource Block* whose RSSI to the *LVAP* is greater than or equal to -65 dBm. The routine then configures the *WTP* transmission power toward the *LVAP* (in dBm):

```
def handover(lvap, wtps):
    # Initialize the Resource Pool
    pool = ResourcePool()

    # Update the Resource Pool
    for wtp in wtps:
        pool = pool | wtp.supports

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

    # Filter Resource Blocks by RSSI
    valid = [blk for blk in matches
             if blk.ucqm[lvap]['ewma_rssi'] >= -65]

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

    # Enable RTS/CTS
    port = lvap.assigned_to[new_block]
    port.tx_power = 30
```

Listing 1. Handoff a LVAP to a WTP to Which RSSI is at Least -65 dB.

The method accepts two input parameters, a *LVAP* (`lvap`) and a list of *WTP*s (`wtps`). The method initializes the network *Resource Pool* with the *Resource Block*s available at every *WTP*. An intersection of the network's *Resource Pool* with the resource blocks supported by *LVAP* is then computed. The resulting set is then traversed filtering-out the *Resource Block*s whose RSSI to the *LVAP* is above -65 dBm. Finally, one random *Resource Block* matching such condition is assigned to the *LVAP* and the *Port* configuration for that *Resource Block* is updated. Notice that, for the sake of simplicity, error handling

TABLE III
PROGRAMMING PRIMITIVES IN THE PYTHON-BASED SDK

Primitive	*Parameters*	*Mode*	*Description*	*Section*
`packets/bytes_count`	lvap, bins, every	Polling	Aggregate TX/RX packets/bytes in the specified bins	IV-A
`link_stats`	lvap, every	Polling	Fetch per–link packet transmission statistics	IV-A
`ucqm/ncqm`	addrs, block, every	Polling	Fetch RSSI statistics for neighboring Stations/APs	IV-B
`summary`	lvaps, keep, limit, rates, every	Trigger	Get fine grained link–layer events	IV-B
`rssi`	lvaps, relation, value	Trigger	Callback when a condition on the RSSI is verified	IV-B

has been omitted. 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.

Packet counters allow programmers to track the traffic exchanged by a certain *LVAP* and to use binning to aggregate such information by frame length (useful in wireless networks due to the fact that short packets incur higher transmission overheads). For example:

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

Listing 2. Accessing packet counters for a LVAP.

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 5000 ms. It is also possible to issue a single query by specifying -1 as polling period. Notice that the query is executed only if the *LVAP* is associated to the specified SSID (*Guests* in this example) otherwise the packet counters are not updated.

Access to the downlink transmission statistics is possible using the *link_stats* primitive. For each supported MCS the average delivery probability and the expected throughput in the last observation window are reported. Moreover, also the total number of successful and failed transmissions since the *LVAP* was moved to the *WTP* are reported.

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

Listing 3. Accessing link transmission statistics to a LVAP.

B. Channel Quality and Interference Maps

The *Channel Quality Map* is exposed to the network programmer by means of two data structures: the *User Channel Quality Map* (*UCQM*) and the *Network Channel Quality Map* (*NCQM*). Both are 3-dimensional matrices where each entry is the channel quality over one *Resource Block* between: an *LVAP* and a *WTP* in the case of the *UCQM*; and between two *WTP*s in the case of the *NCQM*.

In our implementation of the *Channel Quality Map* we use the RSSI measured at each *WTP* as an approximation of the channel quality. A monitor interface created on top of each physical radio available at a *WTP* is used to extract the signal strength field present in the radiotap header of every decoded WiFi frame. To separately build the *UCQM* and the *NCQM*, the *To-DS*, *From-DS*, frame type, and frame sub-type fields present in the 802.11 header are used. For each neighbor within the decoding range, the *WTP*s compute the average of the RSSI over windows of 500ms, an exponential weighted moving average and a N-points smoothed moving average are also maintained. The latter two filters have been selected because they can reduce the noise while being responsive to RSSI changes. Such property is useful when dealing with fast-fading affected RSSI signals. At the same time, fast response allows to promptly react to changes in the channel conditions.

The code below periodically queries the specified *WTP* for its neighboring stations (the RSSI from neighboring APs can be tracked using the *ncqm* primitive):

```
ucqm(addrs='ff:ff:ff:ff:ff:ff',
     block=('04:f0:21:09:f9:96', 36, L20)
     every=5000,
     ssid='Guests',
     callback=ucqm_callback)
```

Listing 4. UCQM query creation.

The query is executed periodically with the period set by the `every` parameter (in ms). Specifying $every = -1$ will result in a single query being issued. In the above example specifying `ff:ff:ff:ff:ff:ff` will return the RSSI of any station within the decoding range of *WTP 04:F0:21:09:F9:96* on channel 36.

It is worth noticing that, we are using the general term *stations* and *access points* instead of, respectively, *LVAP*s and *WTP*s, since the *Channel Quality Map* tracks the RSSI level of any active WiFi device including the ones belonging to networks that are not under the administrative domain of the *SD-RAN Controller*. This includes also wireless clients that are not associated to *any* network but have their wireless interface active (WiFi clients periodically broadcast *Probe Request* messages to discover available networks).

A sample output of the `ucqm` primitive is reported below. In this case the station *a0:d3:c1:a8:e4:c3* is a neighbor of the *WTP 04:f0:21:09:f9:96* on the 802.11a channel 36. The report includes, besides the previously described averages, also the total number of frames received since the query was created (`hist_packets`) together with

the average (`last_rssi_avg`), the standard deviation (`last_rssi_std`), and the number (`last_packets`) of RSSI measurements taken during the last observation window.

```
{
    "a0:d3:c1:a8:e4:c3": {
        "ewma_rssi": -82,
        "hist_packets": 15810,
        "last_packets": 10,
        "last_rssi_avg": -79,
        "last_rssi_std": 7
        "sma_rssi": -82,
    }
}
```

Listing 5. UCQM query output.

The *Channel Quality Map* can be accessed at the frame-level granularity providing the network programmer with a real-time picture of all link-layer events. Each *WTP* tracks the following meta-data associated to link-layer events:

- *Transmitter Address*. The MAC address of the transmitter.
- *TSFT*. The 802.11 MAC's 64-bit Time Synchronization Function Timer. Each frame received by the radio interface is timestamped with a 1 μsec resolution clock by the 802.11 driver.
- *Sequence*, The 802.11 MAC's 16-bit sequence number. This counter is incremented by the transmitter after a successful transmission.
- The frame *RSSI* (in dB), *Rate* (in Mb/s), *Length* (in bytes), and *Duration* (in μsec).

The collected traces are then periodically delivered to the *SD-RAN Controller* where they are synchronized to a common time reference. Notice that, only successful unique frames transmissions are recorded, i.e. frames with incorrect checksum and/or PLCP header as well as retransmitted frames are ignored. The collected meta-data can be exploited for several purposes. In Section VI we report on a simple implementation of a trace merging system. The link-layer events and the associated meta-data can be accessed using the *summary* primitive:

```
summary(lvaps='11:22:33:44:55:66',
        keep=None
        limit=-1
        rates=[6, 54]
        every=5000,
        ssid='Guests',
        callback=summary_callback)
```

Listing 6. Collect link-layer meta-data.

The statement above generates a periodic callback when a traffic trace has been received from a *WTP*. The traffic traces includes all the link-layer events within the decoding range of the *WTP*. The *keep* parameter allows the network programmer to specify how many events must be stored by the controller. Specifying -1 results in the controller storing all events. Specifying a positive integer will result in the controller *flushing* the stored events when their number exceed the specified value. The *limit* parameters instructs the *WTP* to send only a specified number of traces after which the operation is stopped and all the allocated data structures on the *WTP* are freed. Specifying -1 results in the *WTP* sending traffic traces forever. It is also possible to filter link-layer by transmission rates. In the example above only transmission happening at 6 or at 54 Mb/s are recorded. Specifying `ff:ff:ff:ff:ff:ff` as `lvaps` will trigger the callback for any station in the network.

Notice that the entire data structure containing a frame meta-information is 18-bytes long. A saturated 54-Mb/s channel can deliver up to 2336 frames/s.[3] In such scenario the system would generate 42048 bytes of meta-data per second per radio interface which correspond to a signaling bandwidth between the *WTP* and *SD-RAN Controller* of about 336 kb/s which is negligible considering the widespread adoption of Gigabit Ethernet in enterprise networks. Due to channel contention, collisions, and retransmissions the actual number of unique frames that can be practically received by a *WTP* is typically smaller.

Finally, the `rssi` primitive allows the programmer to trigger a callback the first time the RSSI of a *LVAP* verifies a certain condition at any *WTP* in network. For example:

```
summary(lvaps='11:22:33:44:55:66',
        keep=None
        limit=-1
        rates=[6, 54]
        every=5000,
        ssid='Guests',
        callback=summary_callback)
```

Listing 7. Create an RSSI trigger.

After the trigger has fired the first time and as long as the RSSI remains below -70 dBm, the callback method is not called again by the same *WTP*, however the same callback may be triggered by other *WTP*s. Specifying `ff:ff:ff:ff:ff:ff` as `lvaps` will trigger the callback when the RSSI of any *LVAP* at any *WTP* is below -70 dBm.

C. Port

Programmers can fetch the *Resource Block*(s) on which an *LVAP* is currently scheduled together with its *Port* configuration by accessing the *assigned_to* property of an *LVAP* object:

```
>>>lvap.assigned_to
{(04:F0:21:09:F9:96, 36, L20):
  (0C:3E:9F:57:1A:B6, ⟨6,12,24,36,48,54⟩, 27 dBm, 1)}
```

As it can be seen, the dictionary above contains a single entry mapping a *Resource Block* with a *Port* configuration. In this example, the *LVAP 0C:3E:9F:57:1A:B6* has been assigned to the *Resource Block* $\langle 36, L20 \rangle$ scheduled at the *WTP 04:F0:21:09:F9:96*. The *Port* configuration specifies which

[3] At 54 Mb/s an 802.11a radio encodes 216 bits/symbol. The OFDM encoding then add 6 more bits at the end of the frame, so a maximum length frame of 1536 bytes becomes a string of 12288 bits plus 6 trailing bits. The total 12294 bits can be encoded with 57 symbols each requiring 6 μsec for transmission. As a result, ignoring backoff, the minimum time required for transmitting this frame is: DIFS (34 μsec) + DATA(248 μsec) + ACK(24 μsec) = 322 μsec, which corresponds to 2336 frames/second.

range of parameters the *WTP* can use for its communication with the *LVAP*, in this case: single spatial stream, fixed transmission power, and adaptive transmission rates selection (with a constraint on the possible MCS). The current implementation of the *Port* abstraction supports the following parameters:

- *TX Power*. Fixed transmission power (in dB).
- *Modulation and Coding Scheme (MCS)*. List of MCS values that can be used by the rate selection algorithm.
- *MIMO Configuration*. Number of spatial streams.
- *RTS/CTS Threshold*. Frame length value above which the RTS/CTS handshake must be used.
- *No ACK*. The *WTP* will not wait for ACKs.

D. Network Apps

In our SDK, *Network App*s are Python modules loaded either when the *SD-RAN Controller* is started or using the controller REST interface. Every module is required to implement a `launch` method called when the *Network App* is loaded to perform initialization tasks. The launch method must return an instance of the base class *EmpowerApp*. The parameters accepted by the `launch` method are defined by the *Network App*. A mandatory *ssid* parameter must always be provided to specify on which slice the module shall operate. An optional *period* parameter can be used to specify the period of the control loop. The complete code of a simple mobility management *Network App* is reported below.

```
from empower.core.app import EmpowerApp

class MobilityManager(EmpowerApp):

  def __init__(self, ssid, period):

    EmpowerApp.__init__(self, ssid, period)

    self.rssi(lvaps='ff:ff:ff:ff:ff:ff',
              relation='LT',
              value=-70,
              callback=self.low_rssi)

  def low_rssi(self, lvap, wtp, trigger):
    handover(lvap, self.wtps())

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

def launch(ssid, period=None):
  return MobilityManager(ssid, period)
```

Listing 8. Sample mobility management *Network App.*

As it can be seen, this example *Network App* 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 a *LVAP* should be handed-over to another *WTP*; and (ii) the *callback* method which is invoked when the RSSI of a *LVAP* at its hosting *WTP* is going below a certain threshold (-70 dB in this example). Notice how a RSSI trigger matching all *LVAP*s in the network is created in the *class constructor*. Notice also that, as it can be seen for the *rssi* call, all the primitives described in the previous sections can be invoked as class methods without specifying the `ssid` parameter. Finally, to reduce the verbosity, error handling and logging code have been omitted. Moreover, this *Network App* does not take into account the actual traffic generated by the clients and as result could overload some *WTP*s. A more refined implementation would also implement load balancing functionality spreading the traffic across the network possibly using as input the *LVAP*s' packet counters.

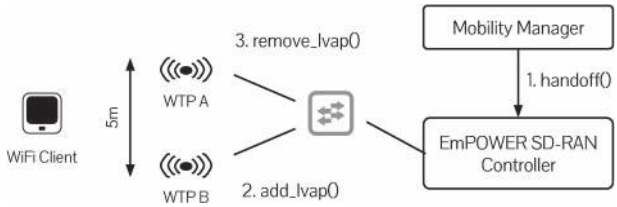

Fig. 5. Network setup for the $LVAP$ evaluation experiment.

V. Evaluation

This section is divided into two parts. In the first part our aim is to demonstrate the usefulness of the *LVAP* concept. In the second we quantify the signaling overhead associated with the statistics subsystem. Notice that, although overhead characterization results are reported only for the packet counters, the general trends apply also to the other subsystem, namely interference maps and triggers. The system has been evaluated over a simple testbed composed of three *WTP*s. Each *WTP* is equipped with two wireless NICs each tuned to a different channel, specifically 6 (2.4 GHz band), and 36 (5 GHz band). In our testbed (a typical office environment) channel 36 is not shared with any other network, while channel 6 is used by several other networks. The network controller is a Dell Laptop equipped with a quad core Intel i7 CPU and 8 GB of RAM.[4] Iperf [25] is used for synthetic traffic generation.

A. Seamless Handovers via LVAP Abstraction

To assess the effectiveness of the *LVAP* abstraction we implemented a simple *Network App* which periodically hands over a wireless client between two *WTP*s. The *WTP*s are 5 m apart and share the same backhaul (an Ethernet LAN), the wireless client is equidistant from the two *WTP*s. The network setup is sketched in Fig. 5. Note that, in addition to the *SD-RAN Controller* presented in this work, also an OpenFlow controller, namely POX [26], has been used to pre-configure the Ethernet switch before each handover.

Client mobility is emulated by progressively reducing the transmission power of the serving AP. We consider a standard WiFi network as the baseline scenario. Recall that, in a standard WiFi network, handovers are triggered by the wireless clients and that 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 is not deterministic and can take a variable amount of time.

[4]Note however that the Python interpreter used to run the controller can leverage only one CPU core.

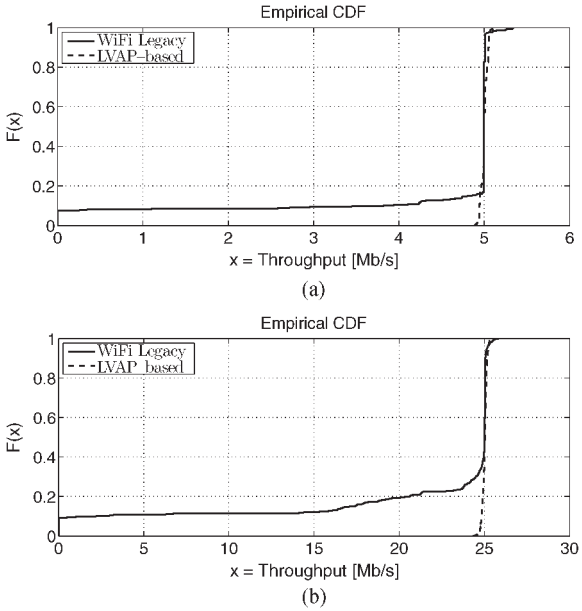

Fig. 6. Distribution of the goodput at the receiver's side when the client is performing a handover every 10 s. (a) 5 Mb/s flow. (b) 25 Mb/s flow.

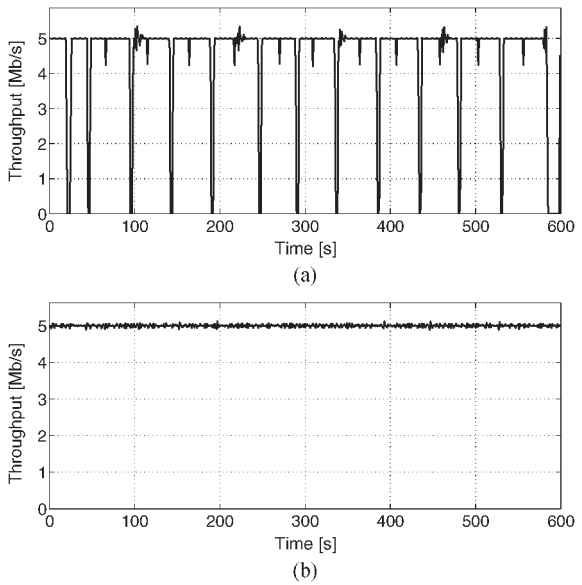

Fig. 7. Instantaneous goodput at the receiver's side when the client is performing a handover every 10 s. (a) WiFi Legacy. (b)$LVAP$-based handover.

The traffic is generated at the wireless client towards a fixed node which shares the backhaul with the two *WTP*s and is made up of a single UDP flow with constant payload size (1472 bytes) and constant bitrate (5, or 25 Mb/s). Fig. 6 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.

The instantaneous bitrate in both the baseline and the *LVAP* (SDN) scenario for the 5 Mb/s flow is shown in Fig. 7. As it

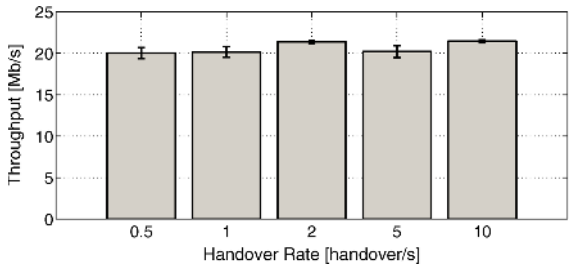

Fig. 8. Wireless client throughput with an increasing handover rate using the $LVAP$-based handover. The client is generating a saturated TCP stream toward a remote host. The client throughput is not affected by the handover rate.

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 a handover a full network scan followed by a DHCP exchange is triggered. This procedure can take up to 1–2 seconds to complete. As this procedure is triggered by the client, it can effectively lead to scenarios where the client 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. Fig. 8 shows the throughput at the receiver's side for an increasing handover rate from 1 handover every two seconds up to 10 handovers per second. 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.

B. Signaling Overhead Characterization

To demonstrate the minimal overhead nature of the querying subsystem at both the *WTP*s and the controller we implemented a simple application which periodically requests uplink and downlink statistics for a LVAP. The polling period is increased from 1 request every two seconds to 10 requests per second. Fig. 9(a) shows the throughput attained by the wireless client under an increasing polling rate. It can be seen that 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.

Fig. 9(b) shows the average bandwidth used for the signaling related traffic between WTPs and the controller under an increasing polling rate. Although the bandwidth consumption increases as 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. This amount of signaling overhead can be easily accommodated by modern Gigabit Ethernet deployments found in enterprise networks.

We also evaluated the network statistics subsystem in terms of controller CPU utilization and signaling overhead with a large number of wireless clients. To simulate a deployment with tens of clients we artificially created an increasing number of *LVAP*s at the 3 *WTP*s in our testbed. Although not associated to an actual client, such *LVAP*s are, from the perspective of the controller, equivalent to a scenario where an equivalent

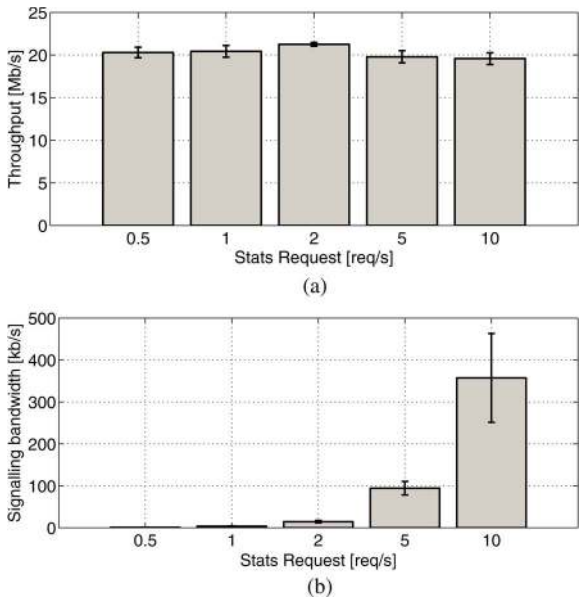

Fig. 9. Signaling overhead and its impact on client perceived performance (throughput). The client is generating a saturated TCP stream toward a remote host. The client throughput is unaffected by the polling frequency (a) even while the required signaling bandwidth rises sharply with the polling rate (b). (a) Throughput. (b) Signaling related bandwidth consumption.

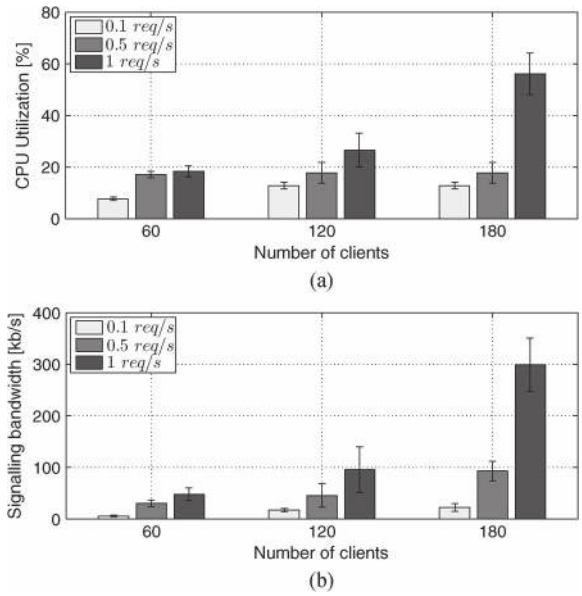

Fig. 10. CPU Utilization (a) and signaling overhead (b) for an increasing number of clients. (a) CPU utilization. (b) Signaling related bandwidth consumption.

number of clients are active and continuously communicating. We performed this experiment using an increasing number of virtual clients as well as an increasing network statistics polling rate. Results are shown in Fig. 10. It can be seen that the CPU utilization increases with the polling rate *and* with the number of clients. However even with 180 active clients and with a polling rate of 1 req/s the CPU utilization remains between 50% and 60%. Similarly the aggregate signaling related bandwidth consumption even in an extreme case is only around 300 kb/s.

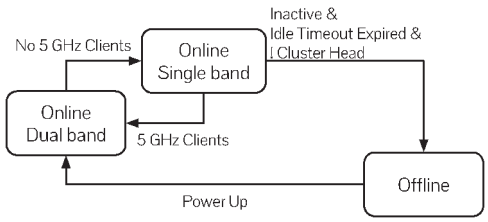

Fig. 11. FSM for the energy management *Network App*.

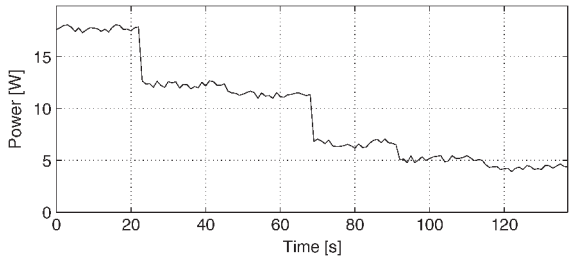

Fig. 12. Network-wide power consumption with varying number of active clients. The reduction in power consumption is due to the joint energy and mobility management *Network App*.

VI. Applications

In this section we describe three *Network App*s implemented using our SDK and tested over a 20-nodes indoor testbed deployed at CREATE-NET premises. The code of these *Network App*s is omitted for brevity.

A. Energy-Aware Mobility Management

A simple energy management *Network App* targeting dual bands (2.4/5 GHz) enterprise WLANs has been implemented. In this scenario APs are partitioned into clusters each with a single *Master* AP and multiple *Slaves* APs. *Masters* are chosen in such a way to provide full coverage and must remain always active while *Slaves* can be selectively turned on/off. Fig. 11 depicts the Finite State Machine (FSM) implemented by this *Network App*. Here in the *Online (Dual band)* mode, an AP and all its wireless interfaces are on. The *Online (Single band)* mode represents the case when the 5 GHz interface is turned off. A *Slave* AP belonging to a cluster with less than N clients and that has been inactive for at least T_{idle} seconds is transitioned to the *Offline* state. Here, inactive means that no *LVAP* is hosted by the *WTP*. The transition from the *Online (Dual band)* to *Online (Single band)* modes and vice-versa is triggered according to the presence of clients supporting the 5 GHz band in the cluster. Finally, if there are more than N clients in the cluster then the *WTP* is brought back to the *Online* mode. As this *Network App* is a simple proof-of-concept aimed at demonstrating the capabilities of our wireless SDN framework, it does not take into account other relevant factors like as traffic patterns and/or channel quality.

Fig. 12 plots the real time energy consumption of a simple network composed of 3 WiFi APs and 5 clients. We assume that all APs are in the same cluster and that the *Master* has been selected manually to provide sufficient coverage. Clients joining the network are handed over to the AP that provides the best performance in terms of SNR. A constraint on the maximum

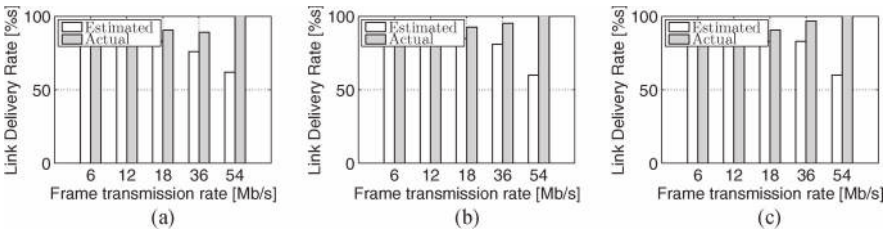

Fig. 13. Actual and estimated uplink frame loss at the three WTPs for different transmission rates when the client has a good link to all the WTPs (≈ 0.95 frame delivery ratio). (a) *WTP* 1. (b) *WTP* 2. (c) *WTP* 3.

number of clients that can be allocated to the same AP is also introduced to avoid overloading APs. During this measurement campaign the value for that parameter has been set to 2. Moreover only 2 out of 5 clients support the 5 GHz band.

Initially all 5 wireless clients are active streaming a low bitrate video from YouTube. After 20 seconds one client scheduled on the 5 GHz band leaves the network. As a result the remaining 4 clients are consolidated on 2 APs and the third one is turned off resulting in a significant power consumption reduction. After 20 seconds one of the clients scheduled on the 2.4 GHz band leaves the network resulting in a sensible power consumption reduction. Clients continue to leave the network thereafter (one client every 20 seconds). Eventually, no client is active in the network and only the master AP is kept powered to preserve network coverage.

B. Uplink Monitoring

Performing handover decisions using as an input only the RSSI strength between clients and APs may lead to suboptimal performance as RSSI may not always be a good indicator of link layer performance. Uplink and downlink frame loss rates together can be better alternative metrics. While the downlink frame loss can be easily tracked by the AP to which a client is currently associated, tracking the uplink frame loss rate is much harder if client modifications are not allowed. This is essentially due to the fact the amount of frames actually sent by a client is an unknown variable at the network side.

This *Network App* exploits the frame tracking capabilities supported by our framework to build a globally synchronized view of all link-layer events in the network. The *Network App* implements a passive synchronization algorithm that does not require the *WTP*s to share a common clock (which would be impractical for μsec-level precision). The *Network App* exploits the broadcast nature of a wireless link and in particular the fact that the same transmission can be recorded by multiple in-range *WTP*s. Moreover, at least for indoor deployments, we can assume simultaneous receptions at the different *WTP*s since the propagation delay even for a 500 m link is less than 1 μsec.

This *Network App* implements a synchronization mechanism similar to the one used by Jigsaw [27]. The algorithm looks for simultaneous receptions of the same frame (identified using the 802.11 sequence number) at different *WTP*s to synchronize the clock of the other *WTP*s to a common reference point. Clock adjustment is performed at the *SD-RAN Controller* where frame summaries are merged into a single globally synchronized traffic trace. Such operations are performed in real-time with low overhead and without requiring network downtime. Once the global clock synchronization has been achieved, the *Network App* can track in real-time the frame delivery rates between one or more *LVAP*s and all the *WTP*s in the network. Frame delivery statistics are computed every 500 ms and are aggregated by transmission rate. An exponential moving average is also maintained.

To evaluate the accuracy of the *Network App* we exploited a network setup composed of a single client and three *WTP*s. Traffic is injected from the wireless client as a single UDP stream. Packet transmission rate and payload are kept fixed at, respectively, 100 packets/s and 1472 bytes. Measurements are taken for different frame transmission rates. Finally, impairments on the link between client and *WTP*s are simulated by randomly dropping received frames with probability p. For all measurements a total of 6000 frames were generated. Confidence intervals were very small for all the data points and have therefore be omitted to improve readability.

Fig. 13 reports the actual and estimated frame delivery rates at the three *WTP*s in the network for different transmission rates. During this campaign the dropping probability has been set for all links to $p = 0.05$ simulating good channel conditions. As it can be seen the estimated frame delivery rates are very close to the actual values.

Fig. 14 reports the outcomes of the second measurement campaign. In this case the dropping probability of two links has been set to $p = 0.8$ simulating poor channel condition while the dropping probability of the third link has been set to 0.05 simulating good channel condition. This scenario is representative of the case where at least one of the in-range receivers of a given wireless client is experiencing a good channel condition. This situation is at least needed to allow the *Network App* to have a good estimate of the number of frames transmitted by the client. As it can be seen also in this case the *Network App* can closely track the actual link delivery rates for all the transmission rates. Note that, measurements have been taken separately for each supported rate and that the links experiencing poor channel conditions are randomly selected at the beginning of the measurements. As a result, for each supported transmission rate the *WTP*s which are experiencing poor channel conditions are different.

Fig. 15 and Fig. 16 reports the actual and estimated frame delivery rates for different payload lengths and for different loads. In both cases the transmission rate was kept fixed at 36 Mb/s. The payload size for Fig. 16 was also kept fixed at 1472 bytes. It can be seen that the estimated frame delivery rates are very close to the actual values.

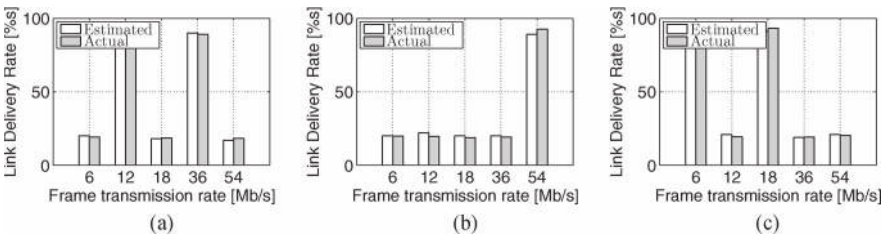

Fig. 14. Actual and estimated uplink frame loss at the three WTPs for different transmission rates when the client has a good link to at least one of the WTPs (≈ 0.95 frame delivery ratio). (a) *WTP* 1. (b) *WTP* 2. (c) *WTP* 3.

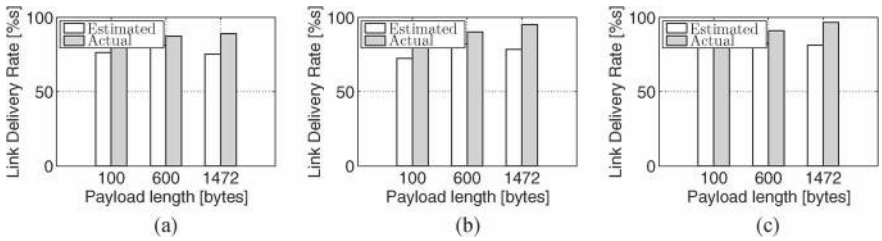

Fig. 15. Actual and estimated uplink frame loss at the three WTPs for an increasing payload length. (a) *WTP* 1. (b) *WTP* 2. (c) *WTP* 3.

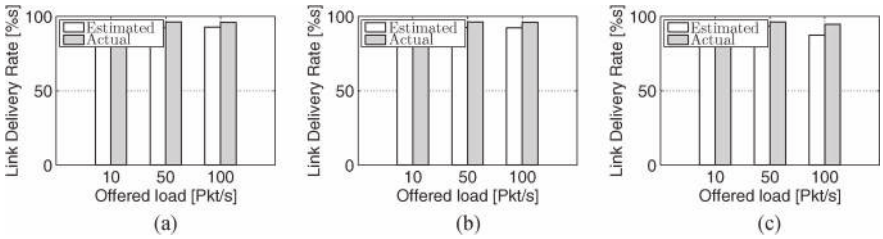

Fig. 16. Actual and estimated uplink frame loss at the three WTPs for an increasing offered load. Transmission rate and payload length were kept fixed at 36 Mb/s and 1472 bytes, respectively. (a) *WTP* 1. (b) *WTP* 2. (c) *WTP* 3.

C. Interference-Aware Channel Assignment

An efficient channel assignment can dramatically improve network performance in dense WLANs. In this section we demonstrate how the *Channel Quality and Interference Map* abstractions can be leveraged for this purpose. Note that the definition of *Interference Map* from Section II-D can be extended to result in a more realistic interference map including *external* but nearby WiFi APs and stations. A simple *Network App* leveraging the SDK's RSSI tracking primitives has been implemented to achieve such a goal.

The *Network App* periodically traverses all the possible *LVAP* pairs (an *LVAP* identifies the bi-directional link between a client and its *WTP*). For each pair the *Network App* checks if a conflict exists between the uplink/downlink of the first *LVAP* and the second *LVAP*. In such a case the pair is added to the *Interference Map*. Fig. 17(a) sketches the *Channel Quality Map* for the setup used in this experiment whereas a graphical representation of the *Interference Map* is shown in Fig. 17(b).

The dashed lines in Fig. 17(a) represent the weighted directed edges in the *UCQM* and the *NCQM* for a given *Resource Block*. As it can be seen given the fact that all the nodes are within decoding range the resulting conflict graph depicted in Fig. 17(b) is almost fully connected. No edge exists between the link $A \rightarrow a$ and the link $B \rightarrow b$ in Fig. 17(b) because due to limitations in the current implementation it is not possible to gather the list of interfering devices *at* the client.

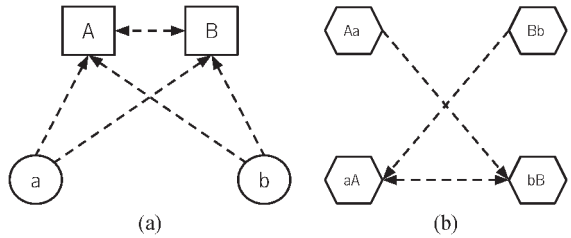

Fig. 17. Interference-aware channel assignment. WTPs are represented as squares while $LVAP$s are represented as circles. (a) *Channel quality map*. (b) *Interference map*.

The knowledge contained in the conflict graph can be effectively leveraged to improve network performance by implementing suitable load-balancing and channel assignment algorithms. A simple implementation of the DSATUR [28] algorithm has been implemented as proof-of-concept. The system performance has been tested before and after the channel assignment. Traffic consists of two saturated TCP connections generated at the two clients toward a node which shares the backhaul with the two *WTP*s. Fig. 18 reports the instantaneous aggregated client throughput before $(w/o\ CF)$ and

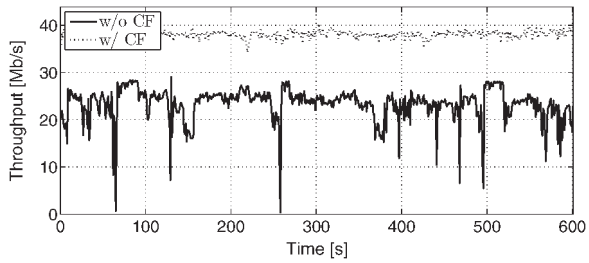

Fig. 18. Instantaneous aggregated client throughput before $(w/o\ CF)$ and after (w/CF) channel assignment. Performance improvement is result of channel assignment.

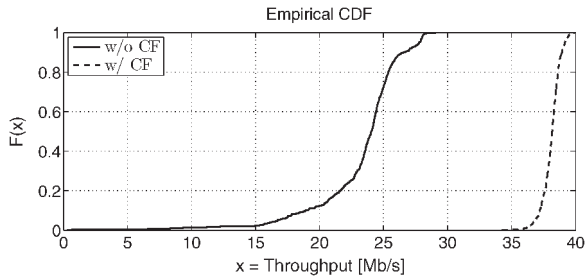

Fig. 19. Distribution of the throughput before $(w/o\ CF)$ and after (w/CF) channel assignment. Performance improvement is result of channel assignment.

after (w/CF) channel assignment. As expected given the very simple topology, when channel assignment is performed the aggregated throughout improves significantly and it is also stable. The latter aspect can be better seen in Fig. 19 where the empirical CDF of the throughput samples is plotted.

VII. Related Work

In this section we review and discuss the most relevant works on: (i) wireless SDNs, (ii) high-level programming primitives for SDNs, and (iii) interference modeling.

A. *Wireless and Mobile* SDNs

In [29], Murty et al. present a software architecture addressing the problem of extensibility in wireless LANs. The authors define a set of APIs allowing clients and APs to be managed by a centralized controller. Odin [4] is a 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.

In [30], [31] the authors argue that SDN can simplify the design and management of mobile networks, while enabling new services. This work is extended in [2], where SoftCell, an architecture supporting fine-grained policies for the mobile core network, is presented. Cloud-RAN (C-RAN) [3] 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 [32]. Similarly, in [33] an OpenFlow-based control plane for LTE/EPC is presented. SoftRAN [1] proposes a fundamental redesign of the cellular RAN by introducing a big base station abstraction mapping the resources of all base stations in a certain area.

Although the above works demonstrate continuing interest in addressing the complexity of future mobile networks using SDN principles, none of them put the focus on providing programmers with a high-level interface to control the next-generation RANs. Some of these works tackles the challenges from the architectural perspective [3], [30]–[33]. Some attempts at providing a more high-level view of the network can be found in [2], [4] but these works either address the challenges in the mobile core (EPC) or focus only on wireless clients.

Closest by goals and principles to our work is SoftRAN [1] and CoAP [34]. In the former work the authors introduce the big base station abstraction to address the challenges raised by the densification of the cellular RAN while in the latter work the authors tackle the challenge of orchestrating the operation of a multitude of residential wireless gateways. Both works however do not elaborate on the primitives to be exposed to the programmers, the focus of our work.

B. *Languages and Abstractions for* SDNs

Several recent works have put their focus on applying high-level programming primitives and techniques to SDN [9]–[15], [35]. Their target however is to enable programmability in *wired* networks typically relying on OpenFlow as data-plane control API. In contrast our work focuses on modeling the most critical aspects of a WiFi-based RAN (gathering network state, interference-aware resource allocation and network reconfiguration and adaptation) and exposing them to the network programmers through a set of technology agnostic programming primitives.

Finally, the argument for handling fast timescale events as close as possible to the place where they are originated is made in [19], [20]. However, both works do not address the way global channel quality information shall be exposed to the network programmer (as we do via the *Channel Quality Map* abstraction) nor they propose a viable network reconfiguration model (the *Port* abstraction).

C. *Interference Modeling and Management*

Given the vast amount of literature on this topic our discussion here focuses on empirical approaches with a real-world evaluation. For a broad survey on interference modeling techniques in 802.11 networks we refer the reader to [36]. Passive and/or active measurements are leveraged by several authors to derive either the conflict graph or the interference graph of a wireless network. In [37], [38] active measurements are exploited to study how mutual link interference affects packet delivery ratio and throughput. In [39] micro-probing (i.e., active measurements lasting few milliseconds) is used to detect conflicts between links. Passive interference graph construction techniques are presented in [40], [41]. WIT [42] and Jigsaw [27] are two other examples of passive interference monitoring techniques aimed at modeling cross-link interference. A framework capable of performing root cause analysis in WiFi networks is presented in [43].

Conflict graphs are leveraged in [44] to manage the effect of interference when multiple transmitters employ variable channel widths. An architecture using micro-probing to jointly address channel assignment and transmission power control is presented in [45]. Centralized scheduling is exploited in [46] to mitigate hidden and exposed terminals issues in WiFi-based networks. Interference modeling plays a key role in the client scheduling problem.

VIII. Conclusion

This paper aims at lowering the barrier for developing and deploying the next generation application and services for wireless and mobile networks. Towards this end, we propose a set of programming primitives that empower the developers with expressive tools to control the network while hiding away the implementation details of the underlying technology. The proposed primitives have been exposed to the network programmer through a Python-based SDK and have been used to implement a number of applications ranging from energy-consumption-aware load balancing to channel assignment. The declarative nature of the primitives allows for a powerful yet concise programming language.

The proposed abstractions address four wireless networks control aspects, namely: wireless clients state management, resource allocation, network monitoring, and network reconfiguration. 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 wireless networks. This essentially translates to separating policies (e.g., the set of transmission rates for use in link adaptation), from mechanisms, i.e., the way knobs need to be turned 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.

Our current work aims at enhancing the runtime system to improve system performances, extending the language to support QoS-related primitives, and investigating how to properly address more advanced applications and services such as content caching and and edge computing. Moreover, the entire stack including the datapath implementation, the reference *SD-RAN Controller*, and the Python-based SDK have been released under a permissive license making it available to the broad research community. Finally, we also plan to extend and validate the proposed programming abstractions in a mobile network scenario by adding support for LTE and LTE-Advanced technology.

References

[1] A. Gudipati, D. Perry, L. E. Li, and S. Katti, "SoftRAN: Software defined radio access network," in *Proc. ACM HotSDN*, 2013, pp. 25–30.

[2] X. Jin, L. Li, L. Vanbever, and J. Rexford, "SoftCell: Scalable and flexible cellular core network architecture," in *Proc. ACM CoNEXT*, 2013, pp. 163–174.

[3] M. HadZialic, B. Dosenovic, M. Dzaferagic, and J. Musovic, "Cloud-RAN: Innovative radio access network architecture," in *Proc. IEEE ELMAR*, 2013, pp. 115–120.

[4] L. Suresh, J. Schulz-Zander, R. Merz, A. Feldmann, and T. Vazao, "Towards programmable enterprise WLANs with Odin," in *Proc. ACM HotSDN*, 2012, pp. 115–120.

[5] G. Bianchi *et al.*, "MAClets: Active MAC protocols over hard-coded devices," in *Proc. ACM CoNEXT*, 2012, pp. 229–240.

[6] R. Riggio *et al.*, "Programming software-defined wireless networks," in *Proc. IEEE CNSM*, 2014, pp. 118–126.

[7] R. Riggio, T. Rasheed, and M. K. Marina, "Interference management in software-defined mobile networks," in *Proc. IEEE IM*, Ottawa, ON, Canada, 2015, to be published.

[8] N. McKeown *et al.*, "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. ACM WREN*, 2009, pp. 1–10.

[10] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing software-defined networks," in *Proc. USENIX NSDI*, 2013, pp. 1–14.

[11] A. Voellmy and P. Hudak, "Nettle: Taking the sting out of programming network routers," in *Proc. ACM PADL*, 2011, pp. 235–249.

[12] N. Foster *et al.*, "Frenetic: A network programming language," *SIGPLAN Notices*, 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. ACM HotSDN*, 2012, pp. 43–48.

[14] A. K. Nayak, A. Reimers, N. Feamster, and R. Clark, "Resonance: Dynamic access control for enterprise networks," in *Proc. ACM WREN*, 2009, pp. 11–18.

[15] C. Monsanto, N. Foster, R. Harrison, and D. Walker, "A compiler and runtime system for network programming languages," in *Proc. ACM POPL*, 2012, pp. 217–230.

[16] R. Cohen and G. Grebla, "Joint scheduling and fast cell selection in OFDMA wireless networks," *IEEE/ACM Trans. Netw.*, vol. 23, no. 1, pp. 114–125, Feb. 2015.

[17] J. Schulz-Zander *et al.*, "Programmatic orchestration of WiFi networks," in *Proc. USENIX ATC*, 2014, pp. 347–358.

[18] D. Lee *et al.*, "Coordinated multipoint transmission and reception in LTE-advanced: Deployment scenarios and operational challenges," *IEEE Commun. Mag.*, vol. 50, no. 2, pp. 148–155, Feb. 2012.

[19] S. H. Yeganeh and Y. Ganjali, "Kandoo: A framework for efficient and scalable offloading of control applications," in *Proc. ACM HotSDN*, 2012, pp. 19–24.

[20] J. Schulz-Zander, N. Sarrar, and S. Schmid, "Towards a scalable and near-sighted control plane architecture for WiFi SDNs," in *Proc. ACM HotSDN*, 2014, pp. 217–218.

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

[22] B. Pfaff *et al.*, "Extending networking into the virtualization layer," in *Prof. ACM HotNets*, 2009, pp. 1–17.

[23] 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.

[24] M. Bansal, J. Mehlman, S. Katti, and P. Levis, "OpenRadio: A programmable wireless dataplane," in *Proc. ACM HotSDN*, 2012, pp. 109–114.

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

[26] "POX." [Online]. Available: http://www.noxrepo.org/pox/about-pox/

[27] Y.-C. Cheng *et al.*, "Jigsaw: Solving the puzzle of enterprise 802.11 analysis," in *Proc. ACM SigComm*, 2006, pp. 39–50.

[28] P. San Segundo, "A new DSATUR-based algorithm for exact vertex coloring," *Comput. Oper. Res.*, vol. 39, no. 7, pp. 1724–1733, Jul. 2012.

[29] R. Murty, J. Padhye, A. Wolman, and M. Welsh, "Dyson: An architecture for extensible wireless LANs," in *Proc. USENIX ATC*, 2010, pp. 1–14.

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

[31] H. Ali-Ahmad *et al.*, "CROWD: An SDN approach for DenseNets," in *Proc. EWSDN*, 2013, pp. 25–31.

[32] G. Sun, G. Liu, H. Zhang, and W. Tan, "Architecture on mobility management in openflow-based radio access networks," in *Proc. IEEE GHTCE*, 2013, pp. 88–92.

[33] S. Ben Hadj Said *et al.*, "New control plane in 3GPP LTE/EPC architecture for on-demand connectivity service," in *Proc. IEEE CloudNet*, 2013, pp. 205–209.

[34] A. Patro and S. Banerjee, "COAP: A software-defined approach for home WLAN management through an open API," in *Proc. ACM MobiArch*, 2014, pp. 31–36.

[35] M. Casado, N. Foster, and A. Guha, "Abstractions for software-defined networks," *Commun. ACM*, vol. 57, no. 10, pp. 86–95, Sep. 2014.

[36] P. Cardieri, "Modeling interference in wireless ad hoc networks," *IEEE Commun. Surveys Tuts.*, vol. 12, no. 4, pp. 551–572, 4th Quart. 2010.

[37] D. Niculescu, "Interference map for 802.11 networks," in *Proc. ACM IMC*, 2007, pp. 339–350.

[38] C. Reis, R. Mahajan, M. Rodrig, D. Wetherall, and J. Zahorjan, "Measurement-based models of delivery and interference in static wireless networks," in *Proc. ACM SIGCOMM*, 2006, pp. 51–62.
[39] N. Ahmed, U. Ismail, S. Keshav, and K. Papagiannaki, "Online estimation of RF interference," in *Proc. ACM CoNEXT*, 2008, p. 4.
[40] V. Shrivastava, S. Rayanchu, S. Banerjee, and K. Papagiannaki, "PIE in the sky: Online passive interference estimation for enterprise WLANs," in *Proc. USENIX NSDI*, 2011, pp. 337–350.
[41] M. Vutukuru, K. Jamieson, and H. Balakrishnan, "Harnessing exposed terminals in wireless networks," in *Proc. USENIX NSDI*, 2008, pp. 59–72.
[42] R. Mahajan, M. Rodrig, D. Wetherall, and J. Zahorjan, "Analyzing the MAC-level behavior of wireless networks in the wild," *SIGCOMM Comput. Commun. Rev.*, vol. 36, no. 4, pp. 75–86, Aug. 2006.
[43] D. Giustiniano, D. Malone, D. Leith, and K. Papagiannaki, "Measuring transmission opportunities in 802.11 links," *IEEE Trans. Netw.*, vol. 18, no. 5, pp. 1516–1529, Oct. 2010.
[44] S. Rayanchu, V. Shrivastava, S. Banerjee, and R. Chandra, "Fluid: Improving throughputs in enterprise wireless lans through flexible channelization," in *Proc. ACM MobiCom*, 2011, pp. 1–12.
[45] N. Ahmed and S. Keshav, "SMARTA: A self-managing architecture for thin access points," in *Proc. ACM CoNEXT*, 2006, p. 9.
[46] V. Shrivastava *et al.*, "CENTAUR: Realizing the full potential of centralized WLANs through a hybrid data path," in *Proc. ACM MobiCom*, 2009, pp. 297–308.

Roberto Riggio (M'07) received the Ph.D. degree in communications engineering from the University of Trento, Trento, Italy, in 2008. He is currently a Senior Researcher of the Future Networks team at CREATE-NET, Italy and Guest Lecturer at the University of Trento. His research interests include performance isolation in multi-tenants datacenters, software defined mobile networking, and programmable 5G networks. He has co-authored more than 50 papers in internationally refereed journals and conferences. He serves on the TPC of leading conferences in the networking field, including, e.g., IEEE INFOCOM, IEEE IM/NOMS, IEEE ManFI, and IEEE ManSDN/NFV. He is a member of the ACM.

Mahesh K. Marina (SM'13) received the Ph.D. degree in computer science from the State University of New York at Stony Brook, Stony Brook, NY, USA. He is a Reader in the School of Informatics at the University of Edinburgh, Edinburgh, U.K. Prior to joining the University of Edinburgh, he spent two years as a Postdoctoral Researcher in the Computer Science Department at the University of California at Los Angeles (UCLA), Los Angeles, CA, USA. During 2013, he was a Visiting Researcher at ETH Zurich and at Ofcom London.

Julius Schulz-Zander is a Research Assistant in Prof. Feldmann's Internet Network Architectures group, pursuing the Ph.D. degree in computer science from the Technische Universität Berlin, Berlin, Germany, where he previously received the diploma degree in computer science. His research is centered around Wi-Fi, SDN, and NFV, i.e., software-defined wireless networking. Previously, he was a Visiting Scholar in Prof. McKeown's group at Stanford University, Stanford, CA, USA, where he brought OpenFlow to OpenWrt-enabled wireless routers. Since 2007, he has been a member of the Berlin Open Wireless Lab which operates over 100 OpenFlow-enabled APs indoor and outdoor. From 2007 to 2013, he was affiliated with Deutsche Telekom Innovation Laboratories. He was awarded a Software Campus grant from the German Federal Ministry of Education and Research to pursue his studies.

Slawomir Kuklinski (M'96) received the M.Sc. and the Ph.D. degrees from Warsaw University of Technology, Warsaw, Poland, in 1985 and 1994, respectively, where he has been an Assistant Professor since 1994. Since 2003, he also worked for Orange Labs Polska S.A. as a research expert. He has spent the decade working on Future Internet, mobile systems, wireless mesh networks, and VANET. Currently, he is focusing on autonomic and cognitive network management, SDN, and 5G. In his career he was involved in several European projects, including MIDAS, 4WARD, and EFIPSANS. He also led many national research projects as a principal investigator, published many conference papers, was TPC member of many conferences, and served as a reviewer to many conferences and journals.

Tinku Rasheed (M'05) received the Bachelor's degree in telecommunications from University of Kerala, India, in 2002, the M.S. degree from Aston University, Birmingham, U.K., in 2003, and the Ph.D. degree from the Computer Science Department of the University of Paris-Sud XI in 2007. He is a Senior Research staff member, heading the Future Networks Area within CREATE-NET. Before joining CREATE-NET, he was a Research Engineer with Orange Labs for 4 years. Dr. Rasheed has extensive industrial and academic research experience in the mobile wireless communication area, end-to-end network architectures and services. He has several granted patents and has published more than 60 articles in major journals and conferences. He is a member of the ACM.