Virtualization and Software-Defined Infrastructure Framework for Wireless Access Networks

Heming Wen

Department of Electrical & Computer Engineering
McGill University
Montreal, Canada

April 2014

A thesis submitted to McGill University in partial fulfilment of the requirements for the degree of Master of Engineering.

2014/04/01

Abstract

Virtualization and service-oriented architecture are important concepts that triggered the rapid evolution of cloud computing technologies. Network infrastructure virtualization is now possible by applying similar concepts in conjunction with recent advances in software-defined technologies. Since wireless technologies will play an important role in the future of networking technologies, this thesis proposes Aurora, a virtualization framework and testbed platform for supporting multiple types of virtualization techniques and architectures specifically applied on wireless technologies. In the first half of this thesis, a background survey of works in the recently-emerged field of "wireless virtualization" is made in order to identify potential applications, common trends and future research directions. This is followed by the presentation of three main perspectives: flow-based virtualization, protocol-based virtualization and spectrum-based virtualization. Then, a hypothetical ecosystem scenario of a future virtualized wireless infrastructure in which these perspectives coexist is discussed along with some of the challenges and requirements for a sustainable wireless virtualization framework. In the second half of this thesis, the general architecture and design principles behind Aurora are explained. Aurora is designed to fulfil multiple roles as a powerful tool to combine multiple wireless virtualization technologies, a research platform for developing new virtualization architectures and a service-oriented wireless infrastructure manager. Finally, the first iteration of the software implementation of Aurora inside the Smart Applications on Virtual Infrastructure (SAVI) testbed is exposed in order to demonstrate the feasibility of the framework.

Abrégé

La virtualisation et l'architecture orienté services sont des concepts importants qui ont déclenché l'évolution rapide des technologies de cloud computing. La virtualisation de l'infrastructure de réseaux est maintenant possible grâce à l'application de ces concepts et les avancées récentes dans les technologies définies par logiciel. Puisque les technologies sans-fil joueront un rôle important dans l'avenir des technologies de mise en réseau, cette thèse propose Aurora, un cadre et une plate-forme de banc d'essai pour soutenir plusieurs types de techniques et d'architecture de virtualisation spécifiquement appliquées sur les technologies sans-fil. Afin d'identifier les applications potentielles, les tendances et les orientations futures de la recherche, la première partie de cette thèse consiste d'une étude des travaux dans le domaine de la virtualisation sans-fil. Ceci est suivi par la présentation de trois perspectives différentes : la virtualisation basée sur les flux de paquets, la virtualisation basée sur le protocole de communication et la virtualisation basée sur le spectre de fréquences. Ensuite, un scénario hypothétique dans lequel ces perspectives coexistent est discuté en relation avec les défis et les exigences d'un cadre durable de virtualisation sans-fil. Dans la partie finale de cette thèse, l'architecture d'Aurora est dévoilée. Aurora est conçu comme un outil puissant pour combiner plusieurs technologies de virtualisation sans-fil. C'est aussi une plate-forme de recherche pour développer de nouvelles architectures de virtualisation et un gestionnaire de l'infrastructure sans-fil orienté services. Enfin, la mise en œuvre de Aurora à l'intérieur du banc d'essai SAVI est prèsentée afin de démontrer la faisabilité du cadre.

Acknowledgments

I would like to acknowledge my supervisor Professor Tho Le-Ngoc for both his moral and financial support of my research thesis. I am grateful for his valuable resourcefulness and guidance throughout my research and the writing of my thesis. I also like to thank our lab manager Robert Morawski for his technical knowledge and logistical assistance at acquiring hardware and equipments for the wireless virtualization testbed.

I thank my fellow graduate students in the Broadband Communications Laboratory at McGill University for their advices. More specifically, I greatly appreciate the help provided by my colleague and co-author of the book "Wireless Virtualization" Prabhat Kumar Tiwary. He has assisted me at reviewing my thesis and is already working on the next major extension of the framework at the network management level. Of course, I offer my deepest appreciations to our talented undegraduate students Kevin Han and Michael Smith. Their valuable software engineering skills has made the software implementation of the Aurora possible as they expertly crafted Aurora based on my architecture specifications. I look forward to their future contributions to the evolving open source Aurora project as part of the SAVI Wireless Testbed group at McGill University.

I am thankful to the main SAVI testbed research group at University of Toronto, especially Hadi Bannazadeh who has provided essential advice on the integration of Aurora with SAVI. From the Toronto group, I am also thankful of Jieyu (Eric) Lin and Thomas Lin, who provided debugging support when I encountered technical difficulties at using the SAVI testbed and during our group's demonstration at the SAVI annual general meeting.

I would like to express my gratitude to my parents and close friends for their invaluable non-academic support.

Finally, this work was partially supported by the Natural Sciences and Engineering Research Council (NSERC) through the NSERC Strategic Network for Smart Applications on Virtual Infrastructure (SAVI), and the Fonds québécois de la recherche sur la nature et les technologies (FQRNT) via a scholarship.

Contents

List of Figures

List of Tables

List of Acronyms

AP	Access Point (802.11)
API	Application Programming Interface
ASN	Access Service Network
BS	Base Station
BSS	Basic Service Set
C&M	Control and Management
CAPWAP	Control and Provisioning of Wireless Access Points
CDMA	Code-Division Multiple Access
CoMP	Coordinated Multi-Point
CPU	Central Processing Unit
DSP	Digital Signal Processing
E-UTRAN	Evolved Universal Terrestrial Radio Access Network
EPC	Evolved Packet Core
FDMA	Frequency-Division Multiple Access
FIRE	Future Internet Research and Experimentation
FMDA	Fibre-connected Massively-Distributed Antennas
FPGA	Field-Programmable Gate Array
GPRS	General Packet Radio Service
GPP	General Purpose Processor
GPU	Graphics Processing Unit
GTP	GPRS Tunneling Protocol
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
JSON	JavaScript Object Notation

LUT	Lookup Table
M2M	Machine-to-Machine
MCS	Modulation and Coding Scheme (802.11)
MPLS	Multiprotocol Label Switching
NaaS	Network-as-a-Service
NFV	Network Functions Virtualization
NGN	Next-Generation Network
NIC	Network Interface Card
OFDMA	Orthogonal Frequency-Division Multiple Access
ORBIT	Open-Access Research Testbed for Next-Generation Wireless Networks
OVS	Open vSwitch
PaaS	Platform-as-a-Service
PCF	Point Coordination Function
PCI	Peripheral Component Interconnect
PCIe	PCI Express
PRB	Physical Radio Block
PSM	Power-Saving Mode
QoS	Quality-of-Service
RAC	Resource Allocation Control
RCB	Radio Control Board
REST	Representational State Transfer
RFC	Request For Comment (IETF)
SAE	System Architecture Evolution
SAVI	Smart Applications on Virtual Infrastructure
SDI	Software-Defined Infrastructure
SDMA	Space-Division Multiple Access
SDN	Software-Defined Network(ing)
SDR	Software-Defined Radio
SIMD	Single Instruction Multiple Data
SLA	Service-Level Agreement
SNMP	Simple Network Management Protocol
SOA	Service-Oriented Architecture
SQL	Structured Query Language

SSID	Service Set Identifier
SVL	Spectrum Virtualization Layer
TDMA	Time-Division Multiple Access
UE	User Equipment
URI	Uniform Resource Identifier
UUID	Universally Unique Identifier
VAP	Virtual Access Point
vBTS	Virtual Base Transceiver System
VLAN	Virtual Local Area Network
VM	Virtual Machine
VMI	Virtualization Manager Interface
VMM	Virtual Machine Monitor
VNC	Virtual Network Controller
VPR	Virtualized Physical Resource
VRM	Virtual Resource Manager
WLAN	Wireless Local Area Network
WTP	Wireless Termination Point

Chapter 1

Introduction[1]

1.1 Current and Future Wireless Landscape

The current landscape for wireless technologies is evolving very rapidly. New emerging wireless protocols, services and applications are being introduced, such as machine-to-machine (M2M) communications for sensor networks and smart grid. Some of these applications are based on the re-purposing of existing concepts whereas others are inspired by entirely new research areas such as mobile cloud computing. In all cases, the main challenge of deploying new innovative services and applications on the existing wireless infrastructure is the fact that they are "locked-in" with the current monolithic and increasingly-complex hardware equipments. Many of these services and applications potentially require different quality-of-service (QoS) requirements and different degrees of control and management (C&M) in order to realize their full potential. The future wireless ecosystem will not only be heterogeneous in the sense of having *different* cell sizes, but also in terms of the *coexistence* and *convergence* of *different* wireless technologies and *different* services and applications with *different* requirements. Thus, the future wireless landscape will be dominated by a *dense*, *ubiquitous* and *heterogeneous* network. The development, deployment and management of such a diverse and rich wireless ecosystem will become a major challenge to solve.

At the same time, there are many new emerging research areas and associated technologies in computing, networking and wireless telecommunications. On the computing

[1]Parts of this chapter have been presented at the 2013 IEEE Third International Conference on Selected Topics in Mobile and Wireless Networking (MoWNet'2013) in Montreal, Canada [1] and published as part of the SpringBriefs in Computer Science series [2].

side, *cloud computing* and *server virtualization* have revolutionized the IT industry in the past few years. More recently, the paradigm shift in the service model for server resources has been extended to the network infrastructure with the introduction of concepts such as *software-defined networking* (SDN) and *network virtualization*. On the business side, the concept of *network functions virtualization* (NFV), which consists of relocating network functionalities into the cloud and keeping hardware equipments more generic, is being promoted by major telecommunications operators around the world [3]. This trend leads to the development of a new system architecture and business model that focuses on offering virtualized hardware resources as a *service*. Clearly, this will deeply impact the evolution of the next-generation telecommunications infrastructure. On the wireless telecommunications side, new radio technologies and wireless transmission techniques, ranging from software-defined radio (SDR) and cognitive radio to coordinated multi-point (CoMP), are aimed at making the wireless communications infrastructure more dynamic and efficient. At last, advancements in optical fibre technologies make architectures such as the fibre-connected massively-distributed antennas (FMDA) system feasible [4]. All these technical advancements are like pieces of a larger puzzle that have the potential to generate a "perfect storm" which could result in a major revolution of the wireless landscape: the transition and convergence of the wireless infrastructure into a *virtualized*, *software-defined* and *service-oriented* architecture.

1.2 Virtualized, Software-Defined and Service-Oriented Paradigms

For the development of the next-generation network (NGN) and infrastructure, including the wireless infrastructure, there are three main overlapping but different concepts at work: *virtualization*, *software-defined* technologies and *service-oriented* infrastructure. These concepts combine together to form the basis of the modern concept of an *extended cloud infrastructure*.

First, *virtualization* is a concept that allows the *abstraction*, *sharing* and *partitioning* of resources. The goal of virtualization is to enable a more *dynamic* and *efficient* reuse of resources. In the so-called *server* or *computer virtualization*, computing resources such as processors, memory and storage are shared. For *network virtualization*, the network switching fabric and backplane is shared. In the case of *wireless virtualization*, the main topic of this thesis, the *wireless resources* are shared. The wireless resources can include the wireless network, the wireless access hardware such as access points (AP), basestations (BS) and wireless network interface cards (NIC), as well as the frequency spectrum itself. The virtualization of different types of wireless resources is explored in Chapter 2 and 3 of this thesis. The terminology pertaining to wireless virtualization used in this thesis is drawn from similar or equivalent terminology used in virtualization and cloud computing. A *hypervisor*, also called a *virtual machine monitor* (VMM), is defined as the management layer that resolves conflict and performs resource allocation among *tenants*. The location and implementation of the hypervisor is one important aspect that differentiates one virtualization approach from another. The virtual instance of a set of shared resources allocated to a *tenant* is called a *slice*. A *tenant* is defined as the owner (individual, *group* of individuals or organization) of the *virtual* wireless infrastructure. As such, a tenant should not be confounded with *mobile clients*, who use the wireless services provided by the tenants.

On the other hand, *software-defined* technologies consist of decoupling functionalities from the hardware and providing external *software interfaces* to control these functionalities. The purpose of the software-defined paradigm is to enhance the *flexibility* and *programmability* of the hardware infrastructure. In the case of SDN, the control and processing of network functionalities can be performed by an external controller. In the case of SDR, the baseband processing and wireless protocol stacks can be partially or fully implemented using software modules. Software-defined technologies are often key enablers to the implementation of virtualization in non-computing hardware such as the switching fabric and the wireless access equipments. This is due to the fact that abstraction and sharing are simpler to implement and manage in software than in highly-specialized hardware. In other words, although software-defined technologies are not a strict requirement for virtualization, they can facilitate its implementation.

Finally, the *service-oriented architecture* (SOA) can be applied on the wireless infrastructure both as a system architecture and a business model to offer and sell the abstracted and virtualized resources as *services* to an external party, generically referred to as a *tenant*. For instance, the mobile virtual network operator (MVNO) in the traditional cellular network can be considered as one type of tenant. The service-oriented paradigm aims at keeping the infrastructure functionalities and resources highly *modularized*, *reusable* and *relocatable* ubiquitously across the infrastructure. When applied together with virtualization, each tenant is given the illusion of ownership over its slice defined by a contract such as a *service-level agreement* (SLA). Overall, the combination of these three concepts should allow a more sustainable, flexible and efficient way of using the hardware infrastructure. The modern *cloud computing* is an example of the result of the interplay between these concepts in the computing domain. However, wireless virtualization can be quite different from computer technologies due to some fundamental differences between the ways how wireless technologies and computing technologies are used.

1.3 Motivation for Wireless Virtualization

With the ongoing research on network virtualization and SDN, it is important to determine whether the sole application of network virtualization technologies and techniques, which are mostly designed for the wired network, is sufficient for the future infrastructure. Clearly, the wireless medium, wireless technologies and mobile services behave differently from their wired counterparts. Thus, virtualization technologies and techniques that target specifically wireless technologies must be considered. In addition, wireless virtualization has a wide range of potential benefits in both commercial and academic contexts by enabling a *flexible reuse* of the existing wireless infrastructure.

Commercially, wireless virtualization, when combined with software-defined technologies and the service-oriented paradigm, can lower the capital expenditures and the barrier to entry for emergent wireless service providers by allowing them to dynamically share the existing infrastructure [5]. Infrastructure virtualization offers the tenant service providers a *scalable*, *relocatable* and *on-demand* deployment of *virtual infrastructure*. It also allows them to offer fully-differentiated services to their clients by having total control over their own *virtual wireless infrastructure* without having to own the *physical wireless infrastructure*. This feature is especially useful in the case of smart grid communications, where the utility provider wants to control and manage the telecommunications network in order to provide a critical and essential service without the need to rebuild its own communications infrastructure. Additionally, wireless virtualization can also allow various wireless functions to be decoupled from the hardware and implemented in the cloud [3][6]. This allows the high-volume processing capabilities of the cloud to be leveraged by these functions. This also enables the hardware equipment to be reprogrammable instead of specialized, leading to faster deployment of new technologies. Hence, a more integrated, innovative, diverse and competitive ecosystem is created [7]. In a sense, virtualization allows new applications that require different highly-customized cross-layer network capabilities to be deployed over a common infrastructure.

In terms of academic research, virtualization is already extensively applied in the Future Internet initiative and the *clean-slate design* [8]. The main reason for virtualization to be used in research is the level of flexibility that can be achieved with virtualized experimentation testbeds. Virtualization can shorten the research and development life cycle of new wireless technologies by providing a more open and flexible infrastructure [9]. Since the virtualized infrastructure can be shared among multiple tenants isolated from each other, it can enable the testing and deployment of experimental functionalities on a real production infrastructure without interrupting its regular operations. Furthermore, wireless virtualization can both leverage from and impact many active research areas in wireless communications, ranging from SDR to cognitive radio technologies. Whereas some technologies can make wireless virtualization feasible, other technologies can benefit from a virtualized environment. These interdependencies between wireless virtualization and other technologies is explored in Chapter 2. In short summary, wireless virtualization is one possible method to simplify the development, deployment and management of a dense and heterogeneous wireless ecosystem in both commercial and academic contexts. Such a hypothetical ecosystem is presented in the following section.

1.4 A Hypothetical Virtualized Wireless Ecosystem

A hypothetical futuristic virtualized infrastructure is envisioned as shown in Figure 1.1. The hypothetical scenario predicts the roles of different actors with the introduction of a virtualized, software-defined and service-oriented wireless infrastructure. In this scenario, Eric is the equipment manufacturer. He provides the high-performance and *virtualization-enabled* wireless hardware equipment along with abstraction, control and management *interfaces* for his equipments. These interfaces acts as an abstraction and management mechanism to the high-performance hardware. Eric also has the choice of developing and selling a *virtualization platform*, a software suite of virtualization management tools and *infrastructure operating system*. Since he has the advantage of producing the hardware, he can include unique functionalities that can differentiate himself from other hardware and virtualization platform vendors. Otherwise, third-party companies can develop these platforms, which can access the hardware through the available interfaces.

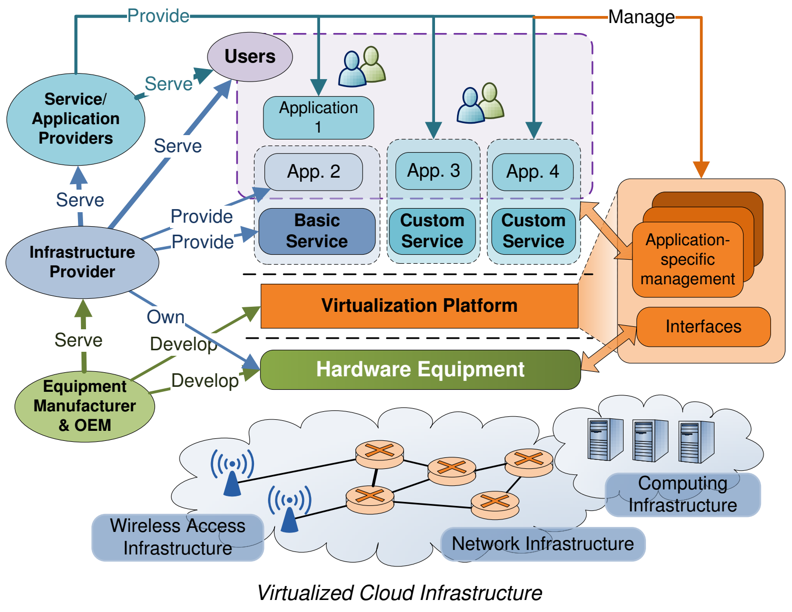

Fig. 1.1: Hypothetical Virtualized Infrastructure Scenario

Irene, the infrastructure provider, owns the hardware infrastructure bought from Eric. Irene can offer virtual instances of the infrastructure to the service, application and content providers Sara and Simon. Of course, Irene herself can act as a basic default service provider. However, she also sells to Sara and Simon the right to provide their services over the same infrastructure without having to own it. Sara and Simon, who traditionally have less control over the delivery of their service or content through the network and wireless infrastructure, are now able to use customized network topologies and optimized protocols to support their services residing entirely in the cloud. They will be able to fully manage their slice via an application-specific management interface without being affected from each other. In fact, Irene, Sara and Simon can actually belong to the same organization. In other words, even *within* a single commercial entity, the virtualized infrastructure can be used to provide agility and decoupling between different groups. For example, the hypothetical infrastructure can support different service teams decoupled from the infrastructure team.

Ultimately, the end users Alice and Bob are unaware of the interaction between the infrastructure and service providers. Alice and Bob simply use customized applications and services that take advantage of the fully virtualized infrastructure. Since these enhanced services are perfectly integrated within the virtualized and ubiquitous wireless access network, Alice and Bob are able to access and use them any time and anywhere. Additionally, virtualization can allow access points bought by Alice and Bob to run multiple virtual infrastructure applications provided by service providers Sara and Simon, giving the providers access to the functionalities of user-bought equipment to enhance their services.

While this scenario might sound similar to network sharing [10], there are important differences such as the degree of *control* and *freedom* over the virtualized resources. Finally, such a scenario might seem far-fetched but similar ideas are discussed and analysed in [6] and [11]. The best analogy of this scenario would be a building owner (i.e. infrastructure owner) leasing out rooms to tenants (i.e. service providers) whereas the tenants can then open outlets to sell their services and goods to customers (i.e. mobile users).

1.5 Thesis Contribution and Organization

Wireless virtualization is still at its early stage of research and development. Thus, the approaches to wireless virtualization are numerous and multi-dimensional. There is not yet a single definite approach that stands out. This thesis first attempts to provide a more comprehensive picture on the different aspects of wireless virtualization and identify how it is related to other emerging research areas. The first contribution of this thesis is a *classification* of different wireless virtualization approaches into three levels. However, instead of proposing a new virtualization technique or arguing for the adoption of one particular perspective, this thesis embraces the idea of *coexistence* between different virtualization perspective targeted for different needs. The most important contribution is the formulation of a *generic* and *evolvable* wireless virtualization *framework* that is able to support and integrate different wireless virtualization perspectives on *virtualized radio nodes*. This framework differentiates itself from other existing frameworks such that it is both a flexible *testbed platform* and a powerful *suite of tools* similar to a wireless infrastructure operating system that enable full reconfigurability of the wireless infrastructure, most particularly radio nodes. This contribution is important because it provides a workable *platform* on which different wireless virtualization technologies and techniques can be *integrated*, *developed* and *experimented on*. One major selling point of the framework, other than its extensibility and high degree of customization, is that it adds no overhead to the performance of the virtualization technologies it integrates. The final contribution of this thesis is the implementation of the proposed as an extensible software platform adapted for 802.11

wireless local area network (WLAN) technologies. The rest of the thesis is organized as follows:

Chapter 2 presents a background overview of existing research topics in wireless virtualization and its different aspects. The topics covered in Chapter 2 include various techniques and emerging technologies that can be used to implement wireless virtualization, such as the use of multiple access techniques and the application of software-defined technologies. General wireless virtualization architectures in the context of Future Internet testbeds is explored. The virtualization of specific wireless technologies, notably 802.11 WLAN, WiMAX and Long Term Evolution (LTE) is also discussed. In Chapter 3, a framework for the classification of different wireless virtualization approaches is shown. Three main approaches are identified: flow-based virtualization, protocol-based virtualization and spectrum-based virtualization. The coexistence of these virtualization perspectives is discussed in the context of a virtualized, software-defined and service-oriented wireless infrastructure. A generic wireless virtualization framework that can encompass all these perspectives in a multi-perspective and evolutionary manner is identified, along with its benefits, justifications, challenges and requirements. The general architecture of such a framework is outlined.

In Chapter 4, the implementation architecture of a virtualization and software-defined infrastructure software platform for wireless access networks, codenamed *Aurora*, is presented. The main purpose of Aurora is to provide a *modular* and *evolvable* resource *abstraction*, *virtualization* and *orchestration* platform for the wireless infrastructure. The foundation of the platform is outlined. More specifically, this thesis focuses on the *Aurora-Agent*, the *local virtualization agent* of the architecture, and the virtualization of *wireless resource nodes*. Aurora-AP, a virtualization agent designed for OpenWrt-based 802.11 access points is presented. In Chapter 5, the implementation and integration of Aurora with the existing OpenStack cloud infrastructure platform and the Smart Applications on Virtual Infrastructure (SAVI) testbed are discussed. The implementation of flow-based datapath virtualization with Aurora-AP is detailed. Possible network integration and datapath configurations are shown in order to demonstrate the flexibility of the architecture, along with an example OpenFlow application supported over Aurora. The extensions of Aurora to SDRs are discussed. At last, Chapter 6 concludes with a summary of the main points of the thesis, including a synthesis of the essentials of the concept of wireless virtualization. Some suggestions on the future direction of research in the field of wireless virtualization are highlighted.

Chapter 2

Current Trends in Wireless Virtualization[1]

Before formulating a framework for wireless virtualization, it is important to understand the scope of ongoing research in wireless virtualization and its related technologies. First, the so-called *wireless virtualization* borrows many aspects taken from virtualization in general. Comprehensive survey of emerging computer networking technologies, including network virtualization, is provided in [8]. The survey [12] identifies wireless virtualization as one of the *frontier* emerging research areas in network virtualization. However, as mentioned in Chapter 1, while the fundamental concept of both network and wireless virtualization remains the same, there are significant differences between their approaches. This chapter provides a background literature *overview* of some recent wireless and infrastructure virtualization trends. A more detailed presentation of these technologies and techniques can be found in [2].

2.1 Different Aspects of Wireless Virtualization and Related Technologies

Since wireless virtualization is a relatively young and active area of research, a framework for classifying all the recent advances in wireless virtualization technologies is quite difficult

[1]Parts of this chapter have been presented at the 2013 IEEE Third International Conference on Selected Topics in Mobile and Wireless Networking (MoWNet'2013) in Montreal, Canada [1] and published as part of the SpringBriefs in Computer Science series [2].

to realize. In this thesis, wireless virtualization is considered a *multi-dimensional* concept broken down to many different *aspects* that are explored in this chapter. This section summarizes the main aspects that are taken into account during the formulation of a *virtualization framework* in Chapter 3 of this thesis.

2.1.1 Scope and Depth of Wireless Virtualization

The *scope* and *depth* of virtualization for wireless resources are important architecture design choices that define the types of wireless resources being virtualized. The *scope* defines which device, technology or logical entity is being virtualized. In terms of scope, virtualization can be applied on a network-wide scope or on the localized access hardware (access point or basestation) and the client hardware (wireless NIC). On the other hand, the *depth* of virtualization is the extent of penetration of slicing and partitioning of the wireless resources on a given device. It defines the granularity of the virtualized resources supported in a given architecture. The capabilities and the level of sharing of resources will vary depending on how deep the virtualization is applied.

2.1.2 Virtualization in Different Wireless Standards

Technology-specific bindings of virtualization are important due to the need to preserve efficiency in the unpredictable multi-user multi-accessed wireless medium. Unlike in server and network virtualization, there is not one single dominant wireless technology, but a few major ones. It is also important to understand that not all these technologies benefit equally from various wireless virtualization perspectives. The benefits of virtualization are most apparent in technologies where the supported bandwidth and the supported number of users are relatively high, leaving enough room for dynamic sharing of resources, such as in 802.11 WLAN and cellular networks. In general, cellular technologies provide more advanced network control and management as an infrastructure-grade technology compared to the relatively simpler WLAN technologies. Some of the existing features of these standards, such as *virtual access point* (VAP) in 802.11 and *network sharing* in LTE, can be used as a basis starting point to support more advanced wireless virtualization, as will be shown in Section 2.5 and 2.6.

2.1.3 Virtualization and Other Research Areas in Wireless Communications

Wireless virtualization is both affected by and can affect other research areas in wireless communications technologies. In some cases, the feasibility of virtualization is positively impacted by areas such as software-defined technologies (Section 2.4 and 2.8). In other cases, there is a synergy between virtualization and other concepts such as cognitive radio (Section 2.9). The following sections will examine different aspects of wireless virtualization as well as its interaction with other research areas and technologies.

2.2 Future Internet Testbeds Using Virtualization

This section covers virtualization applied in next-generation network (NGN) testbeds, including wireless virtualization. These international research initiatives on Future Internet are initially motivated by the need for a change in the current Internet architecture. For instance, the increasing demand for mobility and the shift from host-to-host applications to content-oriented applications prompt the development of new network paradigms [8]. Thus, one objective of NGN testbeds is to overhaul the existing infrastructure in order to support a smarter and more open infrastructure, in this case, through *infrastructure virtualization*. Gradually, focus also shifted towards wireless technologies, which are major players that have to be considered in the foreseeable Future Internet landscape. Thus, NGN testbeds can provide a context in which wireless virtualization can be applied and integrated in.

2.2.1 Global Environment for Network Innovations (GENI)

GENI is currently one of the largest and most complex virtualization-enabled testbed federation in development, initiated by the National Science Foundation. In fact, GENI is described as a *suite of research infrastructure* [13]. The basic requirements of GENI are based on two main concepts: *resource sharing through virtualization* and a *federation ecosystem* of different testbeds among universities across the United States of America [13]. The virtualization architecture of GENI is mostly focused on the slicing and isolation of different experimentations, the main function of the testbed. The inclusion of virtualized wireless technologies as part of the infrastructure is initially handled by the Open-Access Research Testbed for Next-Generation Wireless Networks (ORBIT).

The GENI testbed meta-architecture can be divided into three main sections: the federation of aggregates of virtualized components, the clearinghouse management registries and the experimentation tool services [8]. The component aggregates are the main resource pools in which multiple components are grouped under the control of an aggregate manager that share them through virtualization. The clearinghouse registries provide the book-keeping and security features required for user authentication and slice configuration. The different aggregates must communicate and establish trust relationship with the clearinghouse before allocating resources to a user. Finally, the tools and services contain the important software developed for the GENI tenants to monitor, control and debug their experiments.

2.2.2 Smart Applications on Virtual Infrastructure (SAVI)

Smart Application on Virtual Infrastructure (SAVI) is an experimental testbed under development by the joint effort of Canadian industry and academia [14]. In some sense, SAVI can be considered the Canadian counterpart of GENI. SAVI puts a high emphasis on cloud infrastructure, flexible high-speed wireless access networks and low energy footprint. The goal of SAVI is to address the design of future application platform built on a flexible infrastructure consisting of heterogeneous resources. SAVI is initially based on an extension of the Virtualized Application Networking Infrastructure (VANI), a virtualization testbed implemented by University of Toronto [15].

SAVI is divided into three major components which are the network fabric, the control and management (C&M) center and the resource nodes. There are two types of network fabric supported: over the Internet and over a dedicated optical backbone. The testbed-wide C&M center is a suite of software that includes a clearinghouse system, a testbed monitoring and measurement system, a resource allocation system and a web portal server. Each core and edge node also has its own local C&M system. The management framework extends the OpenStack open source cloud computing platform [16] (see Subsection 2.2.4) to include new types of virtualized resources such as field-programmable gate array (FPGA), graphics processing unit (GPU) and wireless access points[2]. The virtualization of each type of resources is performed by a resource-specific *virtualization agent* [15]. The architecture of the testbed is shown on Figure 2.1.

[2]This thesis is part of the Theme 4 of the SAVI Research Group, which handles the wireless access technologies inside the SAVI testbed. Thus, one of the objectives of this thesis is to integrate virtualized wireless resources into SAVI.

The testbed is divided into two planes: the C&M plane and the application and experiment (A&E) plane [17]. The C&M plane performs the control and setup of the infrastructure whereas the A&E plane runs the tenant applications and experiments. The resource nodes of SAVI are divided into three levels: core, edge and access [17]. Each core node is composed of a large-scale datacenter preferably located close to a source of renewable energy. On the other hand, edge nodes are smaller local datacenters hosted in participating universities. They can contain non-computing resources in addition to scaled-down storage and processing resources. Access nodes are similar to edge nodes with the exception that they have *virtualized wireless* and *optical access* resources. A tenant can configure its own slice of the infrastructure, called a project, by sending RESTful application programming interface (API) calls to the edge controller. The controller or manager then allocates the desired resources and establish their connectivity to the virtual network of the tenant. More details on the SAVI testbed is presented when the deployment of the proposed framework of this thesis is discussed in Section 5.2.

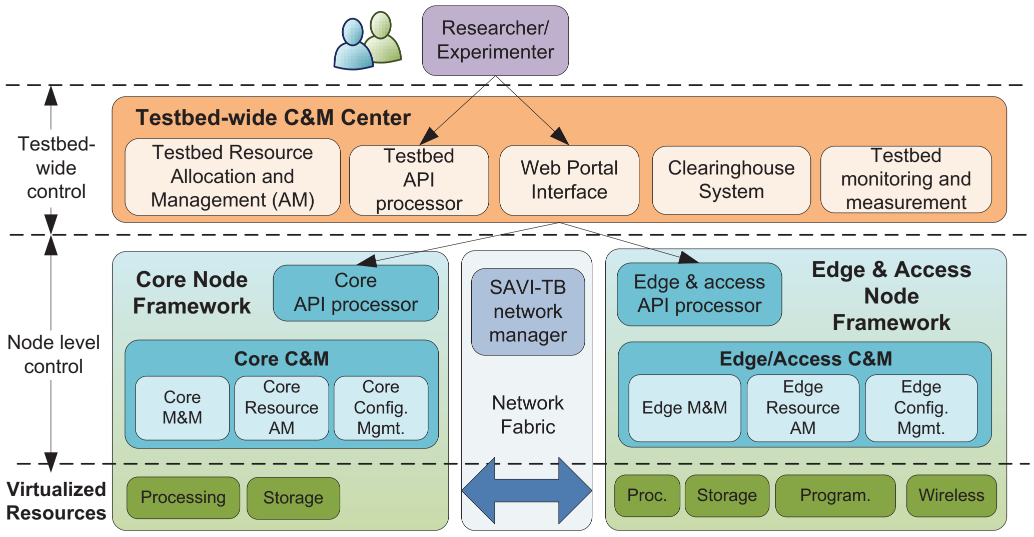

Fig. 2.1: Simplified SAVI Testbed Architecture

2.2.3 General Architecture for Infrastructure Virtualization

Both GENI and SAVI follow a generalized infrastructure virtualization and orchestration architecture. Some generalized control and management frameworks that includes wireless virtualization are proposed in [18] and [9]. In [18], the virtualization architecture is broken down into three general components: the virtualized physical resources (VPR), the virtual resource manager (VRM) and the virtual network controller (VNC). Each type of VPR has its own standardized interface since they can offer different types of services and functionalities. The VRM is the hypervisor that reinforces the QoS requirements of each slice determined by a network service-level agreement (SLA) negotiated between the tenant and the infrastructure provider. It can act as a *resource broker*. The VNCs are used by the tenants to control and manage their own slice of the virtualized infrastructure. This architecture is outlined in Figure 2.2 and bear resemblance to both GENI and SAVI.

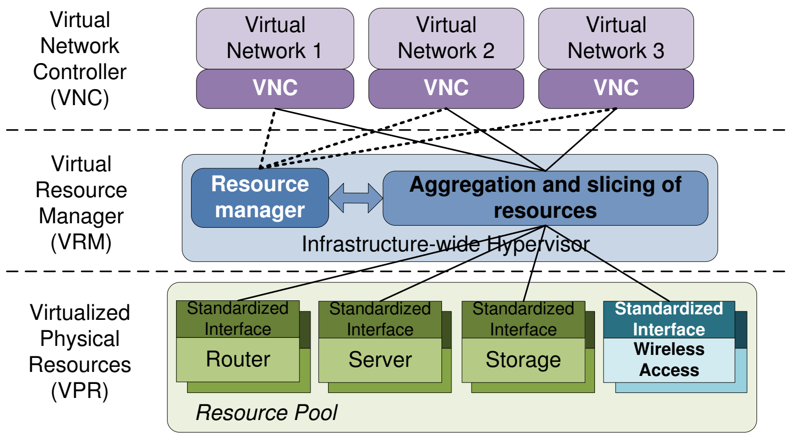

Fig. 2.2: Infrastructure Virtualization With Different Resources

In contrast with [18], which presents an architecture for infrastructure virtualization in general, [9] specifically targets the virtualization of the wireless resource, defined as a radio node. In this case, the resources are *radio resource blocks* delimited by time and frequency. The resources are allocated through algorithms implemented in a resource allocation control (RAC) layer. A virtualization manager interface (VMI) resides in each radio node and acts as a broker and coordinator for tenants to interface with the RAC. In this hypothetical architecture, each tenant interacts with its own *virtual radio node*, which can be completely different wireless protocol stacks. This architecture is outlined in Figure 2.3. Techniques that can support such architecture are discussed in Section 2.8 and 2.9. Overall, these frameworks provide a global perspective on the design, deployment, control and management of the virtualized wireless resources.

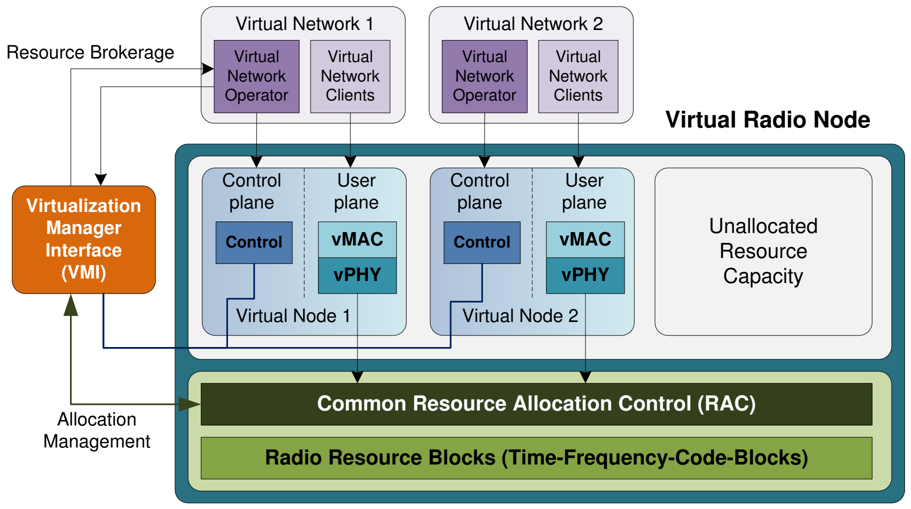

Fig. 2.3: Configurable Virtual Radio Node

2.2.4 OpenStack Cloud Infrastructure Platform

OpenStack is an open source cloud management and orchestration platform initially formed by three main software components: Nova, Swift and Glance [16]. Nova manages virtual machines (VMs) for cloud computing with VM images provided through Glance whereas Swift manages virtualized storage. A fourth component dedicated to networking, Neutron [19] (previously known as Quantum), supports extensible and differentiated network applications and services over a virtualized cloud infrastructure, including technologies such as OpenFlow. There is no direct support for wireless networks inside OpenStack, which prompted the development of Aurora in Chapter 4 of this thesis.

2.3 Multiple Access Techniques in Wireless Virtualization

Virtualization provides the *illusion* of sharing the resources. Ultimately, all high-level resource virtualization eventually breaks down into low-level resource partitioning, as it is fundamentally impossible to truly share an irreducible physical resource. In the case of wireless virtualization, the most fundamental resources is based on the space-time-frequency access to the wireless medium. The allocation and multiplexing of these resources are well known as *multiple access* and *multiplexing* techniques. These techniques consist of partitioning the time, space or frequency dimensions of the channel and allocating them to different users or traffic flows. In some sense, all wireless virtualization architectures rely on some combination of multiple access techniques, whether by design or as a by-product of the implementation. A basic guideline on the usage of different multiple access techniques for different wireless applications depending on the requirements of the application is presented in [20].

Each multiple access technique has its own set of trade-off in virtualization. Different techniques are used to conserve different types of resources [21]. For example, frequency-division multiple access (FDMA) can conserve the spatial resources by allowing multiple tenants to reuse the same node location [21]. Alternatively, time-division multiple access (TDMA) can be used to allocate each tenant to a time slot but with the same frequency, conserving frequency resources [21], at the cost of some additional context-switching time in the order of milliseconds [20]. Space-division multiple access (SDMA) is a third possible technique presented in [20]. In the ORBIT testbed, it is applied to allow different experiments to run on different portions of the grid. SDMA virtualization can also refer to the allocation of different spatial streams of a MIMO system to each tenant although there are limitations in the flexibility of such a design.

Hybrid access methods that use more than a single dimension can often provide a more efficient and flexible virtualization. For instance, orthogonal frequency-division multiple access (OFDMA) is a multi-carrier multiple access technique that allows multiple users to seamlessly share different time-frequency blocks without any switching delay or interference penalties originally described in [20] and [21]. In OFDMA, inter-channel interference can be avoided by preserving orthogonality between resource blocks in frequency and inter-symbol interference can be minimized through the use of guard intervals and cyclic prefix between resource blocks in time. The OFDMA scheduling is performed by the downlink air interface MAC scheduler in modern cellular technologies and is discussed in Section 2.6.

2.4 Wireless Virtualization and Software-Defined Networking With OpenFlow

As mentioned in Section 1.2, one of the main goals of software-defined networking (SDN) is to enhance the programmability of the network fabric and decouple the functionalities from hardware to software. Since the network fabric is the backbone of the wireless infrastructure, there are specific applications of SDN technologies in a network that includes wireless access nodes, often called a *mobile network*.

2.4.1 OpenFlow in SDN and Network Virtualization

Software-defined networking (SDN) enablers such as the OpenFlow protocol [22] allow an external software controller to access the switching functions of the network fabric using flow-based instruction rules. An OpenFlow-compliant switch, either a hardware switch or a software switch such as Open vSwitch (OVS) [23], must have a set of flow tables and support for the OpenFlow protocol [24]. A *flow* is defined by any combination of the header subfields from layer 1 to layer 4 along with possible wildcard fields. The basic operation of the protocol is extremely simple. An ingress packet is first matched with the headers of existing flow entries in the flow table. If a longest prefix match is found, the flow counter is incremented and a list of actions prescribed for that entry is executed. If a match is not found, the packet is forwarded to the controller using the OpenFlow protocol. The controller can then decide to create a new entry in the flow table with a set of prescribed flow actions or additional instructions (as of the version 1.1 of the standard [25]). The semi-persistent state of the flow table allows the packets to be processed at line rate while being managed by a centralized controller. Thus, the protocol effectively decouples the data plane from the control plane. More recent versions have added support for IPv6 [26] and quality-of-service (QoS) control through meter tables [27].

An OpenFlow *controller* is a software platform that offers a set of APIs for developers to use the OpenFlow commands to exert SDN. A single controller can be connected to a network of switches, each having its own datapath identification (DPID). The controller usually offers an event-based platform on event callback can be attached. However, OpenFlow by itself does not imply virtualization. Network virtualization with OpenFlow is achieved by using an OpenFlow hypervisor such as FlowVisor [28]. FlowVisor is a Java-based controller that enables multiple OpenFlow *guest controllers* to share the same set of switches. It acts as an intermediary translation unit between guest controllers and the switching fabric. Therefore, each tenant can control and manage the SDN in its own slice of the virtualized infrastructure.

Overall, OpenFlow can be considered as a form of datapath virtualization technology. Other widely-known datapath virtualization techniques include virtual local area network (VLAN) and multiprotocol label switching (MPLS). The advantage of OpenFlow is that its basic software-defined operations do not rely on the *labeling* of flows, allowing them to travel through the network fabric *unmodified*. This makes it more flexible and extensive than most other datapath virtualization technologies. In fact, both VLAN and MPLS can be supported over OpenFlow. Currently, OpenFlow is compatible with other virtualization frameworks and technologies, such as OVS and OpenStack. Some of the challenges with OpenFlow consist of scalability and fault-tolerance. Clearly, a purely centralized model suffer from scalability and bottleneck issues because of the overhead and delay during each flow setup. Some alternative *distributed* OpenFlow architecture such as DevoFlow attempts to solve these issues [29].

2.4.2 OpenFlow in Wireless Technologies

OpenFlow can also be applied on wireless technologies in order to integrate wireless access and mobility functionalities to the SDN framework. OpenFlow Wireless, also known as OpenRoads, is the addition of OpenFlow-enabled plug-ins on wireless access points and basestations [7]. OpenRoads enables OpenFlow by installing a software switch (OVS) inside the wireless access point firmware and using the simple network management protocol (SNMP) to configure wireless parameters. FlowVisor and SNMP Visor (configuration hypervisor) work in conjunction as shown in Figure 2.4. OpenRoads is based on the NOX controller [30] although other guest controllers can also be used through FlowVisor. Unfortunately, there are limitations with OpenRoads. First, OpenFlow cannot affect any wireless functionality, leaving all configuration tasks to the SNMP. However, SNMP is a generic management protocol not sufficiently specialized to handle all wireless functionalities. For instance, conflicting configurations such as different power levels on the same wireless access point cannot be resolved. Thus, efficient wireless resource allocation and interference management cannot be achieved without more enhancements in the radio hardware. Nevertheless, OpenRoads remains very important because it *extends* datapath virtualization to include the wireless access nodes.

In frequent cases, OpenFlow and SDN are mainly used to increase the flexibility and mobility of the wireless network. For instance, SDN technologies and OpenFlow have been applied to wireless technologies in various ways either to enable virtualization [31] or to improve existing functionalities [32]. In both of these cases, OpenFlow only operates on the switching fabric and not on the wireless hardware. In [31], CloudMAC, an OpenFlow-

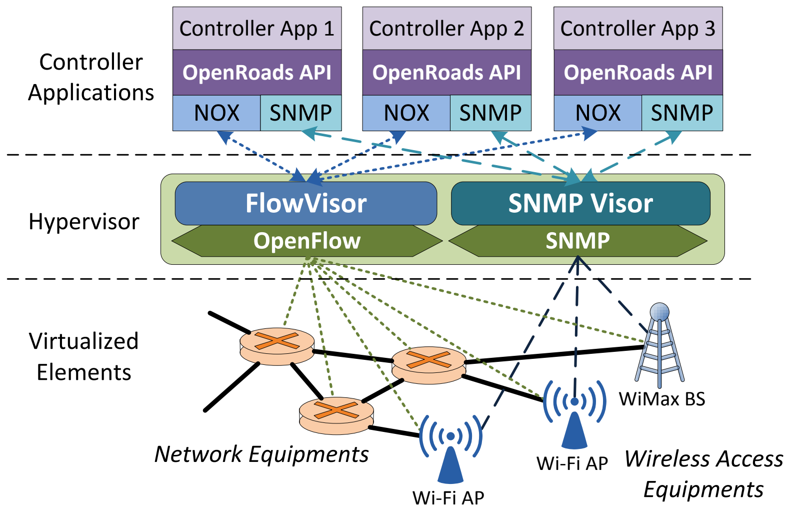

Fig. 2.4: Virtualization with OpenFlow Wireless

based architecture, is used to support the forwarding of 802.11 MAC frames from access points to virtualized servers for processing, partially *relocating* 802.11 functionalities into the cloud. CloudMAC is composed of four main components: the virtual access point (VAP), the wireless termination point (WTP), the OpenFlow-enabled network fabric and an OpenFlow controller. The time-critical functions are processed locally by the WTP on the AP hardware. The other functions are delegated to the wireless firmware running on VMs. Here, the key role of OpenFlow is to dynamically associate WTPs with VAPs by redirecting and reconfiguring the layer-2 tunnels between them. This enables on-demand allocation of new WTPs to a VAP. A single WTP can also be connected to multiple VAPs identified by their MAC address. This effectively enables a single WTP to be shared among multiple VAPs. The OpenFlow controller is also accompanied by an infrastructure-wide hypervisor which manages control and management frames and stores network slicing policies. It reinforces these policies by intercepting and overwriting control headers exchanged between the WTPs and the VAPs through OpenFlow. The CloudMAC architecture is illustrated in Figure 2.5.

In the case of [32], OpenFlow is used to enhance the mobility management of user equipments (UEs) in LTE network by replacing the GPRS tunneling protocol (GTP) by a dynamic OpenFlow-based mobility anchoring system. Since GTP tunnels must be torn down and re-established after each handover, additional signaling overhead is incurred.

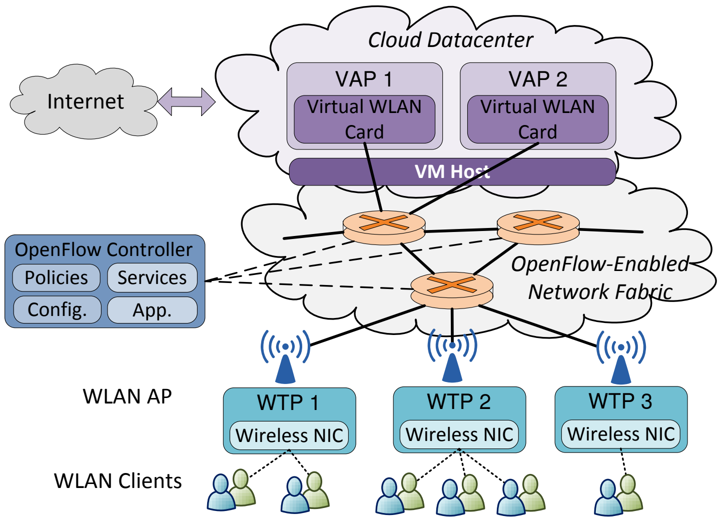

Fig. 2.5: WLAN Virtualization with CloudMAC

OpenFlow can avoid this issue by dynamically allocating *anchor points* that maintains the traffic flow. Only traffic flows between the UEs and the anchor point are affected during a handover, considerably reducing the signaling traffic [32].

2.5 Virtualization in 802.11 Technologies

Compared to cellular technologies, IEEE 802.11 WLAN standards are designed to support faster and easier deployment in a plug-and-play manner. However, as a trade-off, the control and management framework is relatively less sophisticated. In the *infrastructure* mode, WLAN APs can coordinate multiple client devices within their coverage area, just like a cellular basestation. However, there is only support for priority-based QoS and not for scheduling. On the other hand, direct peer-to-peer connection among 802.11 devices is allowed. Thus, applications such as mesh networking are possible. Virtualization becomes an extremely alluring feature in recent and future 802.11 releases that support very high data rates, such as 802.11ac.

2.5.1 Wireless Access Point Virtualization

In 802.11 WLAN, there is already the widely-deployed feature of virtual access point (VAP) (not to be confused with the same term used in CloudMAC in Section 2.4) as an extension to the basic service set (BSS) [33]. A physical access point can have multiple VAPs, each having its own BSS and service set identifiers (SSIDs). Each VAP also has its own beacon frame advertisements, security settings, forwarded authentication to RADIUS servers and attachment to VLANs. Of course, multiple VAPs must share the same radio parameters such as channel and transmission power. An example of virtualization architecture for 802.11 WLAN, CloudMAC, is presented in the previous Section 2.4. Otherwise, there is SplitAP [34], which attempts to control the uplink traffic by installing a plug-in on wireless client stations. While downlink traffic scheduling is relatively simple to perform at the AP, uplink traffic control is much harder to perform in 802.11. These client plug-ins communicate with a controller residing on the AP that runs a fairness scheduling algorithm. One weakness of this approach is that plug-ins must be intrusively installed inside wireless client devices.

2.5.2 Wireless Network Interface Card Virtualization

In terms of implementation, WLAN technologies are more versatile than cellular technologies. With a WiFi radio card, also called a wireless network interface card (NIC), a computer workstation can be configured to act as an 802.11 AP. Many modern commercial APs that run Linux-based firmware on an embedded general purpose processor (GPP) are also using wireless NICs as radio frontend. Thus, 802.11 virtualization can be applied to the 802.11 wireless NIC.

There are existing techniques that are applied on Ethernet NICs such as single-root input/output virtualization (SR-IOV) used in server virtualization [35]. These techniques use hardware-assisted components to facilitate virtualization. For instance, in SR-IOV, memory and address translation are performed by specialized hardware units instead of software [35]. In some instance, dedicated buffer chains and hardware 'lanes' are provided for each virtual interface [36]. Of course, similar mechanisms might be necessary but not sufficient for wireless NICs because they must also perform more sophisticated MAC and PHY functions.

In *Virtual WiFi* [37], 802.11 NICs are virtualized to be shared by virtual machines. A hybrid software and hardware approach is taken because advanced wireless virtualization functions cannot be fully implemented in hardware due to their complexity. As shown in Figure 2.6, the Virtual WiFi architecture is divided into four main components: the guest

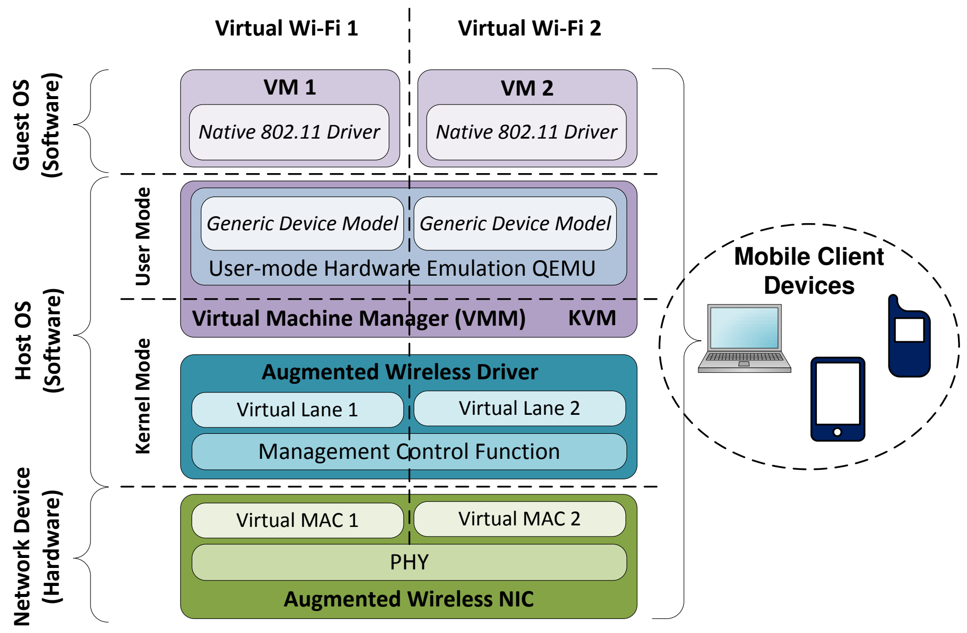

Fig. 2.6: Wireless NIC Virtualization with Virtual WiFi

machine wireless NIC driver, the virtual Wi-Fi device model, the virtualization-augmented device driver and the virtualization-augmented NIC. Similar to [36], each virtual instance has a dedicated and isolated vertical cross-layer software and hardware lane. Through a generic device model interface which only implements basic peripheral component interconnect (PCI) I/O functions, the guest driver communicates with the virtualization-augmented device driver, which implements more advanced NIC-specific functions. The augmented driver passes the management control messages obtained from the virtual drivers to their corresponding virtual MAC layer located on the augmented NIC. The microcode of the augmented NIC is modified to accommodate multiple virtual instances by keeping the state of each slice isolated.

On the other hand, in [38], the existing 802.11 power-saving mode (PSM) and point coordination function (PCF) are used to allow a single wireless NIC to simultaneously participate in multiple BSS by seamlessly switching between them. This technique takes advantage of the sleep state of PSM to maintain the virtual connectivity and avoid interruption due to re-association. There are also many other attempts at virtualizing the 802.11 WLAN. In the case of [39], no major modifications to existing technologies are made. The existing virtual access point is simply connected to a virtual switch which then forwards the traffic to a virtual Ethernet interface connected to a VM [39]. A similar strategy is used in the datapath virtualization implemented in Aurora in Subsection 5.1. In FLAVIAn [40], the existing SoftMAC Linux implementation *mac80211* is enhanced into a less monolithic framework, the *mac80211++*, which is more suited for virtualization.

2.6 Virtualization in Cellular Technologies

One important difference between WLAN technologies and cellular technologies is the presence or lack of elaborate multi-user scheduling and carrier-grade infrastructure support. In other words, basic cellular technologies already have more advanced resource scheduling capabilities compared to WLAN technologies, which are important assets for virtualization. Of course, in the case of 3G networks, there is already support for the concept of *network sharing* [10]. In network sharing, multiple mobile virtual network operators (MVNOs) can share the same basestation (eNodeB). However, the configuration of the basestation hardware and the wireless protocol cannot be controlled by the MVNOs [9]. Thus, deeper basestation virtualization have been considered for both IEEE 802.16 WiMAX and 3GPP LTE.

2.6.1 Virtualization in WiMAX

Despite being superseded by LTE, WiMAX technologies are comparatively more mature than LTE in terms of virtualization. The virtual base transceiver system (vBTS) [41] is a WiMAX virtualization architecture that runs full virtual instances of the basestation on virtual machines. A router-like flow slicing engine lies as an overlay to the physical basestation, inside the *modified* access service network (ASN) gateway. Traffic isolation is provided to each slice by using traffic shaping mechanisms [42] to limit the traffic coming in from different virtual basestation instances. Thus, in this architecture, the physical basestation is treated as a *black box* component. One major advantage of vBTS is the flexibility of supporting different MAC schedulers in different vBTS instances since they are independent from each other and from the physical basestation. However, the effectiveness of virtualization is limited due to a lack of coupling with the real scheduler inside the physical basestation. As such, the schedulers can only provide a coarse isolation between different slices [43].

Some of the limitations of vBTS issues are addressed by *integrating* the virtualization scheduler *within* the physical basestation in the network virtualization substrate (NVS) architecture [43]. The NVS architecture takes advantage of the existing OFDMA-based scheduling and QoS capabilities of WiMAX. In [43], two types of scheduling, *slice scheduling* and *flow scheduling*, represent two different scopes and can act separately from each other. Slice scheduling is the partitioning of resources across different tenants. The NVS framework supports two different slice scheduling policies: the *resource-based* allocation and the *bandwidth-based* allocation. The resource-based allocation consists of partitioning a specific proportion of OFDMA slots to a virtual network during each frame time. Thus, the available resource limits are always known, making the virtual basestation similar to a normal basestation but with reduced data capacity. The bandwidth-based allocation consists of guaranteeing a specific aggregated throughput for the network traffic. This second approach allows the scheduler to dynamically allocate resource slots to satisfy a given minimum throughput requirement.

Flow scheduling allows tenant MVNOs to have full control and customization of the scheduling of downlink and uplink flows *within* their slice. Three different modes of flow scheduling with various degrees of freedom are supported in [43]: scheduler selection, model specification and virtual time tagging. The scheduler selection mode is the equivalent of providing a library of pre-programmed schedulers from which the MVNOs can select from. These scheduling modules are fully autonomous and require little or no user input. The model specification mode allows the MVNOs more freedom at changing the configuration of the virtual scheduler through the model interface and parameters. Finally, the virtual time tagging mode allows the emulation of arbitrary flow schedulers defined by the MVNOs. A virtual system time is tagged on each incoming flow in order to allow the maximum degree of freedom in the scheduling decisions of the virtual schedulers.

2.6.2 Virtualization in LTE

The virtualization of the complete system architecture evolution (SAE) or LTE infrastructure is segmented into different components. For instance, the backbone transport network is virtualized through SDN and network virtualization technologies, as exemplified by the mobility anchoring with OpenFlow shown in Subsection 2.4.2. The evolved packet core (EPC) can be virtualized by running it on virtual servers inside the cloud. On the other

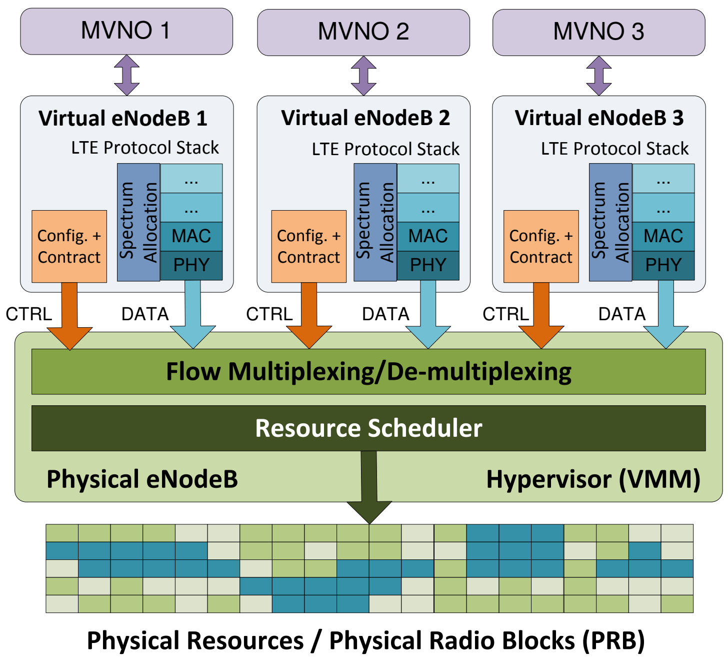

Fig. 2.7: LTE eNodeB Downlink Virtualization

hand, for the radio access nodes, called the Evolved Universal Terrestrial Radio Access Network (E-UTRAN) formed by eNodeB's, wireless basestation virtualization is applied.

LTE basestation (eNodeB) virtualization is explored in [5] through the integration of virtualization-aware algorithms in the scheduling of physical radio blocks (PRBs). Similar to NVS, [5] implements different types of resource allocation within the downlink air-time scheduler at the eNodeB, as shown in Figure 2.7. It attempts to define different types of SLA for entire slice owned by a tenant MVNO: fixed guarantee, dynamic guarantee with maximum bandwidth restriction, best effort with minimum guarantee and best effort with no guarantee. Such an architecture is extremely similar to the virtual radio node architecture from [9] discussed in Subsection 2.2.3.

2.7 Virtualization in Heterogeneous Technologies

Other than sharing resources, one interpretation of virtualization is to provide an *abstraction* layer to allow heterogeneous technologies to be managed through a common framework. The Carrier Grade Mesh Network (CARMEN) architecture is a heterogeneous wireless mesh network infrastructure with a MAC abstraction layer supporting both 802.11 and WiMAX nodes [44]. The CARMEN mesh network is composed of three different types of nodes: CARMEN mesh points (CMPs), CARMEN access points (CAPs) and CARMEN gateways (CGWs). The CMP nodes serve as wireless routing relays to other nodes and forwards packets coming from CAPs with flow-based QoS reinforcements. The CAP nodes are located at the edge of the mesh network and connect the end users to the network. The CGW nodes are the boundary points between the wired network infrastructure and the wireless mesh network. The MAC abstraction layer proposed in [44] simplifies the management and traffic offloading of the mesh nodes. It is divided into the mesh functions sub-layer and the MAC abstraction sub-layer. The mesh functions sub-layer consists of the technology-*independent* mesh node management functions such as mobility, forwarding, monitoring and configuration. The interface management function (IMF) maps the mesh functions from the technology-independent interface to the technology-dependent interface through an adapter module.

2.8 Wireless Virtualization and Software-Defined Radio

The goal of SDR technologies is to enable a more programmable and general-purpose radio hardware, as mentioned in Chapter 1. The usage of SDR can facilitate the implementation of virtualization architectures by decoupling of the control and processing logic from the hardware data plane. With such a decoupling, it is possible to have the same radio hardware simultaneously support different wireless standards. For instance, the GNURadio project [45] provides open source software modules to perform signal processing instead of relying on specialized hardware. Unfortunately, it suffered from poor real-time performance. However, more recent advances in SDR technologies such as Sora SDR [46] and OpenRadio [47] offer feasible real-time performance.

While SDR technologies based on programmable hardware using DSP and FPGA are very fast, they are often more expensive and do not provide the same amount of flexibility as a purely software-based SDR running in GPPs. On the other hand, it is very hard for GPPs to meet the throughput and latency requirements for practical real-time radio transmission. In order to address these issues, the Sora platform [46] provides a pure software-based solution with high performance. Sora is divided into three main sections: the radio front-end hardware, the radio control board (RCB) and the software radio stack.

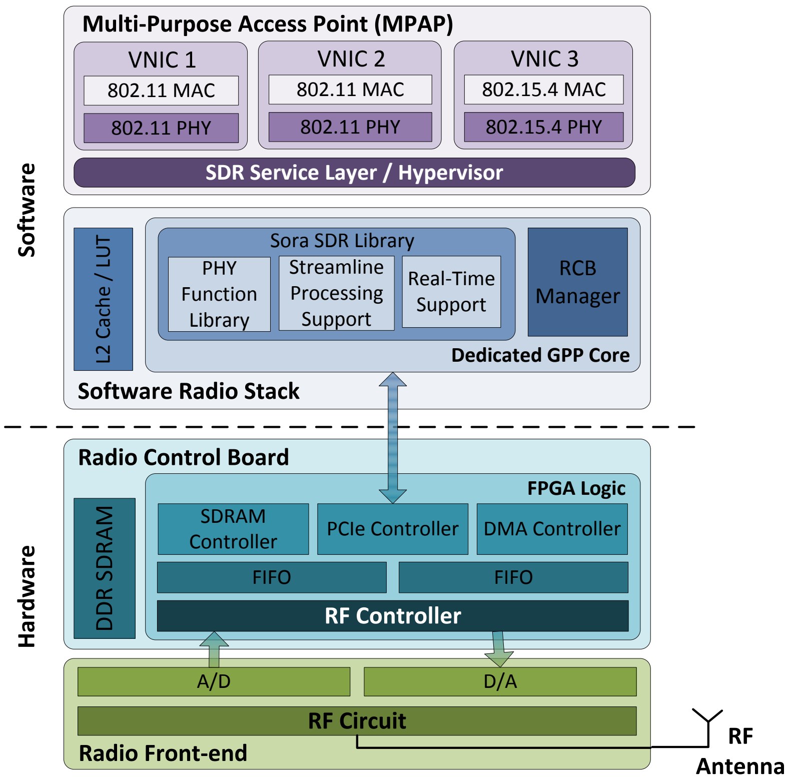

Fig. 2.8: Multi-point Access Point with Sora SDR

The radio front-end hardware consists of the antenna and the digital/analog converters. The digitized signal is directly fed into a host computer through the RCB. Thus, the entire PHY layer baseband processing is performed in software. The RCB is an interface board with a direct memory access (DMA) module and a custom PCI Express (PCIe) controller, which supports high bus transfer rate and low latency sufficient enough to satisfy timing requirements of many wireless standards. The RCB can be installed on a computer workstation like a regular NIC. Additional software-based optimizations are performed in the software radio stack, which resides in the operating system kernel as a driver-like component. It contains a library of PHY functions optimized for multi-core GPPs using streamline processing, look-up tables (LUTs) and single instruction multiple data (SIMD). The multi-purpose access point (MPAP) [48], an example of virtualization using SDR, is shown along with the Sora architecture in Figure 2.8. The MPAP architecture allows a ZigBee radio interface to simultaneously share the same hardware with an 802.11 radio interface.

On the other hand, OpenRadio [47] is a hybrid SDR platform partially based on hardware digital signal processing (DSP) technologies. Its objective is to decouple the wireless protocols from hardware and separate the processing functions from decision functions in wireless protocols. The decoupling of wireless protocols from the physical hardware is performed through protocol decomposition by identifying a set of basic irreducible PHY and MAC building blocks. PHY building blocks are implemented within DSP and FPGA-based modules. Since these functions are common to PHY layers of most wireless standards, a *common dataplane* can be constructed for most if not all wireless standards [47]. The constructed wireless protocol consists of two distinct planes: the decision plane and the processing plane. The decision plane is a logic assembly of branching rules. The processing plane is defined by mathematical operations over the signal data. The decision plane and the processing plane of OpenRadio are separated by the information plane, an interface formed by the configuration and statistics of the processing plane acting as interface between the two planes. This architecture has the advantage of providing *fixed guarantee* on the execution time of functions [47]. This is because the time-intensive data processing is performed in hardware (fixed execution time) while all the decision that are not computationally-intensive are performed in software (variable execution time).

2.9 Sub-PHY Wireless Virtualization

The virtualization of wireless resources can be performed even below the PHY layer of the wireless protocol stack. Such low-level techniques are usually very complex and difficult to implement due to their tight coupling with the physical medium. However, once implemented, they provide the highest degree of flexibility and enable potentially revolutionary applications. These techniques are mostly developed to support advanced radio applications such as cognitive radio and dynamic spectrum allocation. They can complement SDR technologies presented in the previous Section 2.8.

2.9.1 Spectrum Virtualization

One category of sub-PHY virtualization techniques is to apply a virtualization and abstraction layer over the frequency spectrum, as in the spectrum virtualization layer (SVL) [49]. SVL resides between the PHY layer and the RF circuit. As an intermediate layer, SVL does not require major modifications to the MAC and PHY layers of existing wireless standards. Instead, it allows the spectrum shared among *virtual radio* stacks to be managed in a *technology-independent* spectrum manager. These virtual radio instances can potentially run different wireless protocols. SVL is composed of the spectrum manager, the spectrum map, the spectrum reshaper and the software mixer/splitter.

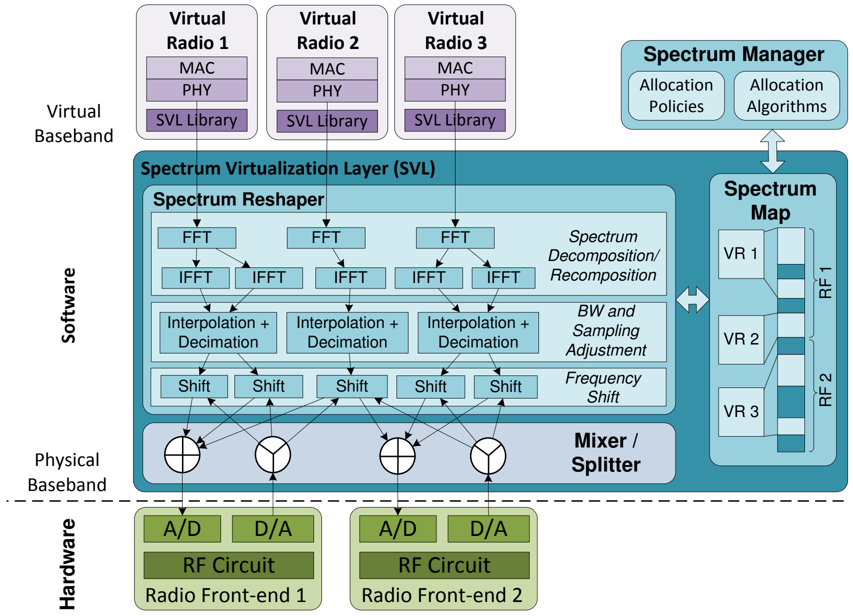

Fig. 2.9: Spectrum Virtualization Layer

The overall architecture is very similar to a 'spectrum router' [49]. The spectrum manager determines the spectrum allocation of different virtual radio stacks. It creates the spectrum map based on allocation algorithms and policies. The spectrum map is an efficient lookup mechanism with very little overhead. The spectrum reshaper performs virtualization by reshaping the baseband spectrum of each virtual radio stack to fit into the spectrum specified by the spectrum map. The reshaping consists of a mixture of signal decomposition, signal re-composition, bandwidth and sampling rate adjustment and frequency shifting [49]. Finally, the mixer and splitter multiplexes/de-multiplexes the baseband signals of each virtual radio before sending the combined spectrum to the radio frontend. The architecture is outlined in Figure 2.9.

Due to the abstraction layer provided by SVL, the spectrum can be mapped to not only a single frontend but to a network of *multiple* frontends. Thus, SVL also behaves as a RF frontend abstraction layer. As such, SVL can work in conjunction with concepts such as the fibre-connected massively-distributed antennas (FMDA) system [4], in which the baseband signals of a network of radio nodes are jointly processed. Cognitive radio techniques can also be applied over such virtualized platform. For example, a white-space networking application that uses the 802.11g protocol to transmit data in the TV channel frequencies is supported by SVL [49]. The SVL architecture is fully implemented in software by using the Sora SDR platform discussed in Section 2.8.

2.9.2 Radio Frontend Hardware and Virtualization

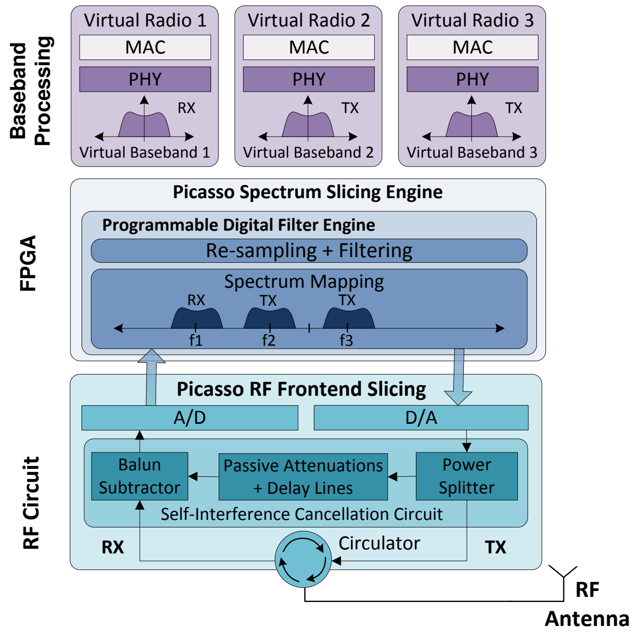

Fig. 2.10: Picasso RF Frontend Circuit for SDR and Virtualization

Physical limitations of the RF frontend in terms of isolation and bandwidth do exist. Thus, the radio frontend circuit itself can be enhanced to facilitate virtualization and various other advanced radio technologies by supporting a more flexible range of spectrum with fewer limitations and more degrees of freedom. For instance, in [50], Picasso provides a nearly full duplex RF circuit that can be used in SDR, cognitive radio and virtualization. It allows simultaneous transmission and reception of signals in the same and adjacent frequency spectrum with a single antenna. In addition, it provides spectrum slicing and virtualization through FPGA-based digital filters, allowing multiple virtual radio protocol stacks to apply dynamic spectrum allocation techniques to share the fragmented spectrum available in the same frequency band [50]. In order to achieve this, two components are used: the *passive self-interference cancellation system* and the *programmable spectrum slicing engine*. The passive self-interference cancellation system is based on a full-duplex circuit composed of a circulator and a passive cancellation circuit with fixed delay lines and passive programmable attenuators. The spectrum slicing engine is implemented using Xilinx Virtex-5 FPGAs and contains a digital filter engine to perform re-sampling, filtering and remapping of the digitized samples. The overall spectrum slicing architecture is comparable to that of SVL except that it is mainly FPGA-based. Additional digital self-interference cancellation is performed inside the slicing engine to provide further isolation of the spectrum. This architecture is shown in Figure 2.10.

2.10 Chapter Summary

In this chapter, different aspects of wireless virtualization have been surveyed in order to identify the existing architectures and implementations of virtualization in wireless technologies. The integration and interdependence of virtualization with other research areas such as Future Internet testbeds, software-defined network and software-defined radio have been explored. This literature survey serves as the basis for classifying different wireless virtualization approaches and formulating a generic wireless virtualization framework in the following Chapter 3. The various techniques and architectures presented in this chapter are taken into account in the design and implementation of Aurora in Chapter 4.

Chapter 3

Generic Framework for Wireless Virtualization[1]

As discussed in the previous Chapter 2, there is a great diversity in the scope, depth, application and feasibility of different existing approaches in wireless virtualization. In order to cover a large range of applications, often more than one of these approaches must be implemented. Currently, there is a lack of a *common framework* that allows these different virtualization perspectives to interact and coexist. Thus, this chapter outlines the architecture of an *evolvable*, *modular*, *generic* and *multi-perspective* virtualization *meta-*framework for wireless resources. Such an evolving framework aims at providing the tools and building blocks necessary to explore and develop various virtualization architectures. As such, it is not intended to be a definitive answer to wireless infrastructure virtualization. In this chapter, the term *virtual resource* refers to an abstracted *slice* of the physical resource.

3.1 Wireless Virtualization Perspectives

Unlike the formal server virtualization requirements defined in [51], there is no well-established guiding framework or formal requirements for the so-called wireless virtualization. Thus, it is important to define a classification framework that can focus the attention

[1]Parts of this chapter have been presented at the 2013 IEEE Third International Conference on Selected Topics in Mobile and Wireless Networking (MoWNet'2013) in Montreal, Canada [1] and published as part of the SpringBriefs in Computer Science series [2].

on different aspects of wireless virtualization. This section proposes a classification of various wireless virtualization approaches into three *perspectives*: *flow-based* virtualization, *protocol-based* virtualization and *spectrum and RF frontend* virtualization.

These three perspectives are mainly defined based on the type of the resources being virtualized and the *objective* of virtualization. The motivation of such a classification is that it is independent from specific wireless standards or technologies. In addition, it can provide some insight on why these perspectives are necessary in a given application. Overall, the different types of virtualization are like pieces of a larger puzzle in the sense that they can *complement* each other. However, these perspectives are not mutually exclusive. Instead, they are markers on a *gradual* scale of virtualization.

3.1.1 Flow-based Wireless Virtualization

In flow-based wireless virtualization, the focus is on the customization and control over the datapath of the wireless infrastructure, viewed as a data exchange and distribution network. Flow-based virtualization takes care of the isolation, scheduling, management and service differentiation between traffic flows. It can be considered as an extension of the flow-based SDN and network virtualization concepts into wireless technologies. Using the same SDN terminology, a 'flow' is an extremely flexible construct defined as a stream of data sharing a common signature, uplink or downlink. It can be managed from within a slice or across different slices of a virtualized network. Despite its resemblance with SDN and network virtualization, flow-based *wireless* virtualization approaches require wireless-specific functionalities such as the scheduling of radio resource blocks in order to reinforce quality-of-service (QoS) and SLA over the traffic flows.

The general architecture of this perspective can be divided into two types: *overlay* flow-based virtualization and *integrated* flow-based virtualization. In the *overlay* case, the wireless access hardware is considered as a black box. The virtualization layer and hypervisor are located outside of the access hardware. They function like a filter, switch and gateway to provide a configurable datapath and control path to the wireless nodes. The overlay virtualization is required in order to integrate existing wireless technologies into a virtualized infrastructure. However, since it cannot access the internal resources of the hardware, the level of guarantee and the granularity of control over the servicing of the flows is limited. Implementations such as OpenRoads [7] (Subsection 2.4.2) and vBTS [41]

(Section 2.6) are examples of overlay flow-based virtualization.

In the *integrated* case, the main difference is that the virtualization layer is located within the wireless access hardware, granting it access to the inner scheduling mechanisms of the radio node. In other words, the resource management on the hardware is modified to be virtualization-enabled and aware of the different SLA of different slice tenants. As such, integrated flow-based virtualization often deals with allocation of radio resource blocks in order to satisfy these service requirements. This allows a more dynamic and efficient way of managing different traffic flows. Architectures such as NVS [43] and virtual LTE [5] (Section 2.6) are examples of integrated flow-based virtualization. Both types of flow-based virtualization architecture on radio hardware are outlined in Figure 3.1 in comparison with a non-virtualized radio hardware.

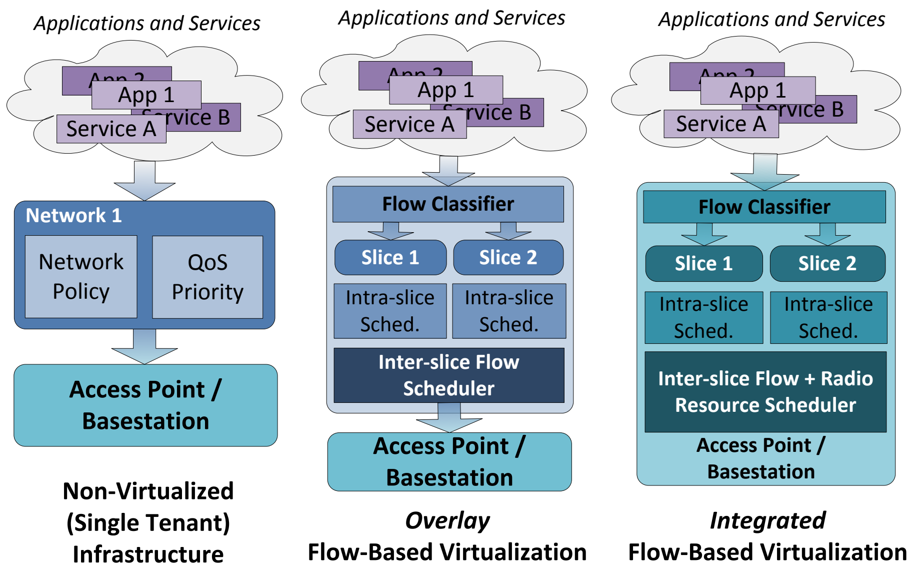
Applications and Services
App 2
App 1
Service B
Service A
Network 1
Network Policy
QoS Priority
Access Point / Basestation
Non-Virtualized (Single Tenant) Infrastructure
Applications and Services
Flow Classifier
Slice 1
Slice 2
Intra-slice Sched.
Intra-slice Sched.
Inter-slice Flow Scheduler
Access Point / Basestation
Overlay Flow-Based Virtualization
Applications and Services
Flow Classifier
Slice 1
Slice 2
Intra-slice Sched.
Intra-slice Sched.
Inter-slice Flow + Radio Resource Scheduler
Access Point / Basestation
Integrated Flow-Based Virtualization

Fig. 3.1: Overlay and Integrated Flow-Based Virtualization Architecture

Flow-based virtualization is the most feasible approach to implement with very immediate benefits. Its main benefit is to provide a more flexible and efficient traffic and resource management. It potentially allows tenants to control the scheduling of flows for clients within their own slice of the infrastructure. In addition, flow-based perspectives are required for the integration of wireless technologies with the rest of the cloud infrastructure.

3.1.2 Protocol-based Wireless Virtualization

Protocol-based wireless virtualization focuses on the wireless protocol stacks and their support on the virtualized radio nodes. This perspective considers the isolation, customization and management of multiple wireless protocol instances on the same radio hardware. It allows tenants to control separate instances of the wireless protocol stacks on the *same* radio hardware, which is not possible in flow-based virtualization. As such, the types of resources being virtualized are mostly MAC and PHY processing resources, which are distributed between software and hardware. Link layer and protocol configuration functionalities are usually located in software components such as device firmware, driver and micro-controller code. Baseband processing operations are typically performed in hardware embedded digital signal processing (DSP) units. With software-defined radio (SDR) technologies, more and more hardware functions are being processed in software, decoupling the wireless protocol from the hardware. The protocol-based perspective attempts to take advantage of these advancements to improve the flexibility and sharing of radio hardware.

In terms of architecture, there are varying levels of protocol-based virtualization depending on how decoupled the protocol is from the hardware. The *partial* protocol-based virtualization consists of tenants sharing the *same* wireless protocol stack but with different *configuration profiles*. In other words, each tenant can have a different set of radio MAC and PHY configuration parameters for their slice, which is not possible in pure flow-based virtualization. This partial implementation requires some software interfaces to dynamically access radio configuration parameters and the protocol stack to process data with changing MAC or PHY parameters depending on the slice it belongs to. The NIC virtualization techniques discussed in Subsection 2.5.2 belong to this category.

On the other hand, *full* protocol-based wireless virtualization allows completely *different* protocol standards to be simultaneously supported on the same radio hardware. This perspective allows the same radio hardware to be reused for different wireless protocols. In order to support this perspective, the radio hardware is fully abstracted and decoupled from the wireless protocols. MAC and PHY resources are allocated based on a *protocol scheduler* mechanism that can vary based on the architecture. The Sora SDR [46] and OpenRadio [47] (Section 2.8) fall into this level. Both levels of protocol-based virtualization along with spectrum virtualization are shown in Figure 3.2.

It is important to note that the challenging aspect of the full protocol-based architecture is the spectrum allocation for different protocols. In this thesis, the protocol-based perspective is defined to be uniquely focused on the mechanisms that support virtual protocol stacks. The management of the spectrum is considered in the following subsection.

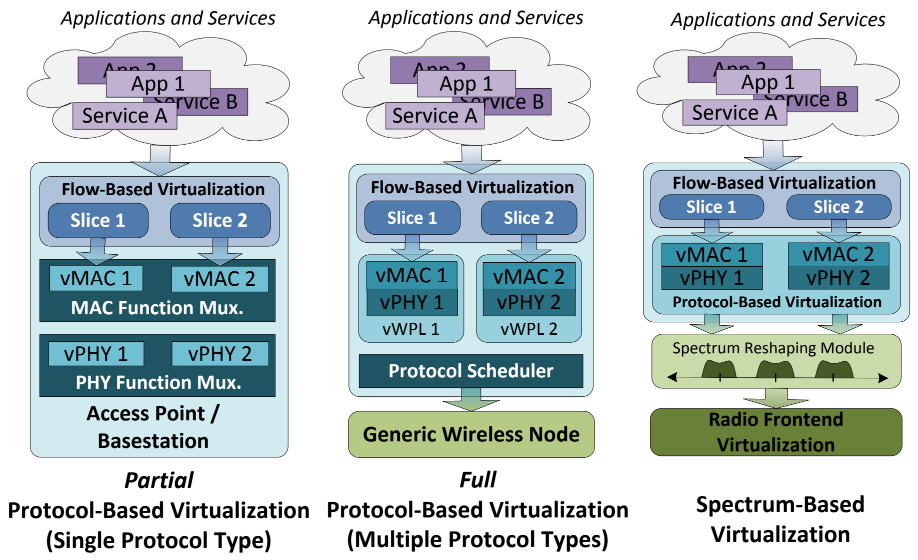
Applications and Services
App 2
App 1
Service B
Service A
Flow-Based Virtualization
Slice 1
Slice 2
vMAC 1
vMAC 2
MAC Function Mux.
vPHY 1
vPHY 2
PHY Function Mux.
Access Point / Basestation
Partial
Protocol-Based Virtualization
(Single Protocol Type)
Applications and Services
App 2
App 1
Service B
Service A
Flow-Based Virtualization
Slice 1
Slice 2
vMAC 1
vPHY 1
vWPL 1
vMAC 2
vPHY 2
vWPL 2
Protocol Scheduler
Generic Wireless Node
Full
Protocol-Based Virtualization
(Multiple Protocol Types)
Applications and Services
App 2
App 1
Service B
Service A
Flow-Based Virtualization
Slice 1
Slice 2
vMAC 1
vPHY 1
vMAC 2
vPHY 2
Protocol-Based Virtualization
Spectrum Reshaping Module
Radio Frontend Virtualization
Spectrum-Based Virtualization

Fig. 3.2: Partial and Full Protocol-Based and Spectrum Virtualization Architecture

3.1.3 Spectrum and RF Frontend Virtualization

The lowest level of virtualization, the spectrum and RF frontend wireless virtualization, focuses on the dynamic allocation and management of the spectrum and the radio frontend nodes. The types of resources being virtualized are the frequency spectrum and the RF frontend. In terms of spectrum virtualization, this perspective aims at providing an *abstraction* layer over the spectrum available at a given region and time in order to support a more *intelligent*, *flexible* and *efficient* spectrum usage. Here, the spectrum allocation is different from that of flow virtualization in the sense that it has a wider and more dynamic scope. As such, it covers the general spectrum potentially used by different standards. Cognitive radio technologies and dynamic spectrum allocation techniques are extensively used in this perspective. In RF frontend virtualization, this perspective considers different techniques that can remove physical limitations in the radio hardware in order to dynamically support a wider range of spectrum, which is an important feature for spectrum virtualization.

In terms of architecture, the spectrum and RF frontend virtualization requires some form of *spectrum sensing* mechanisms in order provide the *spectrum manager* information required to perform allocation and coordination. On the other end, the radio frontend must be able to receive and transmit at a wide range of frequencies with as few restrictions as possible. Ideally, the RF hardware should be capable of supporting wideband transmission with very good isolation between adjacent spectrum, provided either through digital filters or through RF circuit components. Both the spectrum layer and the RF frontends are completely decoupled from the wireless protocol stacks. Such an architecture is illustrated in Figure 3.2. From the previous Section 2.9, SVL [49] is an example of spectrum virtualization whereas Picasso [50] is an example of RF frontend virtualization. Otherwise, a centralized baseband processing unit, as in the case of fibre-connected massively-distributed antenna (FMDA) systems [4], can also help in the implementation of spectrum and RF frontend virtualization over a large region.

In protocol-based virtualization, in order for multiple protocols to coexist on the same hardware, a mechanism to share the spectrum must be applied. Otherwise, there might be conflicts between protocols sharing the same frequency band. Thus, spectrum and RF frontend virtualization is required in order for protocol-based virtualization to reach its full potential. These types of interplay between the different perspectives are discussed in the following subsection.

3.1.4 Coexistence of Different Virtualization Perspectives and Domains

In practical applications, the wireless access network is connected to the core switching network and the Internet. This highlights the importance of integration between different infrastructure *domains* in the Future Internet infrastructure. These *domains* refer to a set of infrastructure components sharing similar types of resources and performing similar functionalities. Cloud computing and SDN technologies cover the *computing* and *networking domains* whereas wireless virtualization is applied in the *wireless domain*. Cross-domain integration of control and customization allows for a more agile infrastructure [18]. In other words, the infrastructure can be viewed as an *active part* of the application or the service, not simply as a support. For instance, wireless virtualization can be integrated with other domains in order to form a cloud infrastructure in which a single slice can span across the entire infrastructure, binding together virtualized computing, networking and wireless resources. The different wireless virtualization perspectives presented in the previous subsections can play *different* roles in such a virtualized infrastructure. Example applications with 802.11 are briefly compared in Table 3.1. Thus, this thesis promotes the idea of *coexistence* between these perspectives. The idea of coexistence is supported by two main reasons: the important roles each perspective play and the gradual evolution of these perspectives.

Table 3.1: Examples of 802.11 Applications Over Different Virtualization Perspectives

Perspective	**Flow-based**	**Protocol-based**	**Spectrum-based**
Resource Type	Packet flow, flow-bound resource allocation	MAC and PHY processing resources	Wireless spectrum, radio frontend
Examples for 802.11	Flow isolation and management among virtual networks, enhanced virtual access point (VAP) functionalities [33], SDN application in wireless networks such as OpenFlow Wireless [7]	802.11 access points with configurable virtual radio profiles, coexistence with other wireless protocols on same radio hardware [47]	802.11 delivered over TV spectrum [48]

In terms of roles, the three perspectives are *complementary* to each other. For instance, flow-based wireless virtualization is required to connect virtual resources together. In [52], flow-based virtualization is used for network-wide applications such as the personalized mobility management. In this case, virtualization is used to maintain different *personalized connection profiles*. However, flow-based virtualization is often not sufficiently granular for more *advanced* and *efficient* wireless communication applications, such as the coordinated multipoint (CoMP) using hardware virtualization proposed in [53]. For the methods proposed in [53], virtualization is applied in the PHY and spectrum layer to provide virtual basestations for baseband processing that can be distributed over multiple physical basestations. On the other hand, even with spectrum virtualization, there is still need for protocol-based virtualization mechanisms to define and maintain the decoupling of the virtualized protocol stacks from each other and from the radio hardware. In all cases, flow-based virtualization is still required to integrate the data and control path with the rest of the infrastructure.

The second motivation behind a *multi-perspective* approach is the evolving nature of virtualization technologies. Not all the perspectives can reach their full potential with the existing technologies and techniques. This is particularly true for protocol-based virtualization, which requires a mature and sustainable SDR platform, and spectrum virtualization, which requires new radio hardware architectures. Looking back at the history of server virtualization, only software-based virtualization was originally possible. Then after the introduction of VT-x and AMD-V, full hardware-based virtualization was possible. Thus, the availability of technology and the transition from one technology to another are key aspects that must be taken into account. As wireless virtualization is still evolving, different perspectives should be simultaneously supported with the same infrastructure in order to leave room for improvement.

3.2 Towards a Generic Wireless Virtualization Framework

As discussed in the previous section, the different wireless virtualization perspectives can be complimentary to each other. The process of integration of virtualization technologies in a virtualized infrastructure should be *evolutionary* and *progressive*. This section discusses the central idea of this thesis, which is the formulation of a generic platform that can support these perspectives under a common framework. In other words, the so-called generic framework is a *meta*-framework that provides a sandbox approach that is modular and flexible enough to *support* various existing wireless virtualization architectures. Thus, the framework promotes a heterogeneous and diversified infrastructure as opposed to a monolithic one. At the same time, complete virtualization is a challenging task due to the complexity of such system. The advantage of a multi-perspectives approach is the integration of different perspectives and technologies from SDN, SDR and cognitive radio as *interworking* yet *separate* solutions to the multiple aspects and challenges of virtualization. In a sense, the generic framework is similar to an *infrastructure operating system*, in which different virtualization perspectives are represented by different software modules. Ideally, such a framework allows researchers and service providers alike to explore, develop, implement and deploy new applications and services on the virtualized infrastructure and is beneficial to the evolution of the future infrastructure. The following subsection identifies the different challenges and requirements for the design of such a generic wireless virtualization framework.

3.2.1 Challenges of a Wireless Virtualization Framework

Even though different perspectives are complementary to each other and can be combined, not all the tenants might be interested in having full control over them. For instance, some tenants do not need protocol-based or spectrum virtualization perspectives because of the nature of their applications or services. Moreover, not all existing hardware equipments can support the same degree of virtualization, as mentioned in Subsection 3.1.4. A sustainable virtualization framework should progressively allow multiple virtualization perspectives to coexist and evolve. Some of the current and emerging challenges of wireless virtualization, along with possible solutions and impact on the proposed framework, are identified as follows.

1. **Flexibility-performance trade-off**: Wireless technologies are inherently specialized and optimized by design. High granularity of control and programmability of wireless hardware can add overhead fatal to time-sensitive functionalities. *Over-generalization* can lead to a reduced efficiency in the performance of a given wireless technology. On the other end, a deeply integrated and specialized virtualization layer can lead to a more efficient design, albeit a less flexible and portable one. It all comes down to the question of determining the right amount of wireless virtualization. Unfortunately, since different applications and services have different requirements, there is no right answer to that question. Typically, existing wireless virtualization techniques are implemented with as few intermediate layers as possible in the *direct datapath* and *time-critical decision path*. In some cases, a *localized* virtualization agent is closely integrated to a specific technology, allowing it to retain performance while providing a common abstraction layer to its functionalities. The proposed framework incorporates all these different approaches as building blocks of virtualization. As such, it operates as an *orchestration plane* over other virtualization technologies, giving the tenants flexibility to build their own virtual infrastructure without sacrificing the performance of the existing technologies. This concept is explained in Subsection 3.3.2.

2. **Scalability of framework**: There are various scalability issues at the different levels of a virtualization framework, notably at the infrastructure level and the radio node level. In terms of infrastructure-wide *control and management*, a centralized virtualization manager can become overburdened as the number of resources increases. In

existing virtualization C&M frameworks, the manager plays the role of a broker, *delegating* as much as possible the direct management of resources to tenant-operated guest controller, as suggested in [9]. Thus, instead of letting the manager handle all the tenants directly, semi-persistent rules and policies are installed on the local virtualization agents. In addition, resources can be clustered into *regions* each with a separate infrastructure manager, as implemented in OpenStack [16] and SAVI [14]. The proposed framework adopts both of these methods. On the other hand, the problem of scalability can also occur in software-defined technologies used to support virtualization. For instance, when baseband processing is pushed to software for virtualization, the scalability of the virtualization layer becomes dependent on the processing capabilities of the external controller and the link capacity between the controller and the radio hardware. Then, time-sensitive functionalities can experience longer delays as distance is increased. Potential methods to tackle these challenges include the use of high-throughput low-delay optical fibre connections and a more efficient partitioning of functionalities between hardware and software. Preferably, time-critical functionalities must be optimized in software and accelerated through hardware. The proposed framework provides a platform to test and deploy such methods.

3. **Complexity of architecture and interface**: If not carefully planned, a meta-framework not only adds complexity to the implementation but also exposes too much complexity to its tenants. By design, the virtualization layer (or hypervisor) must perform overhead virtualization-related functionalities of multiplexing, slice isolation, function translation and policy reinforcement. Thus, coordination among multiple slices can cause complex interactions among different components. In the proposed framework, in order to avoid increasing the complexity of the overall framework, many of these complex operations are *abstracted* and *modularized*. A *selection* of common high-level slice management functions is offered through the framework. To an external user tenant, the complexity of the implementation is hidden under various levels of APIs. A customizable level of granularity of control and configuration are exposed to different tenants with different needs, given to rise to a *scalable complexity*. In other words, the framework attempts to be user-friendly without removing access to the complexity actually required by certain tenants.

4. **Feasibility of deployment**: One of the most difficult challenges is the deployment of such a generic virtualization framework. The practical deployment of the framework requires technology-specific implementation and integration. However, unlike with server virtualization which is mainly dominated by a single technology (x86 architecture), wireless technologies are extremely diverse. Clearly, it is not possible to virtualize everything at once. Therefore, the multi-perspective framework approach can provide a modular platform on which different virtualization technologies can be slowly integrated. Moreover, *backward compatibility* and *retrofitting* are essential to the deployment of the framework as it is unrealistic to throw away the existing infrastructure. This is where a multi-perspective platform can shine since different levels of virtualization can be *abstracted* to offer different *capabilities* and *features* of the wireless resource node. Nodes without support for virtualization will simply be allocated 'as is' with a more limited set of functionalities. More advanced and high-performance capabilities can be then offered as a premium over the basic features. For this thesis, the framework is deployed as part of the SAVI testbed. The integration of existing university access points into the framework is considered in Subsection 5.5.

3.2.2 Defining Requirements of a Wireless Virtualization Framework

Different solutions to the challenges presented in the previous subsection were discussed. Here, some of these solutions are reformulated as requirements and features of a generic multi-perspective virtualization framework for wireless technologies. The partial fulfilment of these requirements by existing frameworks and technologies is also discussed.

1. **Generic, modular and open framework**: Different wireless resources use different virtualization perspectives and technologies. In order to support such a heterogeneous infrastructure, there are many characteristics that need to be taken into account in the design of the framework.

   (a) *Generic core engine*: The core framework should be *independent* from particular perspectives or technologies. General functions such as wireless network and slice creation, deletion and modification should be agnostic to how a slice is defined and implemented. The core framework act as an engine to orchestrate and broker wireless resources from an abstracted point of view, allowing the framework to be easily reconfigurable and extensible. This characteristic is observed in meta-architectures such as GENI [13], SAVI [14] and mobile virtualization [18]. The proposed framework differs from them by providing extending the core framework to the radio resource node level in order to manage wireless virtualization functions.

(b) *Modular components*: Modularity is an important characteristic that must be applied to both *internal* (between modules) and *external* (exposed to tenants) interfaces and functions. Technology-specific functions must be self-contained in an exchangeable software module. A similar concept with MAC functions was applied in [44] but should be extended to include more functionalities. The proposed framework in this thesis attempts to satisfy this requirement by encapsulating different implementation technologies for different wireless virtualization perspectives to offer them as *building blocks* part of a virtualization toolbox.

(c) *Open configurability and interfaces*: All the functionalities offered through the virtualization framework and the different virtualization perspectives should be fully accessible to tenants if they desire so. In other words, different tiers of perspectives can be exposed to the tenants depending on their needs. The main advantage of the proposed framework is that it allows the tenant to setup and configure their own data and control path and implement their own virtualization and software-defined architecture. At the same time, a tenant not interested in such low-level control can simply use the prebuilt *virtualization packages* and *blueprints*.

2. **Evolvability and extensibility of the framework**: Since it is not required to have all three wireless virtualization perspectives integrated at the same time, a progressive approach adjusted to different applications is suggested. As the depth of virtualization evolves, new extension modules and plug-ins can be added to the framework. The potential integration and federation of both existing and emerging virtualization architectures should be considered. Existing Future Internet testbed architectures such as GENI and SAVI are designed with such objectives in mind. The proposed framework follows a similar design pattern but focuses uniquely on wireless virtualization technologies, an area relatively less explored by existing works. In order to facilitate the development of the framework and leave room for contribution by other

research groups, the proposed framework will be based on *open source* technologies and offers a transparent access to internal APIs.

3. **Resource and function abstraction**: In relation to the modularity requirement, the resources should be abstracted as *building blocks*, giving a sandbox-like view of the infrastructure. The abstraction applied in the proposed framework should be flexible enough such that it can even be nested. As an example, one tenant should be able to obtain an abstracted SDR resource to create multiple virtual radio access nodes that can then be abstracted as resources to be used by other tenants. Thus, a portion of the virtualization platform itself can be allocated as a resource to a tenant interested at providing new virtualization services to other tenants, forming an interesting ecosystem of service exchange between tenants on the same infrastructure. Such an ecosystem was briefly explored in Section 1.4.

3.3 Architecture for a Multi-Perspective Wireless Virtualization Framework

This section outlines the overall architecture and concept of a generic wireless virtualization framework that can support multiple coexisting wireless virtualization perspectives and technologies. The framework is divided into two groups of components: the control and management layer (CML) and the virtualization layer (VL). The CML contains a series of components that process wireless infrastructure-wide management functions. It encompasses both the framework manager and the tenant-owned managers. On the other hand, the VL includes a set of virtualization mechanisms and agents that can be configured by the tenants through the CML. These two layers are kept separate in order to support as much as possible the decoupling of functionalities from the physical resources. Here, a physical *resource node* is defined as a radio node or a collection of radio nodes in the case processing is centralized. The rest of this thesis is mostly focused on the design and implementation of components from the virtualization layer. *The detailed architecture of the CML is outside the scope of this thesis.*

The overall architecture of the framework is shown in Figure 3.3, which describes how a tenant can obtain and control virtual wireless resources inside the framework. First, the tenant must send a configuration blueprint to the *infrastructure manager* to request

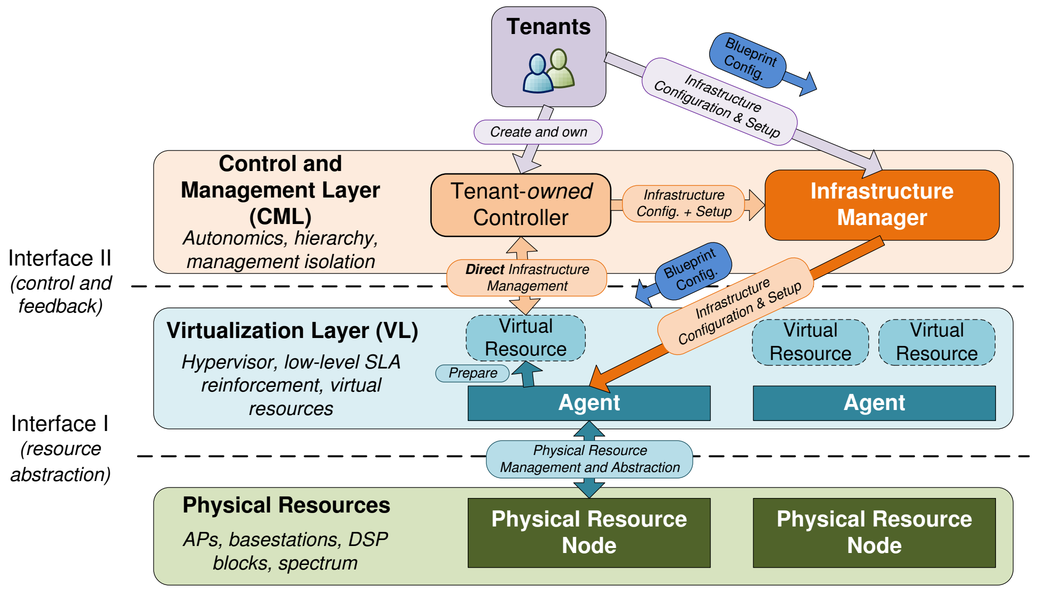
Tenants
Blueprint Config.
Infrastructure Configuration & Setup
Create and own
Control and Management Layer (CML)
Autonomics, hierarchy, management isolation
Tenant-owned Controller
Infrastructure Config. + Setup
Infrastructure Manager
Interface II (control and feedback)
Direct Infrastructure Management
Blueprint Config.
Infrastructure Configuration & Setup
Virtualization Layer (VL)
Hypervisor, low-level SLA reinforcement, virtual resources
Virtual Resource
Prepare
Virtual Resource
Virtual Resource
Agent
Agent
Interface I (resource abstraction)
Physical Resource Management and Abstraction
Physical Resources
APs, basestations, DSP blocks, spectrum
Physical Resource Node
Physical Resource Node

Fig. 3.3: General Outline of Main Layers of Proposed Framework

a specific wireless resource. The manager validates the request and sends a slice creation command to the virtualization *agent* located on the physical resource. The agent prepares a virtual slice and set it up to be connected to the tenant-specified virtual network(s). A direct control and data path between the slice and the tenant is therefore established. Tenant can then use its own controller to manage the virtual resources it acquired. This architecture is inspired by that of [15] and [18]. However, what is newly introduced is the integration of virtualization with *software-defined and service-oriented infrastructure* concepts, which allow the virtual infrastructure to be easily *reconfigured* as a *virtual testbed platform*. In order to support this in a scalable manner, the level of interaction and control offered to the tenants is separated into two phases: the virtual infrastructure *setup and configuration* and the virtual infrastructure *control and management*. Different control paths are used depending on the type of control operations.

1. **Virtual infrastructure setup and configuration**: Infrastructure setup designates any operations that require a change in the configuration of the *physical resource* that cannot be handled within the *virtual resource*. Such operations include modifications to the slice SLA, such as the creation, deletion and resizing of slices. Depending on the capabilities of the virtual resource, it can also include various changes to

the architecture of the virtual infrastructure and the relationships between virtual resources. Overall, the setup operations allow a tenant to fully *redefine* its own virtual infrastructure through a customized blueprint. The setup path is *indirect* since it must be validated through the infrastructure manager, which dispatches the blueprint to agents which reconfigure the resources on behalf of the tenants. This intervention is necessary because these setup operations can potentially affect *other* tenants residing on the same infrastructure. The manager must resolve any potential conflicts between them before letting the agents work on them.

2. **Virtual infrastructure control and management**: Infrastructure control and management refers to operations that fall *within* the jurisdiction of a slice and its tenant. In other words, it includes any operation that does not affect the *definition* and *capabilities* of the virtual resource itself. For example, it can include the control of decoupled functionalities using software-defined technologies. The control path for infrastructure management is *directly* established between the tenant and the virtual resources in order to reduce potential delays. Of course, some control operations can still impact slices from other tenants. These potential conflicts can be resolved by the local agent (as opposed to infrastructure manager). Ultimately, what functionalities fall into which type of control depends on the capabilities of the physical resource. For instance, a physical radio resource node that supports spectrum virtualization can treat channel selection as a control operation whereas a node that does not support dynamic channel configuration treats the same function as a setup operation. The different types of agents used in a multi-perspectives framework are discussed in the following subsection.

The interaction between the tenant and the different framework components is illustrated in Figure 3.4. Most existing architectures support the infrastructure control and management by tenants but only have a limited support for infrastructure setup and configuration. In these architectures, the infrastructure setup is relatively fixed and predetermined, making it difficult for tenants to apply different virtualization architectures and perspectives. The proposed framework offers a more diverse toolbox-like support for *both* infrastructure setup and infrastructure control, as will be shown in Chapter 4.

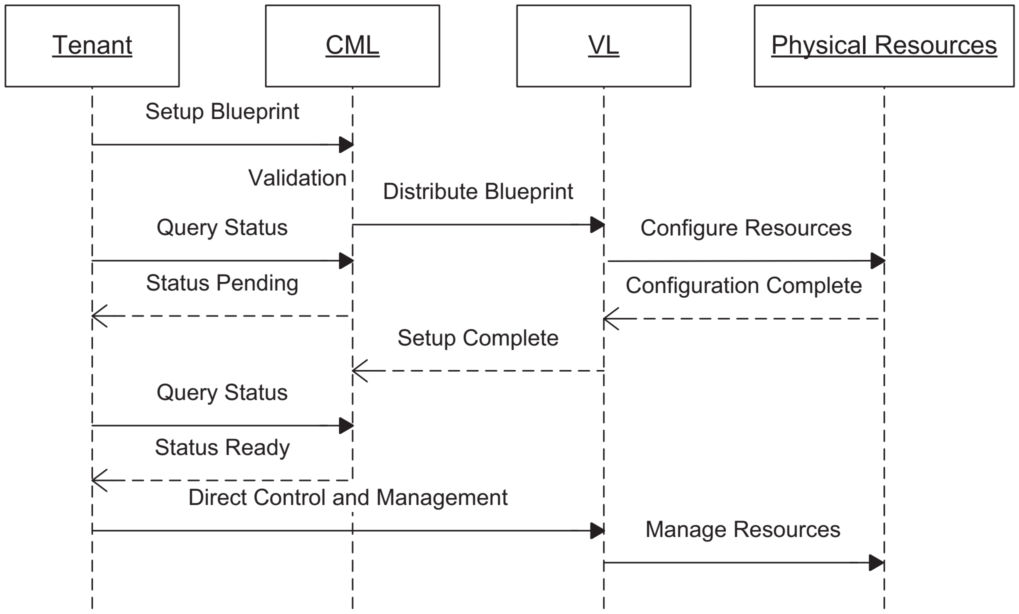

Fig. 3.4: Interaction Between Tenants and Main Layers of Proposed Framework

3.3.1 Multi-Perspective Integration in Virtualization Framework

From an infrastructure-wide point-of-view, a virtual infrastructure is created by assembling different types of resources and defining the connectivity between them. The wireless virtualization perspectives presented in Section 3.1 offer different types of virtual resources with different services. In fact, flow-based wireless virtualization is the only perspective that needs to be integrated along with network virtualization and SDN technologies as a minimum requirement. On the other hand, protocol-based and spectrum virtualization can offer functionalities completely foreign to the flow-based perspective. In some cases, the physical resources used in a given virtualization perspective can actually be *substituted* by a virtual resource obtained through another perspective. Moreover, resource allocation and partitioning should be applied whenever virtualization is not available. The integration of different perspectives within the proposed virtualization framework is illustrated in Figure 3.5 with a few example cases of resources with varying degrees of virtualization.

First, a non-virtualizable wireless resource (case A) can be integrated within the framework by the addition of an abstraction layer and agent to interface with the framework. The agent can establish the connectivity between the resource and the tenant network. Physical resources can then be directly allocated along with virtual resources in order to

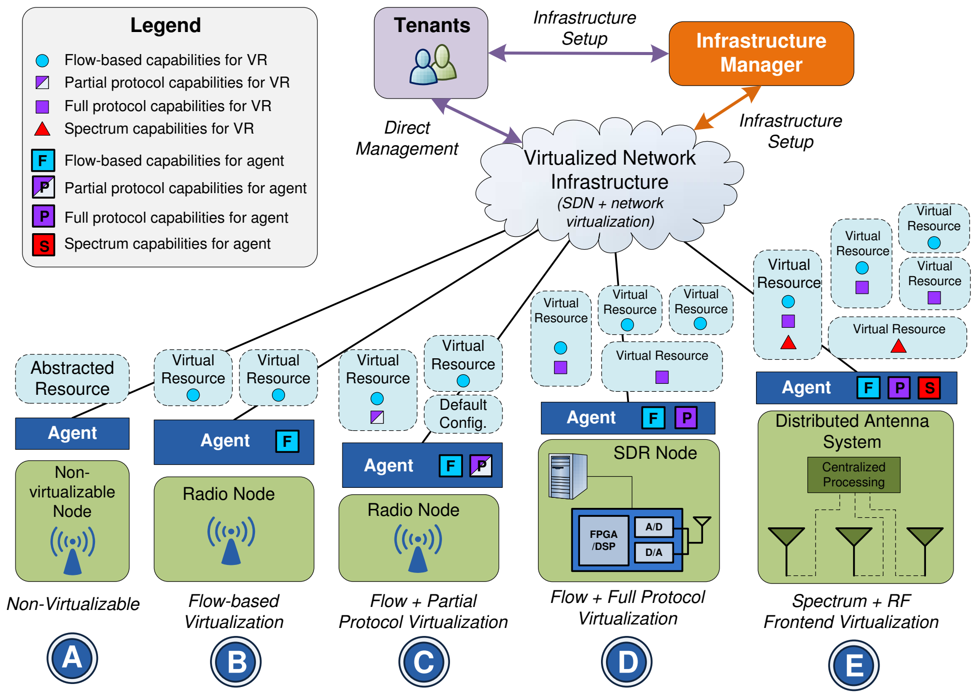
Legend
Flow-based capabilities for VR
Partial protocol capabilities for VR
Full protocol capabilities for VR
Spectrum capabilities for VR
Flow-based capabilities for agent
Partial protocol capabilities for agent
Full protocol capabilities for agent
Spectrum capabilities for agent
Tenants
Infrastructure Setup
Infrastructure Manager
Direct Management
Infrastructure Setup
Virtualized Network Infrastructure
(SDN + network virtualization)
Abstracted Resource
Virtual Resource
Default Config.
Agent
F
P
S
Non-virtualizable Node
Radio Node
SDR Node
FPGA /DSP
A/D
D/A
Distributed Antenna System
Centralized Processing
Non-Virtualizable
Flow-based Virtualization
Flow + Partial Protocol Virtualization
Flow + Full Protocol Virtualization
Spectrum + RF Frontend Virtualization
A
B
C
D
E

Fig. 3.5: Different Perspectives on Virtual Resources Inside the Virtualization Framework

benefit from the service-oriented and sandbox nature of the framework. In a resource node that only supports the flow-based virtualization perspective (case B), tenants must share the same wireless protocol and the same radio parameters. The capabilities that can be offered to tenants include datapath isolation and customization. The virtualization agent on such resources must setup virtual interfaces, virtual bridges, tunneling endpoints, traffic shaping mechanisms, data capturing services as well as SDN-enabled technologies. This type of agent is implemented in Chapter 4.

If partial protocol-based virtualization is available (case C), the resource node still retains the flow-based capabilities but acquire *additional* features such as the ability to allocate unique radio configuration profiles to each tenant. One important concept that is maintained throughout the proposed framework is that resource nodes with more capabilities can still offer virtual resources with lower capabilities. For example, in case C, *default* radio configuration packages are provided to tenants who only need flow-based perspectives. Even though it is preferable not to “waste” resource nodes with more capabilities on tenants who don’t need these capabilities, it is important to support this type of backward compatibility in order to achieve tenant-owned virtualization platforms *within* the framework. This is more apparent when more advanced capabilities are provided, such as in the case of protocol-based and spectrum-based perspectives. In a SDR resource node with support for full protocol-based virtualization (case D), tenants are able to implement and customize the protocol of their choice on their virtual resources. However, these tenants can also offer virtual resources with lower capabilities. An example of this is a tenant who acquired SDR resources through the framework and implemented a modified version of the 802.11 access point. Such a tenant can then offer VAP instances with only flow-based virtualization that other tenants can control. In the advanced case where spectrum and RF frontend virtualization are implemented on a distributed antenna system with central processing (case E), a network of radio nodes can be managed by a single agent which can offer multiple types of resources to different tenants. These resources can range from portions of the spectrum to a virtual radio node running a specific protocol.

3.3.2 Local Virtualization Agent Architecture

In order to maintain the coexistence of the different perspectives as presented in the previous subsection, the virtualization layer must be able to provide a common and modular framework while offering different functionalities to the tenants. In the proposed framework, as illustrated in Figure 3.3, the *local virtualization agent* is an intermediary layer that directly interacts with the physical resource nodes. The design and implementation of a local virtualization agent is one of the main focus of this thesis. Its architecture is key at satisfying the framework requirements identified in Subsection 3.2.2.

The agent is customized and optimized to the different capabilities of the resource node. To maintain modularity, the agent is composed of components separated into the technology-independent core engine and the technology-dependent abstraction layer. As shown on the Figure 3.6, a physical resource node is separated into two planes: the *agent plane* and the *virtualization plane*. The agent plane contains the agent, which handles the setup and configuration of the virtualization plane. The agent can also be located on an external controller depending on the technology available on the resource node. The virtualization plane contains the technologies that sustain the virtualization of the node, called *resource components*. By allowing tenants to directly interact with the virtualization plane once the setup is complete, the overhead of the proposed framework over existing virtualization technologies is mostly averted. In other words, flexibility is achieved in the agent plane without affecting the performance of the virtualization plane.

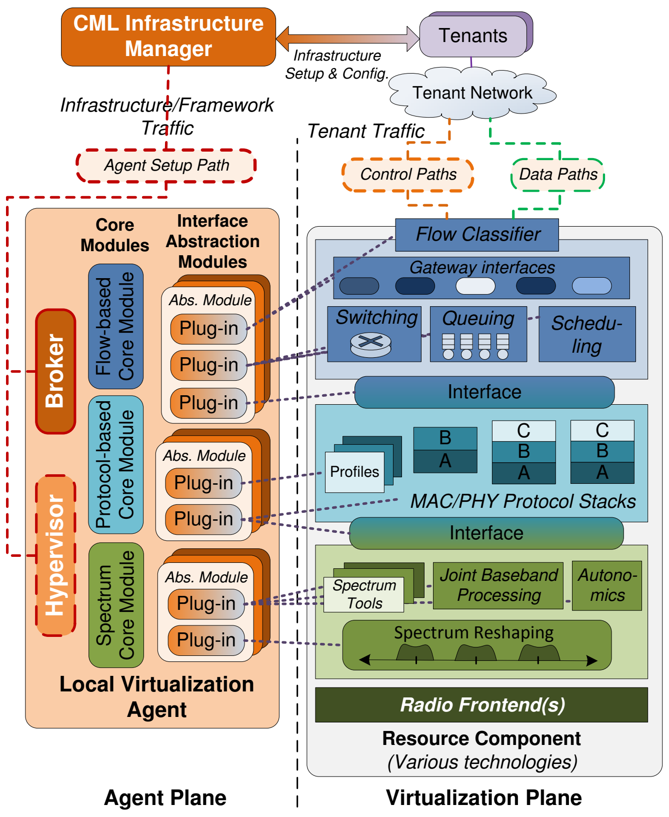

Fig. 3.6: Outline of Agent Architecture Inside a Generic Representation of a Resource Node

The main components of the agent are the *local broker and hypervisor*, the *interface abstraction modules* and the *core modules*. The implementation of the agent is discussed in Section 4.2.

1. **Local broker and hypervisor**: The core function of the local virtualization agent is to provide *localized* setup, configuration and management of virtualization. Whereas the infrastructure manager from the CML takes care of the infrastructure-wide virtualization as a *network* of wireless nodes, the local agent is only aware of resources available on the node it belongs to. There are two types of local virtualization managed through the agent: *broker mode* and *hypervisor mode*.

(a) *Broker mode*: The local *broker* sets up and configures other virtualization technologies to create a slice that can then be directly managed by the tenant. The broker can only initialize components configurable by the tenants. In other words, it is the *slice builder*. In addition, the broker performs some basic conflict resolution during the slice creation and configuration phase. Only the broker mode is implemented in this thesis in Chapter 4.

(b) *Hypervisor mode*: On the other hand, the local *hypervisor* performs resource allocation and conflict resolution of different slices on the node during the *active operation* of the slice. Clearly, conflicts during setup must be caught by the broker in order to avoid them during operation, which are more costly to resolve. However, since not all conflicts can be determined at initial setup, the hypervisor must be able to resolve conflicts dynamically at the expense of added overhead, regardless of whether conflicts actually occur. In most cases, the hypervisor function is actually delegated to the separate abstraction modules implemented by various technologies in the virtualization plane. Thus, while the broker *builds* the slice, the hypervisor provides *inter-slice isolation and management*. Unlike the broker, components created by the hypervisor are not visible nor directly configurable by the tenants. Inter-slice traffic shaping can be considered a hypervisor mode function and is discussed in Subsection 5.4.1.

2. **Core modules**: In order to satisfy the modularity and extensibility requirements mentioned in Subsection 3.2.2, the virtualization core modules are components that help define one or multiple wireless virtualization perspectives on a given resource node. The core modules are independent from a given technology implementation. As such, they are movable and reusable across different resource nodes and implementations. The three virtualization perspectives discussed in Section 3.1 can share some common modules. These core modules maintain an *abstracted* view of the different resources components. These technologies (resource components) that the core modules interact with through the abstraction modules are running concurrently and independently in the virtualization plane. As such, the core modules are used to *orchestrate* these resource components as building blocks of a slice.

(a) *Flow-based virtualization modules*: For flow-based virtualization, the core modules handle the creation of virtual control and data *paths* for each slice on the resource node, which involves the maintenance of virtual network interfaces and virtual bridging services as well as the attachment of different *metering* and *scheduling* services to those paths. The flow core modules configures these paths to connect to the tenant virtual network in a *tenant-defined* manner, integrating them as part of an external network virtualization and SDN framework. Finally, the flow modules provide gateway functions to transport all types of data, from IP packets, raw MAC frames to radio signal samples. The abstraction modules coordinated by this core module are the main focus of Subsection 4.1.4. This core module is explained in Subsection 4.2.4.

(b) *Protocol-based virtualization modules*: In the case of partial virtualization, the protocol-based virtualization core modules handles the management of protocol and radio configuration profiles. In the case of full virtualization, the modules assemble the different software components and interfaces to hardware components that constitute a virtual protocol stack. They also provide the virtual *radio interfaces* that are then used by the flow-based virtualization modules. Only a partial implementation of these modules is explored in this thesis in Subsection 4.1.4.

(c) *Spectrum and RF frontend virtualization modules*: At the spectrum and RF frontend virtualization level, flexible portions of various frequency bands can be dynamically allocated to each slice using cognitive radio and DSA techniques. These core modules can hypothetically interface with spectrum reshaping modules, maintain spectrum management profiles and collect spectrum sensing data. These modules are not implemented in this thesis.

3. **Interface abstraction modules**: The abstraction modules are the binding points between various technologies and the agent core framework. They contain technology-specific *implementations* of the functions called by the core modules. Their actual operation and implementation will greatly vary depending on the technologies available on the resource node. For instance, in software, they can act as inter-process communication plug-ins and system calls to interact with a separate virtualization technology, referred as *resource components*. The resource components they control are the basic building blocks and tools of the framework. They are implemented as *abstraction modules* and *plug-ins* orchestrated by the core modules.

3.3.3 Comparison to Other Virtualization Frameworks

The proposed multi-perspective virtualization framework is designed based on some of the testbed and meta-frameworks presented in Section 2.2. This subsection highlights the main similarities and differences between them. Overall, the general architecture of the proposed framework can be viewed as a both a fusion between the infrastructure virtualization architecture presented in [18] and the Virtual Radio architecture [9], and an extension to these frameworks.

The concepts of the virtual network controller, virtual resource manager and virtualized physical resources from [18] can be respectively mapped to the tenant-owned controller, the infrastructure manager and the virtualization agents. However, in [18], only a general overview of the concept is provided. Wireless resources are treated like other types of resources (network, computing, etc.) without specifying the *architecture* of the standardized interfaces for these wireless resources. The question on what functionalities a virtualized wireless resource should provide is also left unanswered. In other words, while the general infrastructure-level description of virtualization is provided, the architecture of wireless virtualization was not. As such, it does not explore the *full extent* and *potential* of wireless virtualization.

This issue has been partially addressed in Virtual Radio [9] where the architecture of a virtualization-enabled wireless resource node is provided. In the proposed multi-perspective framework, the concept of the infrastructure manager that allocates and sets up the radio nodes before letting them connect *directly* to the tenant is borrowed from [9]. Virtual Radio also provides an outline of virtual protocol stacks and how it interacts with the radio spectrum resource allocation. However, the Virtual Radio architecture requires some major changes in the existing wireless technologies. It neither considers the gradual integration of other wireless virtualization technologies into the framework nor the potential evolution and emergence of new virtualization technologies.

Overall, the existing frameworks are too rigid. None of them provide a framework definition *flexible* enough such that it allows not only different wireless technologies but also different *virtualization perspectives* to coexist. Thus, they do not fully consider the different requirements from different services and applications. This is why this thesis first classified different virtualization *perspectives* in order to identify the different needs from different tenants in a virtualized ecosystem and the technologies that can support those perspectives.

This framework considers these multiple wireless virtualization approaches simultaneously and carefully defines how it interacts with existing technologies. Moreover, the proposed framework integrates most if not all the main features from the other frameworks and extends them with additional *configurability of the framework itself*. As such, it can be viewed as a true *meta*-framework or *toolbox of frameworks* for wireless virtualization, from the infrastructure-level to the spectrum level.

Finally, the framework can also be viewed as an *extension* to VANI/SAVI [15][17] to include *wireless resources* as part of the virtual infrastructure testbed. Even though the proposed framework is independent from SAVI, its implementation is closely related to the SAVI and OpenStack architecture, as will be discussed in the following Chapter 5. In fact, the concept of agent plane and virtualization plane follows the same concept of separation between the control plane and application plane in SAVI [17]. Nevertheless, in this case, the agent plane is entirely configurable by the tenants themselves.

3.4 Chapter Summary

In this chapter, the architecture for a generic and modular wireless virtualization framework has been formulated. First, this chapter presented three different wireless virtualization *perspectives*: flow-based virtualization, protocol-based virtualization and spectrum/RF virtualization. This was followed by a discussion on the reason why these perspectives must *coexist* and not compete with each other. Then, this chapter presented the challenges and requirements of a hypothetical virtualization framework that can simultaneously support multiple perspectives. Following these requirements, the general architecture of such a framework, including the local virtualization agent, was outlined. The wireless infrastructure-wide framework is separated into the control and management layer and the virtualization layer. The virtualization layer itself is then composed of the agent plane and the virtualization plane. How different wireless virtualization perspectives can be supported by the agent was discussed. With an open and evolutionary approach, the proposed framework is not the end solution but is instead the *beginning* of a solution. As such, it provides a balance between a research testbed and a deployable technology.

Chapter 4

Aurora: Virtualization and Software-Defined Infrastructure Platform for Wireless Access Networks

The previous Chapter 3 outlined the general challenges, requirements and architecture for a generic, modular and multi-perspective virtualization framework. The missing element in many of the existing virtualization architectures, such as the ones presented in [18] and [9], is the implementation and deployment of the framework. Thus, this chapter expands on the *implementation architecture* of the framework: a virtualization and software-defined infrastructure *software platform* for wireless access networks codenamed *Aurora*. In this chapter, how the framework requirements are satisfied through the system design of Aurora is explained, with a particular emphasis on the architecture of the local virtualization agent.

4.1 Introduction to Basic Concepts in Aurora

This section details the main objectives of Aurora aligned with the requirements discussed in Subsection 3.2.2 and how they are handled in this implementation.

4.1.1 Motivation, Inspiration and Objective of Aurora

As discussed in Subsection 3.3.3, the general framework is greatly influenced by the architectures presented in [18] and [9]. The implementation of Aurora is oriented towards being a service component for the cloud infrastructure platform OpenStack (Subsection 2.2.4) and the SAVI testbed (Subsection 2.2.2). Currently, OpenStack does not have any service that can explicitly interact with wireless technologies. The closest service is *Neutron*, which offers APIs for the network virtualization and SDN functions on the virtual infrastructure. The SAVI testbed architecture extended OpenStack with additional software-defined infrastructure services such as graph-based topology service (*Whale*) and SDI management (*Janus*). Aurora also attempts to extend the OpenStack family of services by providing orchestration and virtualization tools for wireless resources with different wireless virtualization perspectives.

The reason why Aurora was not made an extension to Neutron but kept as a separate service is the need for low-level wireless virtualization features (i.e. protocol and spectrum perspectives) that has no direct relationship with the network functions of Neutron. However, *why choose OpenStack for the first implementation of the generic framework?* The main reason is that OpenStack is already widely deployed in both commercial and research environment, including the SAVI testbed. Aurora, by providing "OpenStack-like" control and management client console commands and REST APIs, makes wireless virtualization concepts easier to grasp for users who are familiar with OpenStack.

4.1.2 Resource Abstraction and Orchestration

Resource abstraction has been identified as one of the main requirements for a generic wireless virtualization framework in Subsection 3.2.2. In Aurora, there are three classes of resources abstracted in software: virtual wireless networks, virtual resource slices and physical resource nodes. The logical relationship between the different classes of resources is shown in Figure 4.1. These abstracted resources are defined as the building blocks of a virtual infrastructure in Aurora.

1. **Physical resource node (*wnode*)**: The physical resource node exist as an entity referred as *wnode* within the Aurora software platform. Although the full physical nodes are usually not allocated as resources to tenants, both the infrastructure manager and the tenants use them as a reference point to obtain virtual resources. Unlike

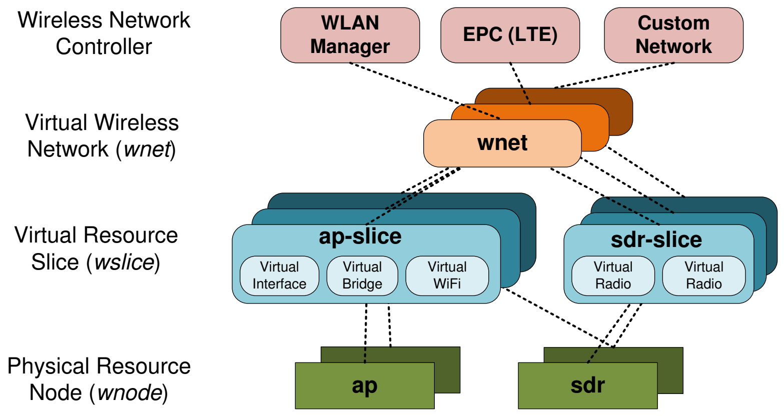

Fig. 4.1: Classes of Resources in Aurora Framework

with server virtualization, the *location* of the physical node is extremely important in the deployment of a virtual wireless infrastructure. Thus, the *wnodes* are *tagged* with *searchable* information about its physical characteristics and capabilities, including location. The Aurora framework can then perform filtering over *wnode* tags to satisfy the tenant's requirements. Different types of radio nodes are represented by different *flavors* of *wnode*, which can share similar functions but differ in terms of capabilities. The different *wnode* can include 802.11 access points (*ap*), software-defined radios (*sdr*), cellular basestations (*base*) and other wireless technologies. Even within a type of resource node, the capabilities will differ depending on the implementation. At the same time, a single *wnode* can support more than one type of virtual resource slice (*wslice*). Only the implementation of *ap* and, to some extent, *sdr* are covered in this thesis.

(a) *802.11 wireless access point (ap)*: This *flavor* of physical resource node represents an 802.11 AP that is virtualized using Aurora. This type of *wnode* only supports flow-based virtualization with possible partial procotol virtualization. In the current implementation, it only offers one type of virtual resource slice: *ap-slice*. This is the only type of resource node implemented to a workable state in this thesis. This resource node is implemented on PC Engines *alix3d2* APs [54] with two 802.11ab/g mini-PCI radio interface cards although it can be extended to any AP running Linux-based firmware.

(b) *Software-defined radio (sdr)*: The software-defined radio resource node can potentially support all three virtualization perspectives. It can offer more than one type of virtual resource slice. For instance, it can offer an instance of the SDR platform itself (*sdr-slice*) or an instance of a particular pre-packaged protocol standard, such as 802.11 (*ap-slice*). This resource node is planned on the WARPv3 FPGA-based SDR boards [55]. Possible implementation is discussed in Subsection 5.4.2.

2. **Virtual resource slice (*wslice*)**: The *wslice* is the virtual instance of the wireless resource node that is allocated to the tenants. The *wslice* is not a single virtual resource but is more a *package* or *container* of different virtual resources (bandwidth, VAP, etc.) supported on the *wnode* that can be configured and controlled by the tenant. As such, even within the same type of *wslice*, there are large variations in the capabilities offered on the slice, just like the *wnode*. The *wslice* is defined by a setup contract specified by the tenants and validated by the infrastructure manager in Aurora. Then, the tenant has direct control and management of the *wslice* within the limitations defined by the setup contract. Each *wslice* can join a *wnet*. The *wslice* also inherits some tags from the *wnode* from which it spawned depending on the capabilities it obtained from the *wnode*. Then, *tenant-defined tags* can be added in order to partition *wslice* into customized management groups by the tenants.

   (a) *ap-slice*: This *wslice* contains mostly *resource components* used to setup and configure flow-based virtualization, such as virtual interfaces, bridges, queues, schedulers and radio interfaces. It can support an instance of virtual access point (VAP). These resources components are interfaced through plug-ins presented in Subsection 4.1.4. One particular implementation of ap-slice is discussed in Subsection 4.2.4.

   (b) *sdr-slice*: This *wslice* contains similar virtual resources to *ap-slice* except with the addition of multiple radio configuration profile for different radio interfaces and potentially the support for loading tenant-made protocols. The *sdr-slice* can also be configured and abstracted as a *ap-slice* by defaulting the additional capabilities.

3. **Virtual wireless network (*wnet*)**: A *wnet* is a *group* of one or multiple *wslice* of different types. It is similar to a subnet but has additional wireless network manage-

ment functionalities. The networking functionalities of the *wnet* is delegated to an external network virtualization and SDN framework (see Section 5.5). A *wnet* is used by Aurora to offer support for *wireless* networking functions that can be adapted to a specific standard (WLAN or cellular). In order to achieve this, the *wnet* connects the virtual wireless network to a *wireless network manager*, a software controller that manages the policies, virtual resources and coordination of the wireless nodes. This manager can be a tenant-owned custom controller or a default controller provided by the Aurora framework. It can be an entire network of its own, such as the Evolved Packet Core (EPC) in the case of LTE technologies. The network-wide conflict resolution and contract reinforcement is then performed by the management modules inside *Aurora-Manager* (see Subsection 4.2.2), which are not covered in this thesis. Thus, the current implementation of *wnet* is only a place holder for a future full implementation.

This arrangement of resources allows a building block-based creation of virtual infrastructure. In order to support the concept of *scalable complexity and separation of concerns* for different tenants, the configuration blueprint of a virtual resource slice is accessible to all tenants in the form of JavaScript Object Notation (JSON) files. Pre-made *default* infrastructure blueprint packages are then used to provide simple virtual infrastructures for tenants who do not need or want full reconfigurability (see Appendix A.6). The tenants can also create their own packages by writing orchestration scripts using the Aurora APIs and the JSON configuration files. The Aurora framework is mainly implemented in Python [56] due to its dynamic typing features. Python is also the language of implementation for OpenStack and most of the SAVI testbed services, facilitating the integration between Aurora and these platforms. The relationship between the different resources are stored using a centralized structured query language (SQL) database, more specifically MySQL [57]. Each class or resources has its own SQL table similar to the implementation inside OpenStack. Samples of Aurora database format are shown in Appendix A.3.

4.1.3 Multi-tier Tenancy

With the different classes of resource abstraction presented in the previous subsection, there is a great variety in the types of virtual wireless infrastructure a tenant can assemble. In fact, as mentioned in the previous chapter, different tenants have different needs and requirements. As shown in Figure 4.2, different tenants can own different portions of the infrastructure, not necessarily with wireless resources. This gives rise to the concept of *multi-tier tenancy*, a concept necessary to support the hypothetical ecosystem presented in Section 1.4.

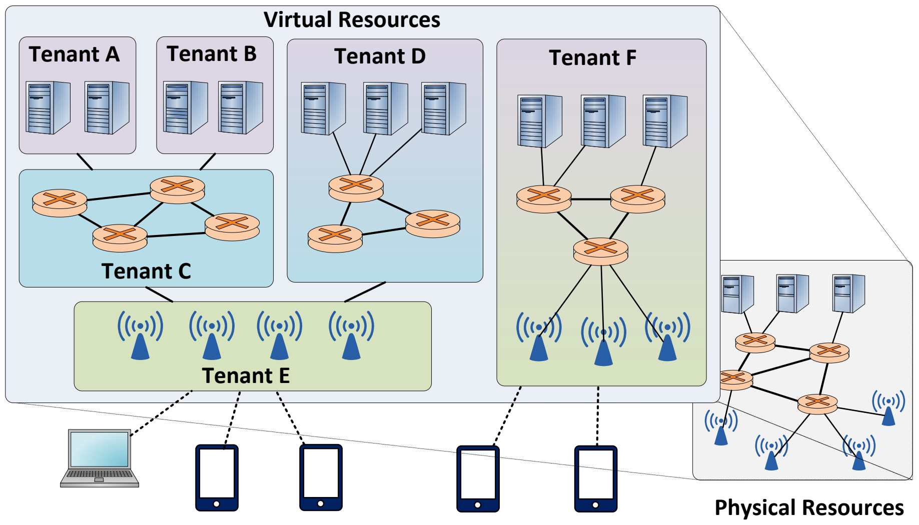

Fig. 4.2: Multi-Tier Tenancy in a Virtualized Infrastructure With Wireless Resources

There are different *tiers* of tenants that can interact with the virtual wireless resources. They can be grouped as two main tiers: wireless *infrastructure-level* tenants and wireless *service-level* tenants.

1. **Wireless infrastructure-level tenants**: Infrastructure-level tenants own wireless resources inside Aurora (Tenant E and F in Figure 4.2). The minimum requirement for ownership is the configuration and creation of a *wslice* by the tenant. Even within infrastructure-level tenants, there are various sub-tiers of tenants that differ from each other in terms of capabilities. For instance, a tenant does not necessarily need to perform active control and management of its wireless resources. As a practical example, how the different types of tenants supported on standard 802.11 APs in Aurora are detailed as follows:

   (a) *Base tenant with full radio interface and base BSS*: Even though in a regular enterprise-grade 802.11 AP there can be multiple radio cards, only one radio configuration profile is supported per physical radio interface. Thus, without the implementation of partial protocol-based virtualization, only a *single* tenant can configure the radio parameters (channel, mode, power, etc.) per physical radio interface. In such a scenario, Aurora must *allocate* the full radio interface to a tenant that requires control over radio configuration parameters. Such allocation (not virtualization) is limited by the number of physical radio cards. However, since standard 802.11 allows multiple BSS over the same radio interface, Aurora also allows tenants *with lower capabilities* to share the same radio interface. In other words, there is a *base tenant* that owns the radio interface with a master or *prime* BSS. This tenant can then specify how many additional BSS the radio interface can support and if the radio is *shareable* to other tenants. Such limitations are non-existent in resource nodes with at least partial protocol virtualization implemented since each tenant can have its own radio profile.

(b) *Guest tenant with additional BSS over same radio interface*: The feature of multiple BSS on the same radio exists as the concept of VAP. Each VAP has its own BSS (ESSID, BSSID, encryption, etc.) and virtual wireless interface. Aurora allows additional *guest tenants* to share the radio interface with each other and with the base tenant (with its permission of course). Since the radio parameters can only be controlled by a single base tenant, guest tenants have a more restricted set of capabilities. The downside of this approach is that if the base tenant disables the interface or deletes its slice, all guest tenants attached on the radio interface will lose their BSS. In future implementations, it will be possible to migrate BSS over different radio interfaces. However, in practical scenarios, such as the deployment of Aurora inside the SAVI testbed, there will a default base tenant for accessing the SAVI network that remains active at all time (see Section 5.2).

2. **Wireless service-level tenants**: Service-level tenants are not actual tenants per se in the wireless infrastructure as they do not own any wireless resources. Instead, these are mainly tenants in the computing and networking infrastructure that require wireless access to their virtual network and services (Tenant A-D in Figure 4.2). They do not necessarily care about the wireless configuration or the wireless network setting. In the case of the SAVI testbed with OpenStack and Aurora, there are advantages of having direct connectivity from *within* the virtualized infrastructure. First, there is a

dedicated network that can support experimental network architectures developed for Future Internet, with different innovative security and mobility features. For instance, a subscription-based access model in which service-level tenants can only be reached *at the infrastructure level* by mobile clients subscribed to that particular service, unlike the public access format of the current Internet infrastructure, can be reinforced. In the case of the SAVI testbed, a default SAVI base wireless infrastructure-level tenant can manage the wireless access network to which other service-level tenants can subscribe to. Therefore, service-level tenants are often not managed by the framework itself but by another infrastructure-level tenant. The setup and configuration of such a service in Aurora is described in Subsection 5.5.

4.1.4 Modular and Evolvable Framework

One of the important requirements defined for the framework in Subsection 3.2.2 was the evolvability of the framework. Aurora, as a software implementation, is designed with such requirements in mind. The Aurora framework must allow different virtualization perspectives to coexist and integrate different virtualization technologies. The addition of new perspectives and technologies at the virtualization layer functionally affect Aurora at all levels: the operation of Aurora virtualization agents, the Aurora resource management and the tenant interaction with Aurora. In order to maintain modularity, different software design decisions were made at the various layers of the Aurora framework. Overall, the "toolbox" approach is applied, in which the implementation of a particular function is selected from different *flavors*. Similar functions of different *flavors* are grouped inside an Aurora *module*. A specific flavor of a module is referred to as a *plug-in* and interact with a specific type of resource component. In the basic implementation of Aurora in this thesis, only flow-based wireless virtualization is supported on infrastructure-mode 802.11 APs. Ongoing effort on supporting partial protocol virtualization on FPGA-based SDR is under way.

Three Aurora modules are used by the virtualization agent for achieving basic flow virtualization on 802.11 APs: *virtual interfaces*, *virtual bridges* and *virtual WiFi radio*. These are implementations of the abstraction module and plug-in inside the agent from the general framework architecture presented in Subsection 3.3.2 (see Figure 3.6). These modules allow the tenants to define and configure their own *control* and *data paths*, which are important assets for implementing flow-based virtualization. Detailed operations of these modules are provided in Subsection 4.2.4, 4.2.2 and 5.1.

1. **Virtual (network) interfaces**: Virtual network interfaces (*VirtualInterfaces*) represent binding endpoints of a virtual link where data is exchanged. In Aurora, there is a distinction between *network* interfaces and *radio* interfaces. Here, the radio interface refers exclusively to *wireless* interfaces that connects the AP to the radio interface card. Network interfaces include all other interfaces (both physical and virtual). In Aurora, different flavors of virtual network interfaces are provided to tenants, each fulfilling different roles ranging from tunneling to VLAN. In this thesis, the *Capsulator* [58] and virtual Ethernet (*veth*) plug-ins are implemented. Other types of interfaces, such as VLAN and GRE tunnels, can be developed separately as extension plug-ins to the *VirtualInterfaces* module.

    (a) *Capsulator tunneling interface*: Capsulator is a custom over-IP tunneling program [58] that provides basic flow isolation between tenants and is used in many research projects involving OpenFlow such as [7] and [31]. For outbound traffic, Capsulator takes normal packets from a *border port*, encapsulates them with a *tunnel ID* and send them out on the attached interface to an endpoint IP. The inbound traffic is decapsulated based on tunnel ID and sent out on the matching border port. The original program only provides a match based on tunnel ID. The version inside Aurora has been modified to support match based on both tunnel ID and endpoint IP in order to allow different tenants to have the same tunnel ID.

    (b) *Virtual Ethernet interface (veth)*: Veth is a basic virtual interface that has its own MAC address and duplicates the traffic to and from another interface on which it is attached to. In Aurora, these interfaces are attached to the virtual wireless interfaces to *mask* them as virtual network interfaces. This procedure is necessary if wireless interfaces are to be attached to an instance of Open vSwitch (OVS). Wireless interfaces lose connection to the wireless driver when directly attached to OVS, as OVS intercepts the traffic on the interface before the drivers.

2. **Virtual bridges**: In Aurora, the *VirtualBridges* modules provide the internal layer

2 connection *between virtual interfaces*. Most of the virtual interfaces already implicitly function like a bridge between the attached interface and the virtual interface. However, they only operate on a single interface (and *creates* the other). Virtual bridges operate on *two or more* interfaces (virtual or not). The current version of Aurora includes plug-ins for different programs that behave like bridges inside the Linux operating system such as Open vSwitch [23] and the actual Linux bridge.

(a) *Open vSwitch (OVS)*: The OVS software [23] allows tenants to create a virtual switch with multiple virtual interfaces attached as ports. OVS also supports the OpenFlow protocol, enabling tenants to control the virtual switch using an OpenFlow controller, satisfying the direct control and management by tenant requirement. This plug-in can be used by infrastructure-level tenants (*direct* OpenFlow control) as well as by the Aurora framework itself as the default bridge in conjunction with FlowVisor (*delegate* OpenFlow control). The different modes of OVS inside Aurora are explained in Subsection 5.1.

(b) *Linux bridge*: This plug-in simply provides the wrapper for Aurora to interact with the basic Linux bridge, which sends packets arriving on one interface to the other interfaces and vice versa. It is used as the most basic form of gateway connection between the AP and the wired network.

3. **Virtual WiFi radio (interfaces)**: *VirtualWifi* is a 802.11-specific module that can be used in both flow-based and protocol-based virtualization core modules. In flow-based virtualization on APs, this module contains plug-ins to interact with the wireless drivers of the AP. This module allows tenants to configure the wireless radio parameters of the radio interfaces. It can also create virtual radio interfaces with which the *VirtualInterfaces* and *VirtualBridges* modules can interact with. The management and allocation of new 802.11 BSS to tenants is performed through this module. It also counts the number of radio and BSS resources available. The only flavor currently implemented in Aurora is the plug-in for OpenWrt.

(a) *OpenWrt*: OpenWrt is an open-source Linux-based firmware for wireless access points and wireless routers [59]. The OpenWrt flavor of *VirtualWifi* can interface with OpenWrt-specific configuration system known as the *unified configuration interface* (UCI) and the *hostapd* 802.11 driver. The concept of base BSS and guest BSS mentioned in the previous subsection is implemented through the OpenWrt plug-in. The radio parameters are configured separately from the BSS parameters. However, the AP is only operational when both radio and base BSS configuration are completed.

4. **Traffic scheduling**: Additional modules for flow-based virtualization perspective, such as scheduling and queuing, are not fully implemented in the current iteration of Aurora. This is because the pure usage of such mechanisms on wireless APs is not sufficient to guarantee QoS. These modules are discussed as extensions in Subsection 5.4.1.

The addition of a new module or plug-in, such as new scheduling and queue mechanisms, does not impact the core engine of the Aurora agent or manager. Aurora implements the core engine as basic generic routines that call abstracted functions in a fashion similar to pseudo-code. Recall that there are two types of interaction tenants can have with the virtual resources allocated to them (Section 3.3). Only the setup and configuration operations are handled through the Aurora framework. Direct control and management functions are handled through the technologies orchestrated by Aurora but not directly through the framework itself, with the exception of OpenFlow (see Section 5.1). For setup and configuration, both the Aurora manager and agents use *JSON configuration files* to exchange information. JSON is chosen for its relatively simple structure, its flexible format and its close resemblance to Python dictionary objects. As such, modules are grouped as independent entries to a *list* of *configuration modules*. The JSON configuration format of different Aurora modules is described in Appendix A.2.

Each resource component associated with a plug-in has two fields: *flavor* and *attributes*. The *flavor* field identifies the type of plug-in that has to be loaded for a given module. The *attributes* field is a container for a dictionary of configurable plug-in parameters. The JSON parser in Aurora dynamically loads modules based on the contents of the slice configuration blueprint and the capabilities of the affected resource node. The detailed parsing, validation and generation of slice configuration blueprints are delegated to individual modules and plug-ins. As such, both modules and plug-ins can be added or removed without affecting the rest of the Aurora framework. This also means that the Aurora software is easily relocatable to operate on different types of resource nodes (assuming support for Python) by using different plug-ins to match with the specific technologies available on the *wnode*. In such

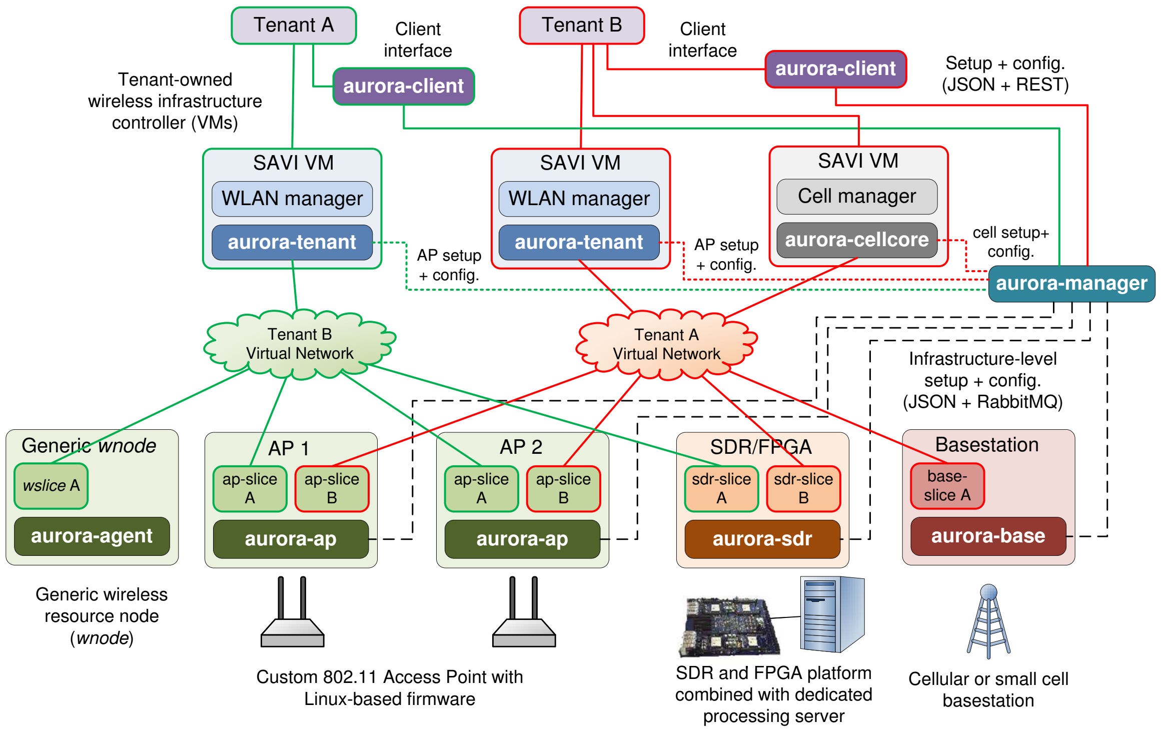

Fig. 4.3: General Aurora Architecture With Different Types of Aurora-Agents

a way, Aurora is able to act as a common platform for different virtualization perspectives and technologies. The detailed operations of the different entities inside Aurora, including the agent and the manager, are described in the following section.

4.2 Aurora Architecture

The implementation architecture of Aurora is based on the generic framework architecture presented in Section 3.3 (see Figure 3.3), which is itself based on a mixture of different infrastructure virtualization architecture, as explained in Subsection 3.3.3. The high-level Aurora architecture, as shown in Figure 4.3, is divided into four main components: *Aurora-Agent*, *Aurora-Manager*, *Aurora-Client* and *Aurora-Tenant*. The first three are the core components of the framework whereas *Aurora-Tenant* is a set of additional tools for tenants that complements the framework. This thesis is mostly focused on Aurora-Agent, in particular Aurora-AP for 802.11 access points. The operation of other components is also presented but only the modules in direct relationship with Aurora-Agent are covered.

Thus, the various management modules that can be integrated within Aurora-Manager and the accompanying tools in Aurora-Tenant are not presented in this thesis.

4.2.1 Aurora-Client: User and Tenant Interface to Virtualization Framework

In Aurora (and OpenStack in general), a tenant represents a project or a set of virtual infrastructure resources. As such, a tenant is most likely operated by individual *users*, who set up and configure the resources of the tenant slice. The Aurora-Client is the Python-based console that provides the console commands for users to do so. It is designed to be similar to the OpenStack client. The architecture of Aurora-Client is shown in Figure 4.4.

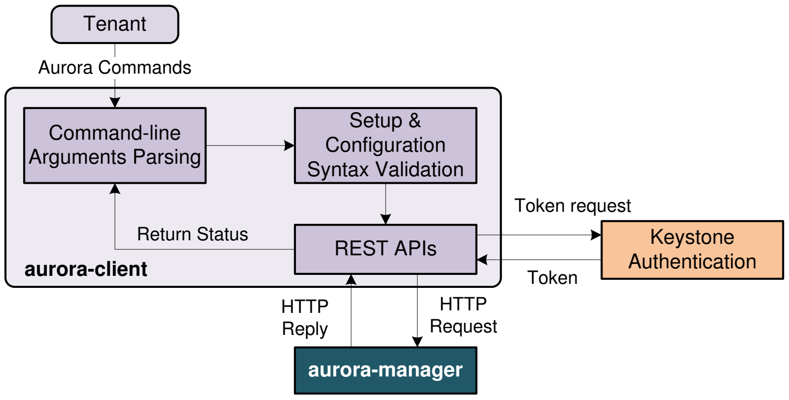

Fig. 4.4: Aurora-Client Implementation Architecture

Aurora-Client takes console commands from a user and performs basic syntax validation over them. Advanced validation requiring information about the state of the infrastructure are performed at the Aurora-Manager. Authentication of the users and tenants is performed through a token-based authentication service provided by OpenStack called *Keystone* [60]. Then, the client request is sent out to the Aurora-Manager through a HTTP-based REST API with the token obtained from Keystone. The implementation details of the authentication and Keystone are not of much interest to this thesis and can be found in the OpenStack manuals [61]. The basic interaction for Aurora-Client commands is illustrated in Figure 4.5

As mentioned in Subsection 4.1.2, the main classes of resources are *wnode*, *wslice* and *wnet*. Aurora-Client supports similar sets of commands for each of these categories. In the current implementation of Aurora, only APs *wnode* and *wslice* are supported. As such,

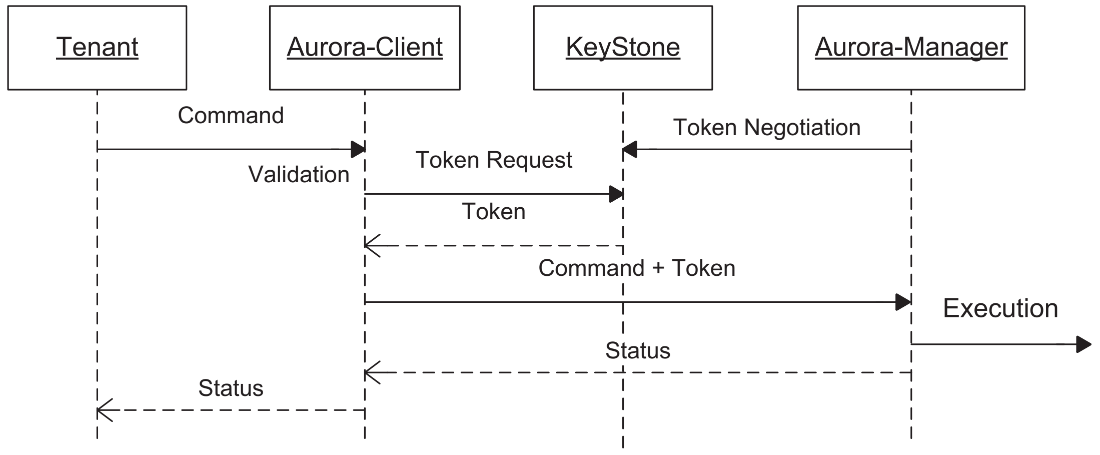

Fig. 4.5: Aurora-Client Handling of Tenant Commands

the Aurora-Client commands operate on *ap*, *ap-slice* and *wnet*. In general, *list* and *show* commands work on all classes of resources. The *list* command typically displays the list of resources of a certain type available to the tenant. It is *scoped* such that only resources owned by the tenants are visible to its users, with the exception of physical resources (*ap*) which are available to all tenants. The *show* command gives more details on a specific resource referred by name or universally unique identifier (UUID). Virtual resources, such as *wslice* and *wnet*, have *create*, *modify* and *delete* commands. The *modify* command allows the slice to be recreated based on new slice configuration blueprint without regenerating the metadata of the slice (i.e. UUID and relationship with other resources). For simple parameters, a console is sufficient. For the full customization of the virtual resources, JSON slice configuration blueprints are used. Additional commands specific to each class of resources are provided, such as the ability to *add* and *remove wslice* from the *wnet* or the ability to add and remove tags from *wslice* (see Subsection 4.1.2). A more detailed list of the different commands in Aurora-Client is provided in Appendix A.1. The infrastructure administrator can also use a similar client but with additional commands and capabilities available. The administrator has full knowledge and control of resources from all tenants.

4.2.2 Aurora-Manager: Virtualization Manager for Wireless Infrastructure

The Aurora-Manager is the implementation of the *infrastructure manager* presented in Section 3.3. It performs three main roles inside the framework. The first role is to provide

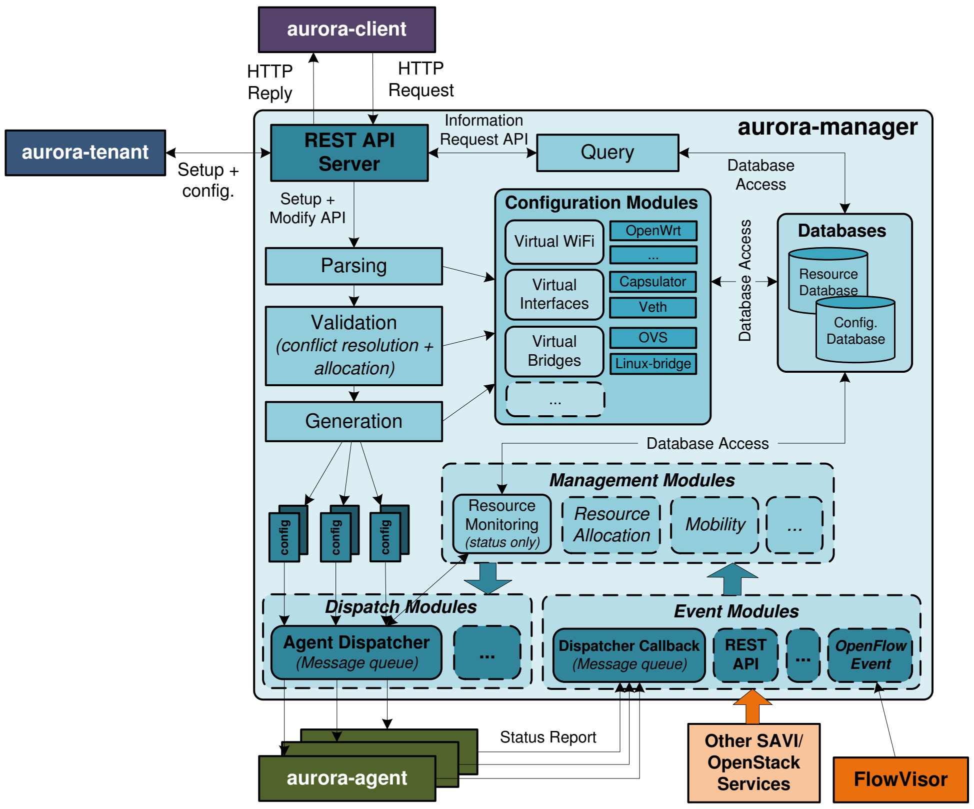

Fig. 4.6: Aurora-Manager Implementation Architecture

access to infrastructure information. The second role is to redirect the setup and configuration of the virtual wireless resources from the client to the virtualization agents, including the detailed settings of the resource components of each slice. The third role is the hosting of wireless infrastructure resource allocation and management services. The first two roles result from the processing of API commands received from Aurora-Client instances.

For the first two roles, a REST API server is deployed inside Aurora-Manager to handle both API requests from Aurora-Client instances from different tenants and API requests from Aurora-Tenant VMs (see following subsection). The current REST API server is only a HTTP server without the full RESTful implementation yet. The overview of the architecture of Aurora-Manager is shown in Figure 4.6. Aurora-Manager contains two *persistent* management databases to keep record of the status, attributes, capabilities and configuration of both physical and virtual resources. The garbage collection of both databases are maintained through the configuration modules and the status monitoring module.

1. **Resource database**: Even though MySQL [57] provides a very fast and reliable database, configuration information are too complex and flexible to be implemented only using MySQL tables. Therefore, the first database is a SQL-based *resource database* and contains only simple resource attributes (name, UUID, tenant, status, tags, etc.) and *relationships* between resources. Access control (read-write concurrency) is handled automatically by MySQL. Each class of resources (*wnet*, *ap*, *ap-slice*) and each type of tags (location and tenant-added, see Subsection 4.1.2) has its own MySQL table. Thus, a total of five MySQL tables are currently used: *ap*, *ap-slice*, *wnet*, *ap-tag* and *ap-slice-tag*. The format of the SQL tables are provided in Appendix A.3.

2. **Configuration database**: The second database is a JSON-based *configuration database* that stores the full *wslice* configuration blueprints for each tenant. Blueprint files are grouped by tenant and a series of blueprints owned by a single tenant can only be edited by a single module at any given time to preserve database integrity. The format of the configuration database is provided in Appendix A.2 and A.4.

For handling client requests of infrastructure information, Aurora-Manager queries the resource database and the configuration database. The manager then returns the result of the query to the client through the API server. For setup and configuration such as *slice creation*, Aurora-Manager loads different *configuration modules* based on *dependencies*. Each module is associated with an abstraction module on Aurora-Agent. The lower-level and "foundation" modules are always checked first. In this implementation of Aurora, the order of modules to check are, in order: *VirtualWifi*, *VirtualInterfaces* and *VirtualBridges*. Aurora-Manager uses the modules to parse and functionally validate the JSON configuration file obtained from the client. Conflicts such as invalid or duplicate names and unsupported capabilities are resolved. In other words, the manager has to determine (1) what the tenant wants and (2) if the resource nodes can support it. This operation potentially requires access to the databases and other management modules. If the validation is successful, JSON files are generated for each *wslice* and posted to the Aurora-Agent through a *dispatch module*. More specifically, a subscription-based dispatcher using RabbitMQ [62] is used. At this point, the client request is returned with apparent success. The dispatch operation is registered to a pending queue in a *resource status monitoring module*, a wrapper for handling callbacks from the replies of Aurora-Agent.

The last role of Aurora-Manager is to host *management modules* (i.e, not agent-related unlike the configuration modules currently implemented) that provide various wireless network management functionalities. These functionalities can include, but are not limited to, mobility management services, dynamic provisioning of radio nodes, migration of virtual radios, outage handling and dynamic spectrum reuse. Of course, some of these functions can be directly implemented by the tenant themselves using their own management system within their own slice (see next subsection on Aurora-Tenant) or implemented by a specific technology on the resource node (i.e, cognitive radio). However, any autonomic functions that are abstracted away from the tenants or involve cross-tenant allocation must be provided as part of the infrastructure framework (Aurora) itself. The simplest management module would be a resource monitoring module to receive status updates from agents and synchronize the databases. None of these management functionalities, with the exception of a basic resource status monitoring module, are implemented as they fall outside the scope of this thesis, which is only the bare minimum system architecture of the framework. However, they can be added as extension modules to Aurora-Manager, as shown in Figure 4.6.

1. **Resource status monitoring module**: The status monitoring module has two tasks: to monitor the status of virtualization agents and to handle slice creation/-modification return status received from them. When a slice setup or modification command is issued by the client, the slice configuration blueprint is saved or updated in the configuration database and an entry for the slice is created in the resource database (if it is a new slice). The command dispatched to the agents is registered on the resource status monitoring module, who watches for operation status updates coming from the agents. There are two general categories of failures: *operation failure* and *node/agent failure*. The operation failure occurs when there are errors that prevent a slice from being fully created. The node/agent failure occurs when the connectivity to the node is broken or when the node or agent crashes or powers off. Thus, the connectivity between the manager and the agents is periodically verified by this module to detect any node/agent failure. The general rule is that once an agent is marked as down, it must be restarted once the connection is established. The agent sends an initialization request when restarted. If an agent previously marked as down is suddenly active without such initialization request, it is forced to restart by

the manager. This behavior can be modified in future extensions once more robust fault-tolerant and database synchronization mechanisms are implemented. The state transition and possible status of a slice is shown in Figure 4.7. These states are valid and accessed throughout all modules of Aurora-Manager, not just the resource status monitoring module.

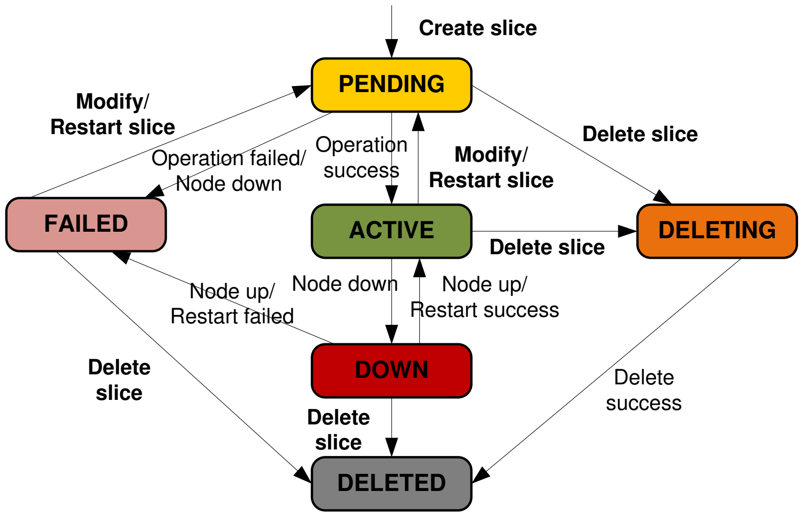

Fig. 4.7: Wireless Slice Creation/Modification/Restart Status Cycle

(a) *PENDING*: The status of the slice is set to PENDING when a create, modify or restart command is issued on a new or existing slice, before receiving any status update or confirmation from the agent instances. During this state, the agent is assumed to perform the slice creation routine (see Subsection 4.2.4).

(b) *ACTIVE*: The status of the slice is updated to ACTIVE if the create, modify or restart operation is successful and confirmed by the agent through the status monitoring module. The slice also enters this state if it is recovered successfully from a DOWN state (node/agent failure).

(c) *FAILED*: The status of the slice is updated to FAILED if an operation failure to a command is returned by the agent, if a node/agent failure occurs when the status is PENDING or if the re-initialization of the slice from a DOWN state (node/agent failure) is unsuccessful. A FAILED slice can be restarted or modified by the tenant.

(d) *DOWN*: If a node/agent failure occurs when the status of a slice is ACTIVE, the status is changed to DOWN. This state is used to allow the agent to automatically restore previous ACTIVE slices after node/agent recovers from failure. The recovering process when the node and agent come back online is handled through the *agent initialization* process covered in Subsection 4.2.4. On the other hand, PENDING slices are not restored if a node/agent failure occurs. Instead, the slice status changes from PENDING to FAILED and has to be restarted manually by the tenants.

(e) *DELETING*: The status of the slice is changed to DELETING if a slice deletion command is issued when the slice is in PENDING or ACTIVE state. This status is used to mark slices that needs to execute de-initialization operations in order to be deleted. It also blocks further operations on them. If a delete command is issued when the status is FAILED or DOWN, the DELETING state is *skipped* as there are no additional routines that need to performed by the agents. During this state, the agent performs the slice deletion routine (Subsection 4.2.4).

(f) *DELETED*: In general, slice entries in the resource database are not automatically removed when failure is encountered. Instead, they persist until the tenant removes them manually through Aurora-Client (or through automated scripts). Nevertheless, even when the tenant successfully deletes a slice through Aurora-Client, its database entry is not removed from Aurora-Manager immediately. Instead, its status is marked as DELETED. Then, the DELETED entries are purged after a certain time period. DELETED slices do not show up through the normal *list* commands.

To support some of these management functionalities, *event modules* are used to capture *events* from the resources managed by Aurora-Manager. An event abstraction layer is required to handle different types of events coming from different sources and through different technologies. For instance, callbacks from Aurora-Agents using RabbitMQ can be considered events. Input from other OpenStack/SAVI services, such as OpenFlow-based callbacks from the SAVI FlowVisor controller, can also be considered events to the Aurora management modules. Finally, a dedicated REST API server for agents is used as an event module during the initialization of the agent to which agents can send requests for retrieving all relevant slice configuration blueprints (see Subsection 4.2.4).

4.2.3 Aurora-Tenant: Tools for Tenant-Owned Network C&M

As mentioned previously, the tenant can deploy its own wireless network controller inside the virtual infrastructure, typically on a VM allocated through cloud computing services, in this case Nova from OpenStack/SAVI. Aurora-Tenant is the official framework-provided ”package” to do so. This service component is not a mandatory part of Aurora because tenants can implement their own tools to manage their resources and are encouraged to do so. However, Aurora-Tenant aims at providing pre-made tools that improve the accessibility of Aurora, similar to *platform-as-a-service* (PaaS) in cloud computing. These tools can range from SDN-based controllers to wireless network management software.

The Aurora-Tenant can also call the REST APIs to the Aurora-Manager to perform setup and configuration of the virtual wireless infrastructure. A dedicated control path can potentially be used for these requests as the Aurora-Tenant can reside inside the virtualized testbed infrastructure. This is to allow tenants to co-locate their own setup orchestration tools (through Aurora-Manager) with the direct control and management of virtual resources (directly to virtualization plane of physical resource nodes). A different set of commands and APIs can be exclusively supported for special management functionalities. Aurora-Tenant is not implemented within the scope of this thesis. Some of its possible features are discussed as future works in Section 6.2.

4.2.4 Aurora-AP: Local Virtualization Agent for 802.11 Access Points

The generic service component that implements the local virtualization agent from Subsection 3.3.2 is called the Aurora-Agent. The specific implementation of Aurora-Agent must be customized to work with particular wireless technologies, such as 802.11 APs, SDRs and potentially cellular basestations. The implementation covered in this thesis is the Aurora-AP, a local virtualization agent for 802.11 APs. Aurora-AP is fully implemented in the *agent plane* (see Subsection 3.3.2). Only the *broker mode* is currently implemented. In other words, Aurora-AP only orchestrates the setup and configuration of other technologies, called *resource components*, through plug-ins grouped by *abstraction modules* (see Subsection 4.1.4) in order to build the wireless slice (*ap-slice*). The actual slice isolation and hypervisor functionalities are implicitly achieved through resource components and not directly by the agent. As such, the operational performance of the virtualization is determined by the performance of the individual technologies and the complexity of the setup, not by Aurora-Agent. The architecture of Aurora-AP is shown in Figure 4.8.

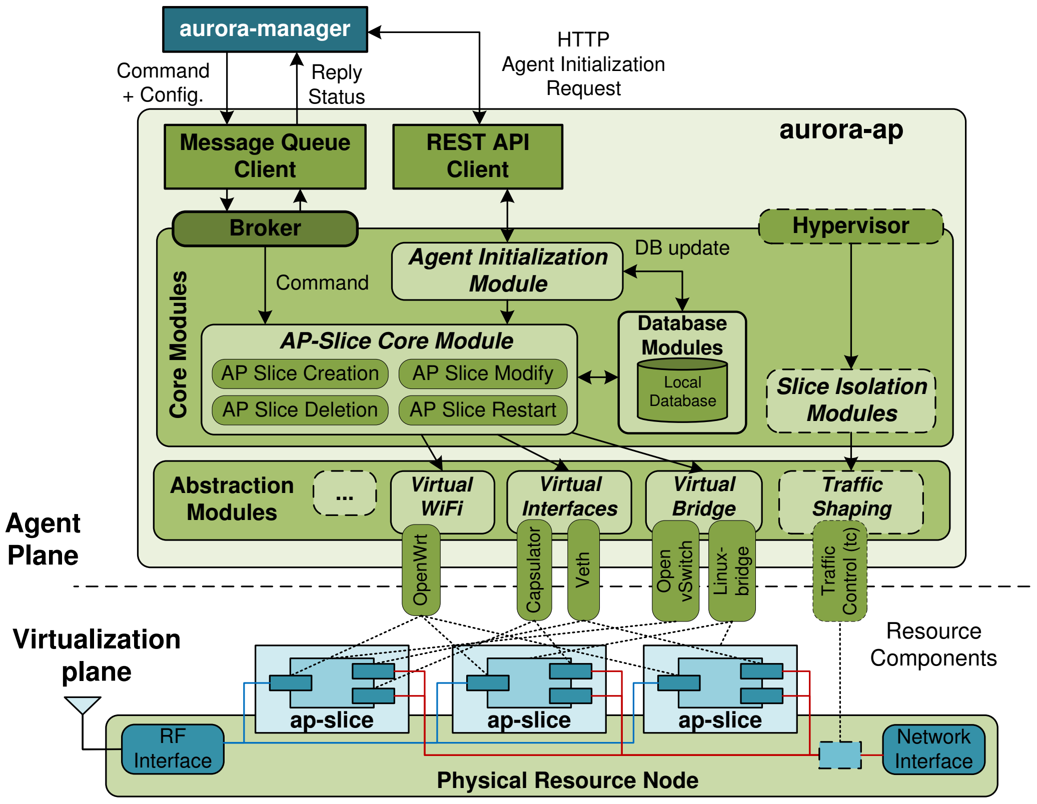

Fig. 4.8: Aurora-AP Virtualization Agent Implementation Architecture

In Aurora-AP, the *abstraction modules* are abstraction layers to specific technologies (resource components) interfaced through plug-ins. The current implementation only has abstraction modules sufficient to achieve very basic flow-based virtualization at the network level. As for the exchange of configuration information, the same JSON blueprint format used by Aurora-Client is applied but is assumed to be already *sanitized* and *validated* by Aurora-Manager. Aurora-AP has a *local database* to store configuration information of wireless slices currently active on the resource node. Currently, this local database is *non-persistent* as it only exist in the active memory of the agent software. This is in order to reduce the frequency of read/write access to the AP persistent memory, which is typically a flash drive with limited access lifetime. The slice information inside the database are grouped by tenant such that one tenant cannot access another tenant's slices. The structure of the local database is shown in Appendix A.5.

The core modules, as described in Subsection 3.3.2, contains the core logic of the agent. The core agent modules currently implemented in Aurora-AP include the *agent initialization module* and the *ap-slice core module*. These modules execute various "routines" to perform various tasks requested by the Aurora-Manager or to handle special situations. These routines can be different for different virtualization perspectives. The routines call into various abstraction modules, which in turn call into individual plug-ins to interact with a specific resource component.

1. **Agent initialization module**: Aurora-AP (like other Aurora-Agent) is assumed to be actively running on all physical resource nodes (*wnode*). The first time Aurora-AP starts or when it recovers from a failure, an initialization of the agent is performed. During this initialization, a synchronization of the *local database* is performed by rebuilding the slice from the configuration blueprint stored on Aurora-Manager. An agent initialization notice is sent to Aurora-Manager through a dedicated REST API server for agents. The configuration blueprints of all slices that are marked DOWN in the resource database on the manager are then sent to the agent for restoration. Finally, during agent initialization, a metadata file describing the physical capabilities of the resource node is parsed. Samples of such files are included in Appendix A.5.

2. **AP-slice core module**: The *ap-slice* core module handles all commands related to *ap-slice* instances, which is mostly a broker-mode flow-based virtualization module with some possible extensions to partial protocol-based virtualization (see Subsection 5.4.2). The basic routines implemented for this core module are: ap-slice creation, ap-slice deletion, ap-slice modification/restart and remote API processing.

   (a) *AP slice creation*: Recall that an *ap-slice* is a *wslice* with flow-based virtualization support for 802.11 APs. The agent has the task of orchestrating resource components to build a slice of the resource node specified through the configuration blueprint provided by the tenant and validated by Aurora-Manager. For flow-based core modules, Aurora-AP builds the data and control path on the virtualized resource node. It initializes the different abstraction modules and plug-ins specified through the configuration blueprint following a specific order, as specified in Subsection 4.1.4. The radio interfaces are created first, using the *VirtualWiFi* module. Then, the virtual interfaces attached to the Ethernet

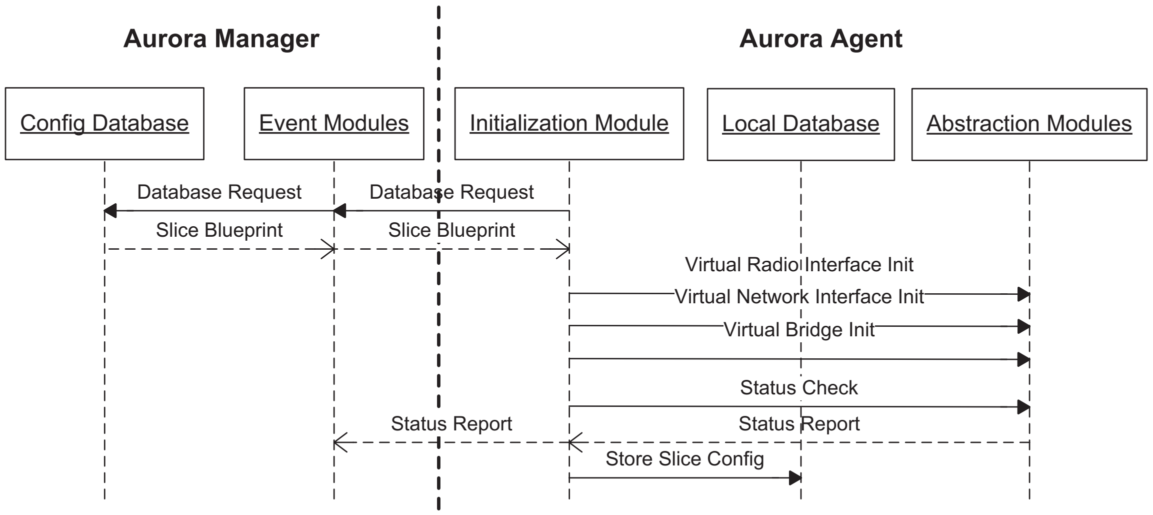

Fig. 4.9: Agent Initialization and AP Slice Creation

interfaces are initialized using the *VirtualInterfaces* modules. The virtual radio and virtual Ethernet interfaces are then connected together through the *VirtualBridges* modules. The order of resource component initialization is important during slice creation since interfaces must be functional before attaching them to a bridge module. Once initialized, the ap-slice instance is fully sustained by the resource components running as separate processes from the Aurora-AP agent. The slice configuration information is added to the local database if this operation is successful. Agent initialization and AP slice creation are both shown in Figure 4.9.

(b) *AP slice deletion*: The slice deletion command is only sent to the agent by the manager if the status of the slice is ACTIVE or PENDING. Otherwise, the slice is not confirmed to be active on the resource node (and thus no need for deletion). The resource components are tore down by a de-initialization routine during deletion. The order at which the components are deleted is exactly the reverse of that of the slice creation. Some components are Python sub-processes launched by Aurora-AP through the plug-ins. Others are existing daemon processes or services that the agent plug-ins merely interact with. Thus, the exact disabling mechanisms for these technologies are *implementation-specific* and performed through the plug-ins. In general, the Python sub-process library is used whenever possible. The configuration information of the slice is removed from the local database after this operation.

(c) *AP slice modification/restart*: Unlike the resource and configuration database on Aurora-Manager, the local database only stores slice configuration of successfully initialized and active slices and its resource components. Thus, if a failed slice must be restarted, a new albeit identical configuration blueprint must be sent from the manager (i.e. failed attempts are not stored on Aurora-AP). Overall, for both the modification and restart commands, the routine checks if the slice is active on the agent. If it is, the slice deletion routine is performed followed up by a slice creation operation with the provided configuration blueprint. Otherwise, only the slice creation routine is required. In other words, modify and restart are functionally the same at the agent level in the current implementation. Future extensions might include partial slice modifications without restarting the ap-slice. Some of these modifications can be performed through the *remote agent APIs*.

(d) *Remote agent API processing*: The remote agent API processing feature is initially used to debug the abstraction modules by allowing the manager to remotely call individual functions to modify the state of the ap-slice without tearing down the entire ap-slice. This feature is implemented by packing the module name, function name and function arguments inside a JSON structure (see Appendix A.2) and sending it over the message queue. This functionality is currently not fully available to regular tenants. Only selected functions are available for modifying the virtual bridge settings without resetting the entire *ap-slice*. However, in future extensions, it is possible to extend this feature to grant permissions to tenants for them to interact directly with slices they own, potentially through Aurora-Tenant. In other words, these APIs can be transformed into control and management operations instead of setup operations. Some of these extensions are discussed in 6.2.

In the current implementation with the *broker mode*, the signaling of events by Aurora-AP to Aurora-Manager is mostly limited to the notice of initialization using the REST API and the reply to commands using the message queue. Other events are directly handled by the individual components orchestrated by agent and not *through* the agent. This is also why the proposed agent architecture does not directly impact the performance of the existing technologies because the broker performs orchestration in the agent plane, out of the way of the datapath of the slice. The performance of each slice is then directly impacted by each tenant through their selection and configuration of resource components during slice creation.

Of course, in the full implementation, the virtualization plane is not entirely transparent to each tenant. For instance, certain modules for the *hypervisor mode*, such as inter-slice traffic shaping modules, are part of the framework and cannot be configured by the ap-slice tenants. The "hidden" performance cost of the traffic shaping module to provide slice isolation can be considered as virtualization overhead by the tenants. However, the key point is that these hypervisor-mode modules are still modularized. They are not fundamentally tied to Aurora and can be exchanged with other technologies. Additionally, even hypervisor-mode components can be offered as broker-mode components to a low level tenant (support for multi-tier tenancy). As such, Aurora provides high level of customization flexibility and modularity to both the tenant users and developers of the framework itself. In fact, traffic shaping modules in the AP are not even necessary in certain implementation scenarios when protocol or spectrum-based virtualization (MAC scheduler) is available in conjunction with an advanced network virtualization framework (network traffic shaping and flow QoS). Thus, Aurora is designed with extensibility in mind without over-relying on one particular technology.

4.3 Chapter Summary

In this chapter, the architecture of Aurora, the software implementation of the proposed multi-perspectives wireless virtualization and SDI framework, was presented. The motivation and inspiration behind Aurora, as well as some of its key concepts, were exposed. More specifically, the concept of *multi-tier tenancy*, different *classes* of resource abstraction and *modularity* of the architecture are explained in the context of Aurora. The architecture of the different Aurora service components, such as Aurora-Client, Aurora-Manager and Aurora-AP, have been outlined. Overall, the *broker mode* of the virtualization agent is an slice orchestration tool designed to offer flexibility to the tenant without directly interfering with the operation of virtualization plane components. However, how virtualization is achieved by the virtualization plane is covered in the following Chapter 5.

Chapter 5

Application, Integration and Deployment of Aurora

The previous chapter covered the implementation architecture of a generic multi-perspective wireless virtualization framework called Aurora. The emphasis was made on the *flexibility* and the *orchestration* capabilities of framework. In this chapter, the actual *application* of the Aurora to assist the implementation of wireless virtualization and software-defined infrastructure is detailed. Overall, Aurora represents a "virtualization operating system" implemented as a service platform that attempts to offer flexibility while providing useful tools for the development and implementation of wireless virtualization. Thus, the architecture will be validated by implementation example and practical application scenarios. First, this chapter explains how datapath isolation and basic 802.11 access point virtualization can be achieved through Aurora-AP. Then, the integration of Aurora with the SAVI research testbed and OpenStack is presented. In order to demonstrate the flexibility of a virtual wireless infrastructure created through Aurora, the support of various example virtualization projects is shown. At last, extensions of Aurora-Agent to include hypervisor mode traffic shaping, SDRs and partial protocol-based virtualization are discussed.

5.1 Datapath Virtualization in Aurora-AP

Overlay flow-based virtualization is the only perspective currently implemented in Aurora-AP. Recall from Section 3.1 that flow-based perspectives aim at allowing tenants to customize the servicing of their traffic without affecting each other. Also recall that Aurora-AP, as the local virtualization agent, does not actually virtualize the physical resource nodes directly. In other words, Aurora-AP is *not a new wireless virtualization technology*. Instead, it must be considered a *meta*-framework and powerful *toolbox* for wireless virtualization that leaves as much as possible the virtualization process itself modular and openly controllable by the tenants. However, how is flow-based virtualization actually achieved by the tenants using Aurora? How do the different abstraction modules and resource component plug-ins come into play in wireless virtualization?

One of the original challenges of a generic framework for wireless virtualization is how to support a large variety of virtualization techniques and perspectives on the same infrastructure to satisfy different tenants. The solution provided by Aurora is that the tenant is given freedom to use the various resource components provided by Aurora-AP to build their own *ap-slice*. In some sense, the flow-based modules of Aurora-AP play a similar role as a VM manager except that it is not the operating system firmware that is being virtualized. Instead, only selected *networking* and *wireless functionalities* are virtualized. On top of this, Aurora-AP gives the freedom to tenants to customize the content of their *ap-slice*, which has no equivalence in other frameworks. That being said, Aurora-AP does not make it an obligation for the tenants (*scalable complexity*, see Subsection 3.2.2). The tenants have the freedom to customize their slice or simply use a predefined virtualization package.

In a typical enterprise-grade 802.11 AP, the concept of multiple virtual access points (VAPs) [33] on the same physical AP is already widely applied. Each VAP can have its own 802.11 BSS, authentication methods and connectivity to a VLAN. The VAP is often used to manage and isolate access policies of wireless clients. Aurora-AP uses these existing concepts and functionalities as the basis for further extensions. In Aurora-AP, traffic from each tenant go through different components defined by configuring the *ap-slice*. The general rule identified for wireless APs is that the data traffic must ultimately go through both the radio interface and the backbone network interface (Ethernet or optical). Thus, the *path between* the radio interface and the network interface can be subject to various type of customization by different technologies. Aurora-AP *enhances* the VAP by providing additional components to build and customize these traffic paths, fully in line with the concept of flow-based virtualization. Ultimately, the isolation of the traffic between the different *ap-slice* depends on the implementation technology of each data path component and the interaction between them.

The physical hardware supporting the first version of Aurora-AP is the *alix3d2* board [54]. It has an embedded 500 MHz AMD Geode LX800 processor with 256 MB DRAM and Compact Flash card slot. Its computing power and performance are slightly superior than most wireless router commercially available. It has two miniPCI radio card slots, one 10/100 Ethernet port, one UART serial port and one USB 2.0 port. The radio cards are based on 802.11b/g Atheros chipset. The wireless firmware is a customized OpenWrt [59] based on Linux kernel 3.2 loaded on a 4GB Compact Flash card (although the image itself is well under than 100MB). The OpenWrt firmware has Python and all resource component technologies and software installed. The architecture of typical *ap-slice* instances on the *alix3d2*-based AP resource node is shown in Figure 5.1.

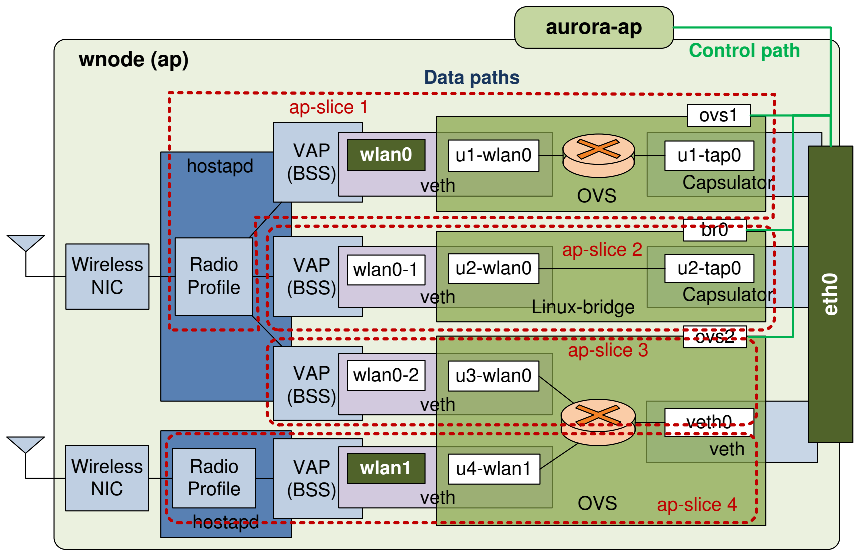

Fig. 5.1: Flexible Datapath Configuration of AP-Slice on AP Resource Node

5.1.1 Wireless and Radio Interface Technologies

OpenWrt uses the Linux daemon *hostapd*, which has the ability to turn any radio interface card into a 802.11 access point. The *hostapd* daemon supports VAP functionalities and binds each BSS to a separate Linux interface, as shown in Figure 5.1. Uplink traffic from wireless clients connected to different BSS are separated inside *hostapd* since they have different ESSID and BSSID (MAC). The prime interfaces (*wlan0*, *wlan1*, etc.) are linked with the first BSS created on a radio card. All subsequent BSS are given a virtual interface (i.e. *wlan0-1*, etc.). One limitation with vanilla *hostapd* is that it does not support dynamic addition and removal of BSS. In other words, whenever a BSS is added or removed, the entire radio interface must be brought down and restarted. For the implementation of Aurora, the vanilla *hostapd* has been modified to support dynamic BSS creation and deletion through the *hostapd_cli* interface with Aurora-AP. Another limitation with *hostapd* is that the radio card is deactivated once the prime BSS is removed. This limitation can be removed by changing the way the radio card is initialized by *hostapd*. Finally, there is a limitation that all BSS must share the same radio configuration profile. This issue is resolved by using technologies that support partial protocol-based virtualization, as will be discussed in Subsection 5.4.2. In *alix3d2* APs, up to two radio cards are supported. These radio cards are allocated as separate resources running separate *hostapd* instances. The naming of the virtual radio interfaces is automatically allocated by Aurora-AP.

5.1.2 Network Interface Technologies

There is only a single Ethernet interface on the *alix3d2*-based resource node. Thus, different tenants must be isolated from each other and from the control path of Aurora while sharing the same physical interface. Virtual network interfaces are used to provide such functionalities. Virtual network interfaces can also be extensively used by tenants as checkpoints to apply various traffic policies within their slice. Thus, Aurora-AP allows the tenants to select the type of interfaces they want to use in their *ap-slice*. Of course, different interface technologies have different isolation properties and restrictions.

- For instance, modified Capsulator interfaces in Aurora-AP isolate traffic by encapsulating them and identifying them by a 2-tuple (the tunnel ID and the destination IP). As such, they are suitable to be attached on the physical Ethernet interface to separate the traffic of each tenant and provide tunneling functionality at the same time. With tunneling, a single Ethernet interface is sufficient for both the control and data paths.

- In the case of *veth*, there are no implicit isolation mechanisms. Thus, Aurora-AP must make sure that the tenants do not create ”leaking” *ap-slice* instances by mistake. Typically, *veth* interfaces are mandatory on radio interfaces when using OVS. This is because there is a conflict between the operation of OVS and that of a radio interface, as mentioned in Subsection 4.1.4. An indirect interface technology such as *veth* is therefore necessary. When used like this, there are no isolation issues because wireless traffic are already isolated by *hostapd* on the radio interface. On the other hand, *veth* cannot be attached directly to the Ethernet interface since it does not provide any filtering or isolation capabilities. An Aurora-controlled OVS must be used to isolate the traffic among tenants, as shown with *ap-slice* 3 and 4 on Figure 5.1. The OVS also prevents any Aurora control messages from being seen by the tenants by blocking them, providing control and data path isolation (see Section 5.4).

The naming of the virtual network interfaces are partially given by the tenants. However, Aurora-AP appends a prefix identifier based on the tenant name or ID to the actual name of the interface, forming its own isolated *namespace*. This allows multiple tenants to name their virtual interfaces without worrying about using the same name as other tenants. Of course, one question that arises is whether the tenant want to customize these interfaces. The answer is no for *service-level tenants* but yes for researchers and infrastructure-level tenants. Aurora attempts to support as much as possible different virtualization architectures as a platform and allow for further innovation. Thus, using these virtual interfaces, the tenants can actually build their slice the way they want within reasonable flexibility without the need to virtualize the entire operating system firmware. For instance, if a tenant wants to process 802.11 MAC remotely, they can use Capsulator to tunnel raw 802.11 MAC frames of their *ap-slice* to a VM in the cloud, as done in CloudMAC [31], while another tenant processes its frames locally, on the *same physical AP*. Future resource component plug-in extensions, such as wireless monitoring interfaces, will be developed to allow raw 802.11 frames to be captured in order to make this possible.

5.1.3 Bridging and OpenFlow Support

The most sophisticated and flexible customization of the data paths occurs through the bridging technologies managed through Aurora-AP. The usage of OVS is suggested to tenants, as it is an OpenFlow and SDN-enabled virtual switch. Different OpenFlow applications for wireless networks can be enabled at the AP with OVS, ranging from firewall and gateway services to mobility management. In general, bridges can also be used as a filter or traffic shaper for tenants to manage the client traffic inside their *ap-slice* instances. In Aurora-AP, separate instances of OVS data path can be created for each tenant, as shown in Figure 5.1. Each instance of OVS data path can then be directly connected to an OpenFlow controller in the tenant network through a bridge interface (*ovs1*, *ovs2*, etc.) on the physical Ethernet interface (control path). The isolation of the control path is managed at the network-wide level using OpenFlow. Nevertheless, Aurora-AP can also use OVS as a *default* bridge in hypervisor mode, allowing more than one tenant to share the same virtual or physical interface (i.e. *ap-slice* 3 and 4 in Figure 5.1). In such a case, OVS is connected to a FlowVisor controller inside the SAVI testbed. Then, each tenant's flowspace is redirected to its corresponding tenant-owned guest controller. The Linux *brctl* bridge is an alternative simpler bridge that can be used if OpenFlow functions are not required. However, unlike with OVS, Aurora-AP does not currently support sharing a single *brctl* instance with multiple tenants. In general, only virtual interfaces within the namespace of the *ap-slice* owned by a tenant can be added to the bridge created by the tenant. The exception to this rule is the default OVS bridge, which contains a common interface (*veth0*) shared using OpenFlow. Interfaces created by tenants choosing the default OVS as bridging option are dynamically added to the default OVS. The OpenFlow control of the default OVS can be delegated to a guest controller through FlowVisor. Currently, the default OVS feature is not fully implemented as it requires full integration with the SAVI networking services. It will be completed as a future extension to the OVS plug-in (see Section 5.4).

5.1.4 Inter-slice Isolation Mechanisms

Basic datapath isolation is achieved using the technologies described in previous subsections. However, it does not provide rate limiting mechanisms among slices. Inter-slice traffic shaping modules to throttle the throughput of different tenants are functional but not yet integrated inside Aurora-AP. They will be covered in Subsection 5.4.1. So far, the *ap-slice* isolation discussed is purely implemented on the AP resource node, in a per node basis. However, such localized isolation is not sufficient and must be extended at the network-wide level (integration with tenant virtual network). Such isolation cannot be provided by Aurora-AP and must be handled by wireless network management functions or SDN frameworks inside SAVI/OpenStack. These functions are outside the scope of this thesis and are discussed in Section 5.5. In terms of wireless resource isolation, the technologies interfaced with Aurora so far do not support *integrated wireless virtualization* nor protocol-based or spectrum-based approaches. As such, there are no inter-slice wireless resource scheduling mechanisms coupled with Aurora-AP.

As discussed above, the current version of Aurora-AP is not (yet) a magical swiss knife of wireless virtualization nor provides a comprehensive list of functionalities. Only a limited set of tools are currently available to demonstrate the core ideas of the framework. However, the modular software design of Aurora makes extensions simple to integrate. Some of these extensions, particularly the traffic shaping module and the support for SDR resources, are briefly explored in Section 5.4.

5.2 OpenStack and SAVI Integration

As explained in Section 4.1, Aurora is designed as a component service to OpenStack. Its implementation is achieved as a part of the SAVI testbed. The basic overview of the SAVI testbed is given in Subsection 2.2.2. In this section, the detailed SAVI testbed setup at McGill is presented in order to discuss the deployment of Aurora in the physical testbed.

5.2.1 SAVI McGill Edge Node Testbed Overview

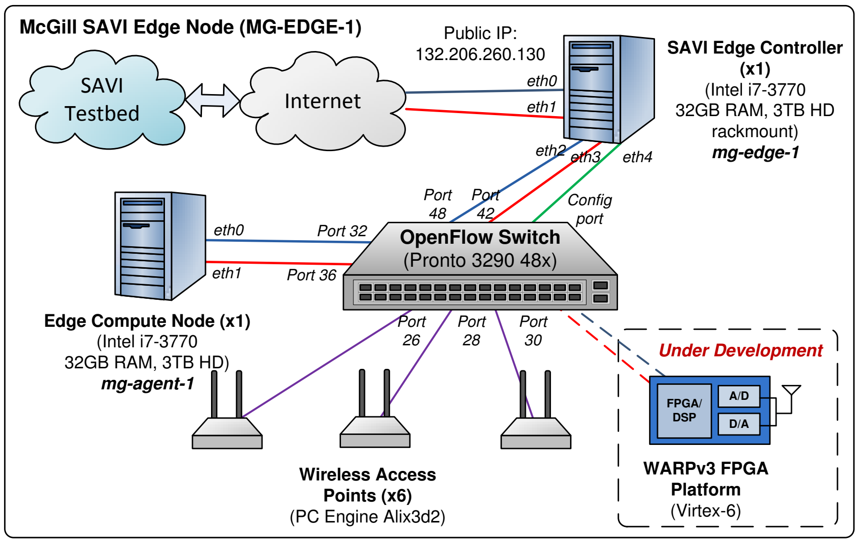

Fig. 5.2: Physical Hardware Setup for the McGill SAVI Edge Node

First, the SAVI testbed at McGill is classified as a *SAVI edge node* with OpenStack region name *MG-EDGE-1*. The edge nodes are designed to be miniature datacenters with specialized non-computing resources. However, the McGill edge node is extremely small in terms of computing power (a single server). It is more focused on the development of virtualized wireless nodes for the SAVI testbed. The McGill edge node is currently composed of one *edge controller* server, a single *compute node* server, one OpenFlow-enabled physical switch and up to six wireless access points, as shown in Figure 5.2. The addition of FPGA platforms used as SDR is currently under development. The McGill edge node is connected to the other edge and core nodes around Canada by Internet. Edge nodes within Ontario are connected by a dedicated optical backbone network. The edge controller hosts the SAVI and OpenStack manager, which includes a Ryu OpenFlow controller and FlowVisor, and the edge node MySQL databases. The OpenFlow-enabled Pronto switch is the central switch connecting all the resources available on the edge node. All VMs allocated inside the MG-EDGE-1 region are instantiated on the compute node. Resources, such as compute nodes, typically use two Ethernet interfaces to connect to the OpenFlow switch. One interface is used for the control path between the edge controller and the compute node while the other is used for the data path. In the case of the wireless access points, only a single Ethernet interface is available for both control and data path (in-band). The isolation of the control and data paths between tenants and the edge controller is achieved through policies set by OpenStack Neutron with its OpenFlow controller. The addition of a second Ethernet interface on *alix3d2* APs using an Ethernet-to-USB adapter has been considered. However, whereas such out-of-band control setup simplifies the isolation mechanism, it is less practical to deploy on wireless nodes, unlike on server nodes. The authentication of users is performed by a dedicated testbed-wide authentication server running OpenStack Keystone. VMs inside the SAVI testbed are reachable from the public Internet if a public *floating IP* is attached to its virtual interface using Neutron.

5.2.2 Interaction and Integration between Aurora, SAVI and OpenStack

The Aurora service components are relatively independent from SAVI and other OpenStack services. Of course, there are different points of integration between the Aurora service components and SAVI/OpenStack in order to provide a unified infrastructure-wide end user experience. These points of integration are well delimited and are briefly summarized as follows:

1. **Tenant and user authentication**: In Aurora and OpenStack, a tenant is not

exactly an individual. Instead, users are individuals belonging to one or more tenant projects. All commands issued by users through Aurora-Client must be validated using a token-based authentication system managed by the OpenStack Keystone service. Aurora-Client is co-located with other OpenStack clients on user access stations. Aurora-related policies must be integrated inside Keystone by configuring authentication *middleware* components [63]. The authentication component is used to identify the user and verify what capabilities that user has with the Aurora service components within a tenant project. These information are then passed to Aurora-Manager, which is co-located with all other service managers (Nova, Swift, etc.) on the SAVI edge controller.

2. **Resource database**: The information about the state of wireless resources and their relationship is stored in the MySQL database on the edge controller, along with other OpenStack and SAVI services. In such a way, the Aurora resource database can allow other services to access information about wireless resources. For instance, some Aurora functionalities can be incorporated into other SAVI services such as Whale, the network topology service.

3. **Network connectivity**: The most important aspect of the integration between Aurora and SAVI is the network connectivity between the wireless resource nodes and the rest of the testbed infrastructure. The physical APs are connected to the SAVI edge node switch through Ethernet, as shown in Figure 5.2. Network virtualization and network-as-a-service (NaaS) inside SAVI are managed by Quantum/Neutron mainly through OVS and OpenFlow-based Neutron plug-ins. Aurora relies on Neutron to connect the wireless slices on physical nodes to virtual networks owned by their corresponding tenant. New OpenFlow rules are added whenever a new virtual wireless network (*wnet*) is instantiated. Wireless-aware OpenFlow controllers must be deployed to handle wireless clients. Even though wireless network functionalities are outside the scope of this thesis, some network connectivity mechanisms are discussed as extensions in Section 5.5.

The projected placement of Aurora components with respect to SAVI and OpenStack components is shown in Figure 5.3. In order to experiment with the proposed framework, a proof-of-concept Aurora framework is implemented *inside* a single SAVI tenant project, as will be presented in the following Subsection 5.2.3. The full integration of Aurora on the same level as the other services (Nova, Keystone, etc.) is expected in future releases of the SAVI testbed.

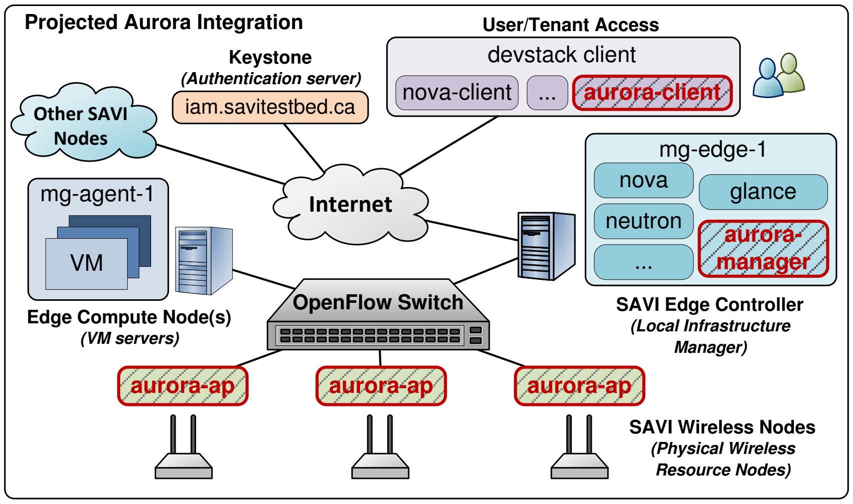

Fig. 5.3: Projected Integration of Aurora Service Components in SAVI/Openstack

5.2.3 Prototype Deployment of Aurora Inside SAVI McGill Edge Node

The SAVI testbed has the advantage of providing a full suite of virtual infrastructure-as-a-service on demand, including VMs and virtual networks but missing wireless resources (and thus one of the motivation of Aurora). In order to test and debug the Aurora software platform during its development, a prototype of Aurora has been implemented within the SAVI testbed. As opposed to the real integration of Aurora within SAVI as a first class service, the components of the prototype are running inside a slice of the SAVI testbed and not as part of the SAVI platform itself. As such, the prototype is isolated from the other SAVI services and cannot interact directly with them. The prototype represents an important milestone before the full support of wireless functionalities in SAVI.

The prototype is implemented in a SAVI tenant project with three networks: *ap-net* (10.5.255.0/24), *mcgill-net* (10.5.8.0/24) and *mcgill-net2* (10.5.254.0/24). These networks are logically isolated from each other by different subnet addresses. They are used to *emulate* the fact that the infrastructure network with the control path (*ap-net*) is isolated from the tenant networks (*mcgill-net* and *mcgill-net2*). The physical AP nodes are bridged

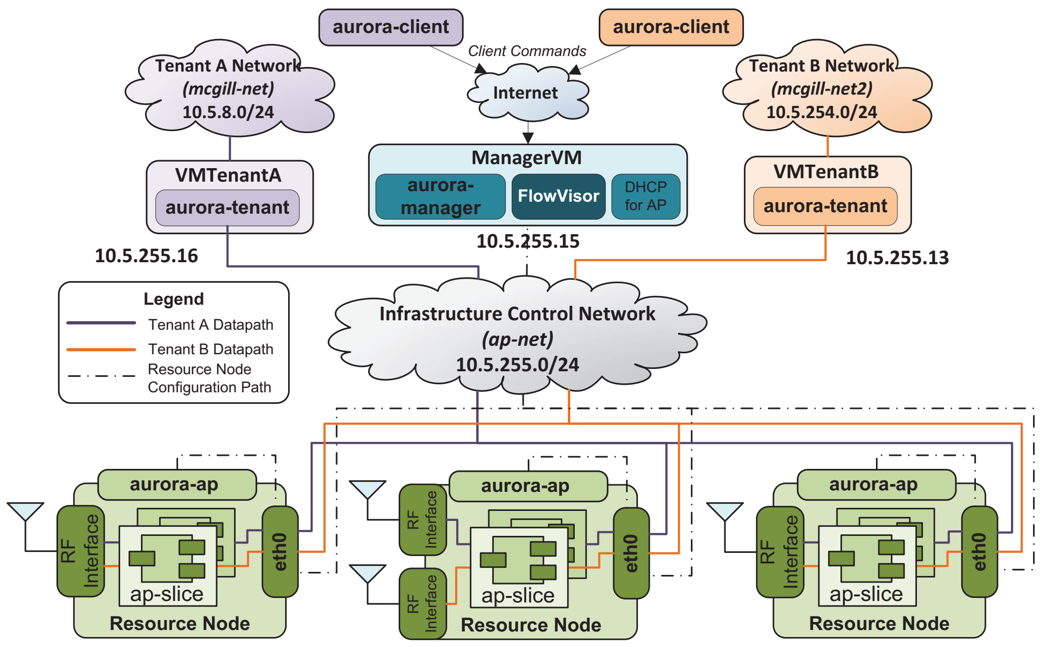

Fig. 5.4: Aurora Prototype Framework Inside a SAVI Testbed Slice

to *ap-net* by modifying OpenFlow firewall rules on the physical port of the switch on which they are connected to. Three VMs are created using Nova: *ManagerVM*, *VMTenantA* and *VMTenantB*. ManagerVM, connected to *ap-net* as 10.5.255.15, hosts the Aurora-Manager along with an instance of FlowVisor. It tries to emulate the SAVI edge controller minus the other SAVI services. The two other VMs and their subnets are emulating two tenants and their networks. The tenant VMs are both connected to *ap-net* (10.5.255.16 and 10.5.255.13) and its corresponding *mcgill-net* and *mcgill-net2*, as shown in Figure 5.4. Of course, there is no traffic isolation between the slices at the network level because all APs are connected to the same subnet (*ap-net*). As such, tunneling is used on tenant datapaths inside the prototype, as will be detailed in the following subsection.

In the prototype setup, Aurora-Client is installed on PC workstations outside the SAVI testbed. A floating IP is assigned to ManagerVM and security group rules (i.e, firewall rules in SAVI) are added, such as port 80 for a HTTP server to receive client commands. Since the Keystone authentication components are not implemented for this prototype, tenant identities are taken at face value by the prototype Aurora-Manager. This basic security setup is sufficient for the prototype since the user still has to obtain the token from Keystone (iam.savitestbed.ca). There are simply no *authentication policies* defined for the prototype (i.e, no customized per-tenant and per-user access to Aurora functions).

Finally, the mere fact that Aurora is actually developed with the help of *other existing SAVI and OpenStack services* reinforces one of the reasons why Aurora was developed in the first place: to provide a powerful framework and testbed platform to facilitate the development of other future frameworks. The ideal vision is that Aurora will join the other SAVI and OpenStack services to accelerate the development of new services, just as the existing services did for Aurora. The next section gives some examples of flow-based wireless virtualization technologies deployed using the Aurora prototype.

5.3 Example Applications of Aurora

As mentioned previously, Aurora is not considered as a new wireless virtualization technology by itself. Instead, it integrates other virtualization technologies and automates slice creation with these technologies. In order to demonstrate the flexibility of the framework, this section briefly presents an implementation of OpenFlow Wireless, a flow-based virtualization architecture from [7], using existing components inside the Aurora prototype framework. Then, in order to showcase the steps necessary to integrate *new* technology components not present in the Aurora prototype, a walk-through on the hypothetical implementation of another flow-based virtualization architecture CloudMAC from [31] is presented. The common point between these two architectures is that they both use OpenFlow, albeit not in the same way. However, the Aurora framework itself is not dependent on OpenFlow. Other SDN technologies and frameworks can also be integrated as resource component plug-ins.

5.3.1 OpenFlow Wireless in Aurora

OpenFlow Wireless was briefly surveyed in Subsection 2.4.2. OpenFlow Wireless essentially allows traffic on wireless access points and basestations to be controlled through SDN by installing an OpenFlow-enabled software switch on them (OVS). The prototype of Aurora simplifies the process of setting up such an architecture by integrating one by one the various technologies that OpenFlow Wireless depends on. This is directly reflected through the first few abstraction modules and technology plug-ins implemented in Aurora, notably Capsulator and OVS. Using such a modular framework, it is possible to construct different wireless virtualization architectures on different slices by making different plug-ins to interface with different component technologies. The setup of OpenFlow Wireless inside the Aurora prototype is shown in Figure 5.5.

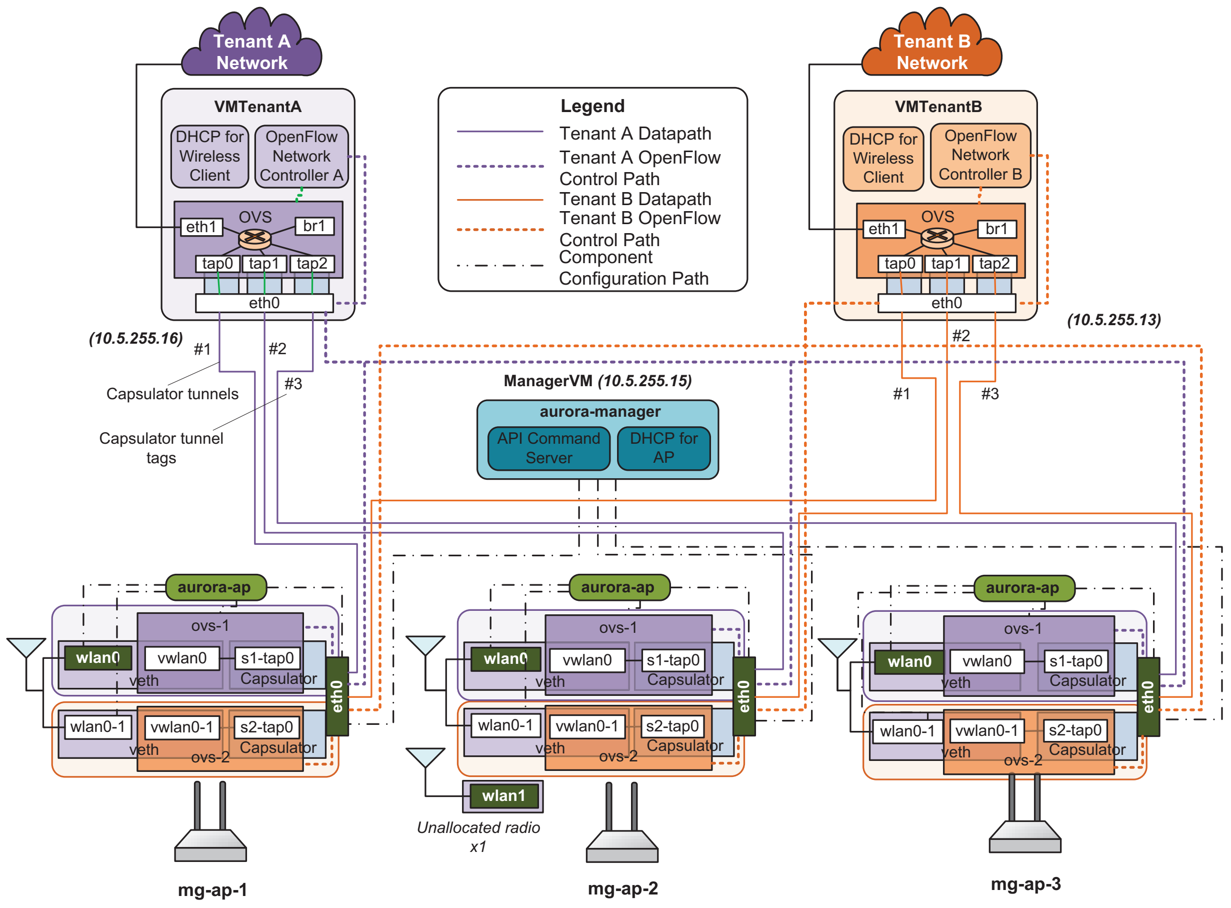

Fig. 5.5: OpenFlow Wireless Deployed Under Aurora Prototype Framework

In the prototype deployment of OpenFlow Wireless using Aurora, tenant A and tenant B each can use their own OpenFlow controller to act upon their own instance of the software switch, as described in [7]. The setup required for OpenFlow Wireless is minimalistic: each *ap-slice* has one Capsulator *VirtualInterfaces* component, one OpenWrt *VirtualWiFi* component and one OVS *VirtualBridges* component. The blueprint of this example deployment is included in Appendix A.6. In this example, the OVS component of each slice is directly connected to the tenant controller respectively located at 10.5.255.16 and 10.5.255.13, without going through FlowVisor as in [7]. This is because in this particular setup, the ports of the VirtualBridges component of each slice are non-overlapping. As such, FlowVisor is not necessary. On the other hand, if the ports are shared, the OVS component can use the default OVS mode described in Subsection 5.1.3 to point to *ManagerVM* on 10.5.255.15. The *VMTenantA* and *VMTenantB* each run a Ryu-based OpenFlow controller for video streaming handover, as in [7]. Capsulator tunnel endpoints are also running in the tenant VMs to de-capsulate the datapath traffic coming from *ap-slice* instances, allowing it to be bridged to its tenant network.

Clearly, the deployment of OpenFlow Wireless inside Aurora is not an exact replica of the original setup in [7]. For instance, there is no SNMP in the example deployment. Instead, the configuration of the radio is performed through the Aurora VirtualWiFi module. In addition, the prototype Aurora-AP actually *extends* the OpenFlow Wireless architecture by binding each *ap-slice* to a VAP. This demonstrates the flexibility of Aurora-AP at abstracting the interfaces between different components, allowing the OpenFlow Wireless architecture to easily integrate into various other configurations and technologies. The *same* modules and plug-ins that OpenFlow Wireless require can be reused in other scenarios and virtualization architectures, assuming the software and hardware requirements of running them are met (i.e, Linux-based operating system with OVS support). In any case, the full configuration of an *ap-slice* instance is consolidated into a single blueprint file. The tenant no longer needs to run its own custom scripts or manually configure the devices. By unifying the different components under a common interface layer, the blueprint allows potentially complex configurations to be easily validated and deployed over a virtualized infrastructure. Of course, new components and new architectures most likely will at the very least require new plug-ins and in many cases, new abstraction modules. In other words, the Aurora framework might be convenient in the long run but certain tenants (i.e, researchers and developers) must be supportive in its earlier stages, where most components and modules are still missing. However, why would any tenant want to extend the Aurora framework just in order to deploy their own wireless virtualization architecture? The following subsection gives an example scenario to illustrate why and how a tenant should write plug-ins and modules for Aurora.

5.3.2 CloudMAC in Aurora

Aurora is mainly based on open-source implementations of various technologies and is itself open to extensions developed by certain tenants. The creation of new plug-ins and modules inside Aurora is relatively simple due to the way Aurora is modularized and does not significantly increase development overhead compared to writing a customized automation script. The advantage of running virtualization projects under Aurora is the fact that other pre-written components are available for maximum technology reuse. Writing new modules and plug-ins is encouraged as it is a more *organized* way of implementing new virtualization technologies. At the same time, these new components are automatically standardized under a common framework (i.e, Aurora), in turn enabling other tenants to reuse them to build their own slice and maintaining a good ecosystem for research and development. This is not to mention that without a unified blueprint, it is very difficult to share the same wireless infrastructure among multiple tenants with different technologies that might interfere or conflict with each other. The process of creating Aurora modules and plug-ins can prevent many of the architecture conflicts right from the earlier stages of their development. As an example to demonstrate how new components are integrated into Aurora, the hypothetical deployment of CloudMAC [31] is outlined as follows.

CloudMAC is briefly surveyed in Subsection 2.4.2 and illustrated in Figure 2.5. In short summary, CloudMAC directly sends 802.11 MAC frames from the access point to a virtual machine running inside the cloud using a tunnelling protocol (Capsulator). CloudMAC requires a *monitoring* interface that can capture and redirect raw 802.11 frames directly from the wireless interface. Such an interface can be integrated as a plug-in to the *VirtualInterface* module that attaches itself to a *custom* radio interface integrated as a plug-in to *VirtualWiFi*. The custom radio interface is needed because CloudMAC requires time-critical functions to be handled locally [31]. However, the current prototype implementation does not have plug-ins for these modules nor the actual implementation of these plug-ins. In order to integrate these components, relatively simple (compared to a scenario without Aurora) steps are outlined as follows:

1. **Implementation of technologies** (or *resource components* in Aurora terminology): First, the implementation of the monitoring interfaces and custom wireless interfaces with local handling of time-critical functions is needed. This step is the same whether or not the Aurora framework is present and can be developed independently from Au-

rora. Even though there are no enforcement by Aurora per se, it is still suggested to *modularize* these implementations to match with the modularity of the Aurora framework in order to maximize the future reuse of the components by other tenants. Additional concerns such as the support for multiple virtual instances of these interfaces on a single physical interface and the isolation of these virtual interfaces from each other are suggested but not required. Of course, if these concerns are not considered by the implementation, Aurora will simply allocate resources using these components *as is*, with *limited sharing* among tenants.

2. **Making plug-ins for Aurora**: This is a new step introduced if these components are to be integrated into Aurora. The developer tenant must write wrappers to interface the Aurora-Agent and Aurora-Manager with these new components. The equivalent steps taken when Aurora is not used would be the creation of automation scripts to facilitate a demo setup of the architecture. The development of wrappers, which includes full component APIs, is generally a more modularized and organized method compared to the composition of ad hoc scripts.

3. **Testing plug-ins for Aurora**: Along with the previous step, this is an additional step taken when integrating new components into Aurora. In order to validate the implementations as well as the new plug-ins and modules, native testing (i.e. with physical access point hardware) and debugging of the Aurora framework itself are required. Such a testbed can be assembled using resources on the SAVI testbed within a *slice* of the infrastructure, in the *same* way Aurora prototype is currently deployed.

4. **Deployment of architecture**: At last, the custom wireless virtualization architecture must be deployed over the infrastructure. In the case without Aurora, it is usually impossible to run multiple wireless virtualization architectures at the same time over the same infrastructure. With Aurora, the resources and components connection topology is fully defined in a centralized blueprint file, facilitating its deployment and any changes to the architecture. At the same time, some components such as Capsulator already exist as Aurora plug-ins. These components can be directly used by the blueprint file without much setup overhead by the tenant. The new components introduced by the CloudMAC tenant can benefit other tenants in a similar way.

As mentioned in step 4, once CloudMAC components are integrated into SAVI, they can be reused by other tenants in various different ways no longer limited by the original CloudMAC architecture. For instance, CloudMAC can be deployed alongside other wireless virtualization architectures over the same infrastructure. In such a scenario, a monitor interface can be allocated to a base tenant, similar to the way BSS are allocated inside the Aurora prototype (see Subsection 4.1.3). On the other hand, guest tenants would not be directly supported on such an interface, unless the monitor interface is modified to distinguish 802.11 MAC frames between slices (additional implementation concerns suggested). This is because a simple monitor interface is otherwise able to capture all traffic on the radio interface. However, for CloudMAC, nothing prevents other tenants from connecting to the base tenant's VMs to use the AP resources. Of course, this is still an idealized narrative of a hypothetical scenario in which everything falls perfectly in place within the Aurora framework. Nevertheless, this extrapolation is based on an achievable integration plan for a real wireless virtualization architecture (i.e. CloudMAC).

5.4 Extensions to Aurora Virtualization Agents

The components for Aurora-Agent (more specifically Aurora-AP for 802.11) described in this thesis so far are only sufficient for the prototype to demonstrate the model of the framework. They are far from being sufficient to support a fully featured virtual wireless infrastructure. For instance, there are no explicit traffic shaping mechanisms to limit the bandwidth of each *ap-slice* instance among tenants. The assumption is that traffic shaping is mostly performed at the network-wide level by the SDN framework within SAVI. However, one basic motivation behind Aurora is to provide flexibility. As such, some example extensions to the Aurora-Agent are covered in this section to provide pointers for future Aurora research and development directions.

5.4.1 Traffic Shaping and Scheduling for Aurora-AP

For *inter-slice* scheduling *across tenants* (hypervisor mode, see Subsection 3.3.2), the steps to guarantee different service levels for different tenants are quite complex. In fact, both downlink *and* uplink traffic are difficult to control on the access point with the current level of implementation. Ideally, the downlink traffic scheduling must be joined with traffic shaping in the network infrastructure or at the source of the traffic. Otherwise, there is no guarantee for the tenant traffic to arrive at the AP in the first place. However, since the link capacity is *asymmetric* on the AP (Ethernet capacity is typically higher than basic 802.11 rates), some *throttling* mechanisms are still necessary to preserve bandwidth guarantee between tenants in the case of over-capacity. In such a case, traffic must be blocked and dropped at the Ethernet interface. Nevertheless, such a basic throttling mechanism can be rendered obsolete by a SDN framework that can control link QoS. Possible extensions for traffic control among slices are already planned by integrating the Linux *tc* (traffic control) plug-in [64], which can be attached on virtual interfaces to modify the queuing discipline (*qdisc*). In fact, in the JSON blueprint file, there are already place holder parameters for defining the throttling threshold of each *ap-slice* (see Appendix A.2).

Actual *scheduling* of the *wireless* downlink traffic is possible on the AP only by manipulating the 802.11 MAC layer, which is not implemented in this thesis. Similarly, on the wireless uplink side, enhancements in the 802.11 MAC layer (i.e. PCF [38]) or modifications on the mobile client (i.e. SplitAP [34]) are required. These types of extensions can be integrated when more advanced radio resources are available, as will be briefly discussed in the following Subsection 5.4.2.

As for *intra-slice* scheduling within the virtual resource slice of a tenant (broker mode), OVS can be managed and controlled by the tenant either directly or by delegation (see Subsection 5.1). The planned flavors of traffic shaping mechanisms will include the *tc* plug-in, which is shared between the Aurora framework (inter-slice) and the tenants (intra-slice) albeit with different permission capabilities. For instance, the configuration of the inter-slice *tc* components will not be available to the tenant but only to the infrastructure administrator or to the framework management services.

5.4.2 Integration of Specialized Radios: WARP and SDR

The proposed Aurora framework is a *multi-perspective* virtualization framework. As such, it is designed to support more than one wireless virtualization perspective. With the *alix3d2* access points resource nodes, only overlay flow-based virtualization is supported since there is no integration with the 802.11 MAC for air-time uplink or downlink scheduling among tenant slices. This limitation reduces the benefits of wireless virtualization as the wireless resources cannot be allocated or isolated by Aurora-Agent. On the other hand, such overlay virtualization is the simplest to deploy on existing technologies. In order to support more advanced wireless virtualization perspectives, technologies such as SDRs must be integrated within the Aurora framework.

Since there are different types of SDRs, the flexibility and modularity offered by the Aurora framework is beneficial to the process of technology inclusion. Reference designs on platforms such as the WARPv3 [55] are not exactly SDRs in the true sense of the term. The WARP board is more a *programmable radio* as opposed to a true SDR, as its MAC and PHY layers cannot be modified once the reference design is loaded into the FPGA. Nevertheless, the flexibility of the FPGA still allows the MAC and PHY layers of a radio to be freely *redesigned* in such a way that software drivers can directly interface with functions previously unavailable through commercial radio cards, making *integrated flow-based* and *partial protocol-based* wireless virtualization feasible. One planned extension to Aurora is the integration of WARP-based SDR nodes. However, Aurora-AP is still applied to WARP resources, as opposed to the hypothetical Aurora-SDR. This is because the current reference design on WARP is limited to 802.11 APs, the main target technology for Aurora-AP. Aurora-SDR will be reserved for full SDR nodes with simultaneous support for multiple protocols, such as GNURadio [45], Sora [46] or OpenRadio [47].

The Aurora-AP agent for WARPv3 nodes is very similar to the Aurora-AP agent for *alix3d2* nodes. *Multiple radio configuration profiles* can simultaneously coexist within the SDR node, effectively achieving partial protocol-based virtualization as described in Subsection 3.1.2. However, the use of a server to host the Aurora agent and the *SDR controller* (if any) is suggested, whether it is a physical or virtual server locally *or* remotely located. A separate server is not mandatory if the SDR node can run a basic Linux operating system on its integrated CPU. The key function of the server is to provide an operating system on which Aurora agents can easily be supported. In the case of WARP nodes, the MAC portion of the 802.11 reference design is divided into the *CPU High*, which handles high-level non time-critical functions such as BSS and authentication, and *CPU Low*, which handles low-level time-critical functions such as ACK responses. The two CPUs are completely decoupled and communicates through a middleman module. Due to this decoupling, the first integration scenario consists of running the *CPU High* on the controller server. In the second scenario, the *CPU High* remains on the embedded microprocessor on the FPGA. The configuration of the reference design with respect to the Aurora framework is shown in Figure 5.6.

The radio configuration profiles define radio parameters that were originally unique per

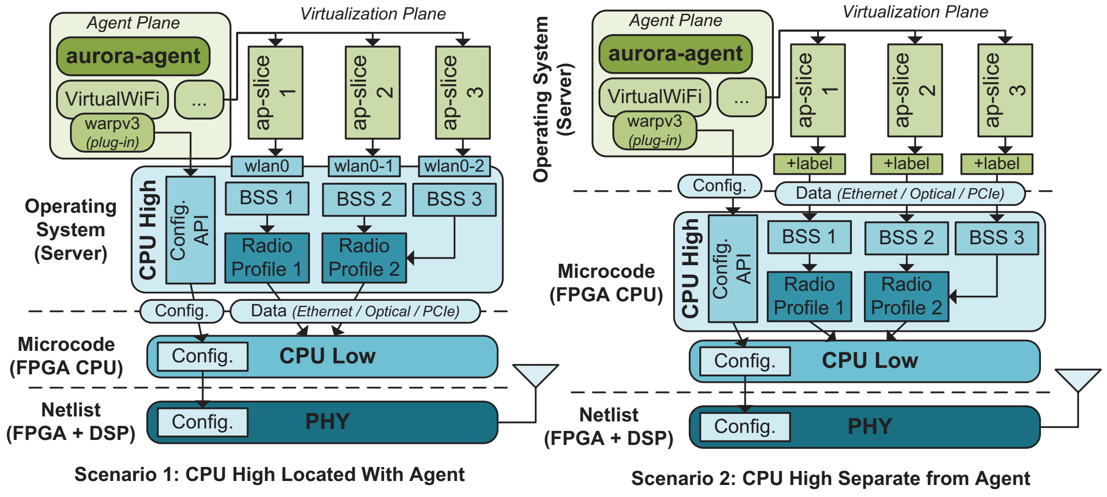

Fig. 5.6: Integration of WARP Resource Nodes Into the Aurora Framework

radio interface for *hostapd* in *alix3d2* nodes. These parameters include transmission power, channel and rate limitations through different modulation and coding schemes (MCS). The reference design can apply different radio configuration profiles for different packets (*per-packet* processing). In fact, there are two scenarios envisioned for integrating the 802.11 AP reference design on WARP. In both cases, dynamic loading and unloading of BSS is supported. In the first implementation scenario, the BSS implemented in *CPU High* can each be *directly* bound to virtual interfaces that act as component plug-ins to the Aurora agent. This setup has the advantage that the de-multiplexing of tenant traffic is reduced, as no additional tagging of traffic is required to bind them to each BSS. On the other hand, in the second scenario, the tenant traffic must be *labelled* or tagged before being sent from the controller to *CPU High* due to the lack of direct binding between the traffic and the BSS. In both cases, Aurora-AP runs on a GPP-based controller external to the FPGA on WARPv3. Moreover, the radio configuration profiles reside inside both the Aurora agent and the access memory of the FPGA. The profiles must be readily accessible with very low latency in order to bind each packet to them in the MAC and PHY layers of the reference design. The extension to Aurora is in the form of a new flavor/plug-in of *VirtualWiFi* virtual radio interface module, referred as *warpv3* (as opposed to *OpenWrt* with *alix3d2* nodes). All other modules such as *VirtualInterfaces* and *VirtualBridges* are the same as the ones for *alix3d2* nodes. However, the basic interactions between BSS and radio interfaces are drastically more flexible due to the removal of restrictions present in *hostapd*.

For full SDR technologies in full protocol-based virtualization, the Aurora-Agent will be significantly distinct from Aurora-AP. For instance, Aurora-SDR might allow one tenant to load their own software protocol stack on the SDR platform while sharing radio resources with another tenant's protocol stack. In that case, Aurora-SDR will interface with a technology component that manages the running protocol instances similar to a hypervisor managing VMs. Unfortunately, WARPv3 cannot yet support such scenarios without significant modifications to the 802.11 reference design. Nevertheless, if these scenarios are supported, the process of integration into Aurora remains relatively unchanged. Thus, the proposed Aurora agent is flexible and modular enough to support a large variety of wireless virtualization technologies, partially due to the decoupling of the agent plane from the virtualization plane in its architecture.

5.5 Extensions to Aurora Network Functions

At last, the prototype framework presented so far in the thesis only covers the virtualization of individual wireless nodes. The fully virtualized wireless infrastructure requires functions at the network level. Even though the Aurora network functions are considered outside the scope of this thesis, this section briefly outlines the flexibility and degrees of freedom provided by Aurora-Agent (Aurora-AP) for network integration.

5.5.1 Connectivity Support for Service-Level Tenants

Recall that there are different types of tenants residing within or on the border of the virtualized infrastructure. In Subsection 4.1.3, service-level tenants simply want their clients to connect to their services through the virtualized wireless infrastructure. In such a scenario, the slices on the virtual nodes are actually owned by an infrastructure tenant. The infrastructure tenant must then provide connectivity to other tenant's virtual networks. Since Aurora will be first integrated into the SAVI testbed, a *default* infrastructure tenant service can be provided by the testbed.

An extension of the Aurora prototype requires the authentication of mobile clients by a specialized authentication and subscription server within SAVI. This server maintains the subscription of a mobile client to a particular service offered by service-level tenants. The DHCP of the client can be performed by this server. Then, using the MAC and IP of the client, an OpenFlow controller can be used as a firewall to restrict the connectivity (IP, ARP, broadcast domain, etc.) of the client only to the services it subscribed to. In such a setup, a client can be subscribed to more than one service-level tenants. The basic outline of this setup is shown in Figure 5.7.

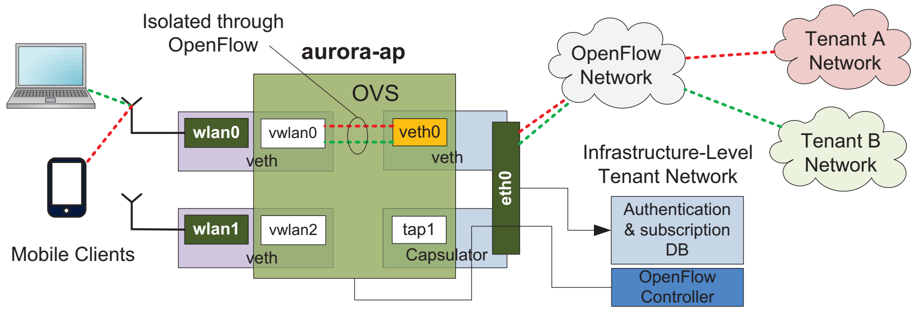

Fig. 5.7: Connectivity Support for Service-Level Tenants by Infrastructure Tenant

An alternative setup would be to instantiate multiple capsulator border interfaces to directly *bridge* client traffic to its corresponding service-level tenant network. Once again, OpenFlow is used to provide isolation and flexibility to the tunneled traffic. However, such a scenario prevents a client from connecting to multiple services at the same time, limiting its application. Nevertheless, a well-designed infrastructure slice can potentially support multiple types of network configurations. By design, Aurora-Agent does not limit the type of network configuration. Standard and common network functions will be included as modules and plug-ins in Aurora-Manager and applied on *wnet* resources, just as technology components were included as plug-ins in Aurora-Agent.

5.5.2 Integration of Legacy Wireless Infrastructure

Recall from Subsection 3.2.1 that legacy infrastructure must be able to integrate within the Aurora framework in order for its deployment to be successful. The deployment of the Aurora framework is equivalent to the deployment of wireless functionalities in the SAVI testbed. As such, the deployment of SAVI and Aurora over university campus wireless access points is one of the first scenarios that should be supported.

In order to minimize the amount of changes required to the existing devices, very limited functionalities are supported over them. For instance, in the case of university APs, the device is owned by the university and cannot be easily upgraded to a custom firmware. This limitation is true to most of the existing infrastructure. However, a VLAN and a

BSS can be obtained from the university. This VLAN can then be routed to a SAVI edge switch. In such a way, the clients can still connect to the SAVI testbed through a dedicated BSS. In such setup, wireless virtualization is clearly not feasible but simple connectivity to service-level tenant networks is still possible. In addition, some of the network-level functionalities can still be implemented to some degree with the help of OpenFlow to reinforce policies between service-level tenants. In such a scenario, there is no virtualization agent. Specialized management modules in Aurora-Manager will be available to handle this scenario.

5.6 Chapter Summary

In this chapter, the application and integration of a prototype Aurora framework was presented in order to demonstrate the benefits and flexibility of the framework. The deployment of Aurora with SAVI is also prototyped within the SAVI testbed itself. The usefulness of a modular and multi-perspective virtualization framework is strengthened by how future extensions can be developed over it. Two example architectures, one already supported (OpenFlow) and one not yet supported (CloudMAC), are shown to provide a walkthrough of deploying wireless virtualization architectures over Aurora. Finally, due to the limited scope of this thesis, some functionalities were left out as future extensions to the framework.

Chapter 6

Conclusion

6.1 Summary

Overall, this thesis is an introductory step in the development of a virtualization framework and platform for the future wireless infrastructure. As discussed in Chapter 1, there are three main essential concepts at work: *virtualization*, *software-defined technologies* and *service-oriented infrastructure*. The first concept, virtualization, is applied to wireless technologies in order to allow efficient allocation and sharing of wireless resources. A survey of different wireless virtualization technologies and architectures was presented in Chapter 2. Testbed architectures and existing general frameworks for infrastructure virtualization were also examined in order to determine the *context* in which wireless virtualization is applied. The resulting analysis indicated that wireless virtualization can be viewed as an important part among other types of virtualization, such as computer virtualization and network virtualization. Wireless virtualization, together with other *virtualization domains*, form the important pillars of *infrastructure virtualization*. Then, wireless virtualization itself is particularly more complex than the other domains. This thesis classifies wireless virtualization into three different perspectives: *flow-based*, *protocol-based* and *spectrum-based*. All three perspectives are deemed necessary because they target different applications and needs. Thus, the framework proposed in Chapter 3 and 4 attempted to support all three and potentially more by being intrinsically modular and evolvable.

The second concept of software-defined principle was explored throughout the survey of different virtualization architectures and related technologies. Software-defined technologies provide a higher flexibility than their hardware-defined counterparts, albeit at the expense of performance. The typical strategy is to *migrate* the loss in performance to less time-critical sections of the system. This mitigation is present in the proposed virtualization framework Aurora, which decouples the different functions into the management plane, the agent plane and the virtualization (technology component) plane in such a way that it does not significantly affect the performance of existing technologies. Overall, software-defined technologies are essential to the implementation of virtualization and service-oriented infrastructure. They remains as part of the basic assumptions on the evolution of wireless virtualization technologies considered within this thesis.

Finally, the concept of service-oriented infrastructure was less explicitly mentioned throughout this thesis. However, it implicitly serves as the foundation of the proposed wireless virtualization framework and software platform Aurora. In order to apply all three concepts, support different virtualization perspectives and different wireless resources (heterogeneous infrastructure), Aurora was designed to follow the service-oriented architecture of the open-source cloud computing platform OpenStack. The entire architecture of Aurora is based on a server-client relationship between the tenant and the infrastructure and among the software components of the infrastructure. This enables the hypothetical scenarios described in Section 1.4. Nevertheless, Aurora does not introduce any new virtualization technologies. At the end, Aurora is just a common *unifying* framework to make wireless virtualization easier to *implement* and *deploy*. It *binds* the different architectures together such that a centralized management and servicing of these architectures become possible.

Overall, this thesis only presents *half* of the picture, mostly focusing on the virtualization of the individual radio resource nodes and the accompanying virtualization agent. However, for a fully-featured wireless infrastructure, the concept of wireless networking is very important. Thus, the management of wireless resources on an infrastructure-wide level, such as the mobility management and dynamic provisioning of resources, is the other half of the picture and can be the subject of further research. In fact, the prototype presented in this thesis can be barely considered the basic skeleton framework required to demonstrate the main ideas behind Aurora. In order to be effective, the framework must be *extended*, as will be discussed in the following section.

6.2 Future Works and Extensions

As previously discussed, there are many shortcomings and missing components in the prototype framework. These future research directions and software development tasks for Aurora are grouped as follows:

1. **Stabilization of Aurora prototype and full SAVI integration**: The current prototype only has the bare essential components that might not be the most optimal implementation. Some minor stabilization and improvements on existing features are required before the actual deployment of Aurora inside the SAVI testbed. The *full* integration of Aurora with SAVI potentially requires some minor changes to both SAVI and Aurora. For instance, the *Aurora REST API server* architecture is not yet defined and is classified as one of the most important features to implement. The REST API server is currently substituted by a simple HTTP server handling JSON messages.

2. **Management functions and modules for Aurora-Manager**: Some basic management modules and extensions to Aurora-Manager are required for a fully functional deployment of Aurora. At the very least, connectivity among wireless nodes and the tenant virtual networks must be automatically maintained. The network connectivity are partially manually configured in the current prototype. The development of OpenFlow controllers and the partial implementation of *wnet* resources are considered the *other half* of the framework.

3. **Advanced scheduling, QoS and isolation of wireless resources**: In order to provide a richer set of functions to manage and control the service parameters of wireless slices, more advanced scheduling and traffic shaping techniques must be integrated within Aurora as extensions, as discussed in Subsection 5.4.1. Some of these functions are wireless technology-specific whereas some functions are tied with the network virtualization framework within SAVI, Neutron. Basic traffic shaping with Linux *tc* has been successful but remains to be integrated within Aurora as a plug-in. More advanced scheduling at the wireless MAC level can also be the subject of integration.

4. **Integration of software-defined radio technologies in Aurora**: The basic in-

tegration of WARPv3 resource nodes was discussed in Subsection 5.4.2. The actual implementation and prototyping of the reference design are left to be done. The predicted integration scenarios can change depending on the engineering decisions that are made during its implementation. The integration of other SDR platforms, preferably full SDRs, should be considered in order to implement full protocol virtualization and spectrum virtualization. More advanced radio technologies such as cognitive radios and distributed antennas are also potential targets for integration with Aurora.

5. **Integration of cellular technologies in Aurora**: Cellular technologies was not extensively mentioned in this thesis due to the lack of infrastructure (i.e. basestations) that can support it within the SAVI testbed. However, the architecture of the cellular network is mappable to the Aurora framework. A basic projection of cellular technologies in Aurora was made in Figure 4.3. For instance, the packet core can be implemented on tenant VMs (Aurora-Tenant) whereas the basestations can be interfaced with a cellular technology-specific Aurora virtualization agent.

6. **Miscellaneous features for improving framework usability**: A large variety of features and functions that can further enhance the service-oriented infrastructure to provide better usability and flexibility remain yet to be implemented. For example, some of these features include more support for IPv6 protocols within the Aurora components. Currently, IPv6 functions are dependent on the implementation technology components but are not exposed through the Aurora blueprints. Other useful features include a monitoring interface (for use in CloudMAC) and additional pre-made virtualization blueprint packages for easier deployment.

Appendix A

Aurora Software Implementation Overview

This appendix provides a very general overview of the software implementation details of Aurora. It is not meant to be the documentation of the code. The actual documentation can be found within the source code itself. In fact, since the Aurora software is in active development at the writing of this thesis, the following overview does not reflect the most recent version of the framework. Instead, this overview aims at providing some insight on how Aurora is structured and what type of functions it can provide. The structures of the different databases are also briefly explained.

A.1 Aurora-Client Commands

The following list describes various commands supported on Aurora-Client. Underneath, these commands will use the REST APIs to communicate with Aurora-Manager (see Subsection 4.2.1). The REST API portion of these commands are not yet implemented in the current version of Aurora. In addition, not all these commands are fully functional. For instance, the resource class *wnet* has no functionalities except an entry in the MySQL database. Therefore, all *wnet*-related commands are in fact place holder functions.

- **aurora ap-list** [--filter <ap tags>] [--i]: This command prints a list of physical *ap* nodes available to the tenant. Typically, an *ap* resource node is available to all tenants (public). However, some resource nodes are private and only available to administrators. The *filter* optional argument takes in a string that describes filter arguments on the tags of the *ap*. Some basic logical expressions are supported, such as: "=" (equal), "!" (not equal), ">" (greater than), "<" (smaller than), "&" (logical and) and "*" (select all). For example, "location=mcgill & number_radio>1" would return all *ap* located in McGill and having more than one radio interface. The optional argument "--i" displays additional *ap* information on the list.

- **aurora ap-show** <ap name/uuid>: This command shows all the detailed status and capabilities information available to the tenant for the specified *ap* by name or by UUID, assuming the resource is visible to the tenant.
- **aurora ap-slice-list** [--filter <ap-slice tags>] [--i]: This command is similar to the *ap-list* command except that it displays the list of all *ap-slice* visible to the tenant. By default, a tenant can only see *ap-slice* it owns. The *filter* optional argument behaves the same as *ap-list* with the addition of *tenant-added tags* (see Subsection 4.1.2). The optional argument "--i" displays additional *ap-slice* information on the list.
- **aurora ap-slice-show** [--i] <ap-slice name/uuid>: This command is similar to the *ap-show* command except that it shows the detailed information for an *ap-slice* specified by name or UUID owned by the tenant. The optional argument "--i" additionally displays the full configuration of the *ap-slice*.
- **aurora ap-slice-create** [--filter <ap-slice tags>] [--file <path>] [--tag <string>]: This command creates an *ap-slice* for the tenant on the specified *ap*. This function can also create an *ap-slice* on each of the *ap* from the optional *filter* argument. If a single *ap* is desired, querying "name=ap_name" is sufficient. If *filter* is not specified, the *ap-slice* is automatically allocated to the first available *ap* (this usage is discouraged). The configuration of the *wslice* is passed as a JSON configuration file whose path is specified using the optional *path* argument. A single configuration file can be used to generate multiple *ap-slice* with variable range of parameters (see Appendix A.2). If a configuration file is not provided, a *default ap-slice* defined by the framework is created instead. The optional *tag* argument allows tenants to add their own tags to identify *ap-slice* they own. This command does not wait until all

the slices are finished building to return (non-blocking). It merely "boots" the slices. Tenants must use *ap-slice-list* or *ap-slice-list* to see the current status of the slice.

- **aurora ap-slice-delete** [--filter <ap-slice tags>]/<ap-slice name/uuid>: This command deletes one or multiple *ap-slice* created by the tenant. The *ap-slice* can also be referred by UUID. A warning message is returned if no filter or name arguments are specified.

- **aurora ap-slice-modify** [--filter <ap-slice tags>] [--file <path>]: This command allows the tenants to modify the configuration of one or multiple *ap-slice* they own and created. The current implementation simply resets the *ap-slice* and applies the new configuration, while keeping the UUID, tenant-added tags and other metadata associated with the existing *ap-slice*. Thus, this is different from *ap-slice-delete* followed by *ap-slice-create*, which resets the *wslice* metadata as well. The optional *filter* argument behaves identically as *ap-slice-create*. The new configuration file is specified through the optional *file* argument.

- **aurora ap-slice-restart** [--filter <ap-slice tags>]: This command restart one or multiple *ap-slice* instances owned by the tenant, specified through the *filter* argument. This command behaves the same as a slice creation command except that it can be applied on existing *ap-slice* that failed during past slice creation or modification. A warning message is returned if no filter or name arguments are specified.

- **aurora ap-slice-add-tag** [--filter <ap-slice tags>] [--tag <string>]: This command allows tenants to add tags to the *ap-slice* they own. One or multiple *ap-slice* can be specified through the optional *filter* command. If there are no *filter* arguments provided, tags are added to all *ap-slice* available to the tenant. Tags are useful tools to group *ap-slice* instances together by attributes defined by the tenant. An *ap-slice* can have an arbitrary number of tenant-added tags. If no *tag* argument is specified, nothing happens (i.e. empty tag).

- **aurora ap-slice-remove-tag** [--filter <ap-slice tags>] [--tag <string>]: This command allows tenants to remove tags from the *ap-slice* they own. One or multiple *ap-slice* can be specified through the optional *filter* command. If there are no *filter* arguments provided, the specified tag is removed from all *ap-slice* available to the tenant. The tag is specified through the optional *tag* argument. The special "*" can be used to remove all the tenant-added tags. If no *tag* argument is specified, nothing happens.

- **aurora wnet-list**: This command is similar to the *ap-slice-list* command except that it displays the list of all *wnet* visible to the tenant. By default, a tenant can only see *wnet* it owns. There are no optional arguments for this command.
- **aurora wnet-show** <wnet name/uuid>: This command is similar to the *ap-slice-show* command except that it shows the detailed information for a *wnet* specified by name or UUID owned by the tenant.
- **aurora wnet-create** [--filter <wslice tags>] [--file <path>] <wnet name>: This command allows the tenant to create a *wnet* instance. The optional *filter* argument specifies the *wslice* (in this implementation only type of *wslice* available is the *ap-slice*) that join the wnet. No *wslice* is added if no filter arguments are specified. Each *wslice/ap-slice* can only be a member of a single *wnet*. The optional *file* argument specifies the path for a *wnet* configuration file. Additional optional *wnet* parameters will be added in the future. The current implementation is only a place-holder for future expansion.
- **aurora wnet-delete** <wnet name/uuid>: This command deletes a *wnet* created by the tenant. The *wnet* must be specified by name or by UUID.
- **aurora wnet-join-subnet** <wnet name/uuid> <subnet name/uuid>: This place-holder command is an integration of Aurora with OpenStack Neutron for a *wnet* to join as part of a *subnet*. This operation is the equivalent of adding all *wslice* under the *wnet* on the ports of the virtual switch represent the subnet. As a consequence, the IP addresses of the *wslice* are allocated through the DHCP present inside the subnet (if applicable). This function is not implemented at the writing of this thesis.
- **aurora wnet-add-wslice** [--filter <wslice tags>] <wnet name/uuid>: This command allows the tenants to add one or multiple *wslice* to a *wnet*. The groups of *wslice* are specified by tags through the optional *filter* argument. If no filter arguments are specified, no *wslice* is added. In addition, if the *wslice* specified is already part of another *wnet*, it is not added to this *wnet*.

- **aurora wnet-remove-wslice** [--filter <wslice tags>] <wnet name/uuid>: This command allows the tenants to remove one or multiple *wslice* from a *wnet*. The groups of *wslice* are specified by tags through the optional *filter* argument. If no filter arguments are specified, no *wslice* is removed. If the *wslice* specified is not part of the *wnet* specified, nothing is removed.

A.2 Aurora JSON Configuration Blueprint Format

The blueprint that the tenants use to define one or multiple Aurora *wslice* instances is a Python dictionary (or equivalent JSON object) in JSON format. There are two main sections to the blueprint of each slice: the general *slice attributes* and the *component configuration*. The slice attributes define some general service parameters on the slice and are used by the hypervisor mode of the virtualization agent to set up inter-slice isolation and service guarantee (currently not fully implemented). The component configuration are grouped by abstraction modules, as shown in Figure A.1 in Subsection 4.1.4. The different components of the blueprint are optional. By default, unspecified parameters have a default value handled at Aurora-Client. If a resource needs to be allocated automatically, the allocation is handled by Aurora-Manager. If crucial information is missing and neither Aurora-Client nor Aurora-Manager can resolve it, a warning message is issued to the tenant.

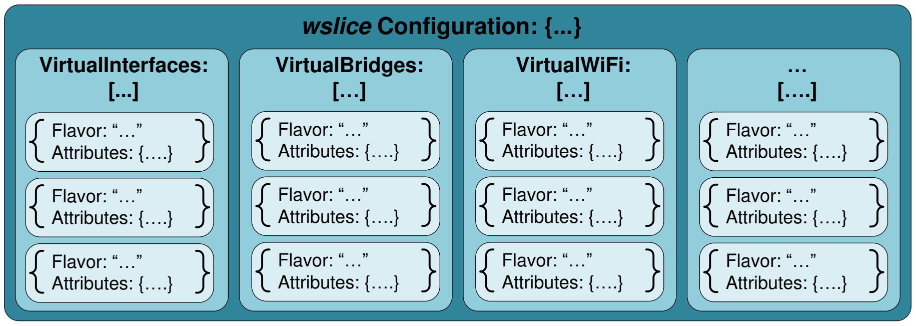

Fig. A.1: JSON Configuration Format for Modules and Plug-ins in a *wslice*

Due to the complexity of the JSON structure, the description of the blueprint file is not in an official JSON schema format. The general conversion rule between Python structures and JSON is that Python dictionary is equivalent to a JSON object and Python list is equivalent to a JSON array. While the blueprint is stored and transported as JSON, it becomes a Python structure when parsed by Aurora. The following sections are entries to the main blueprint dictionary/object. Note that not all attributes are currently implemented. Many entries are left as place holders for future extensions to the framework. Samples of the blueprint file are included in Appendix A.6. The generic format of configuration modules entries for each *wslice* is illustrated in Figure A.1.

1. **id** (Type: string): UUID of the *ap-slice* for identification.
2. **attributes** (Type: dictionary/object): Dictionary of general slice attributes.
    - **nw_downlink_rate** (Type: number): Network downlink rate for the slice. *(currently not implemented)*
    - **nw_uplink_rate** (Type: number): Network uplink rate for the slice. *(currently not implemented)*
    - **wl_downlink_rate** (Type: number): Wireless downlink rate for the wireless node of the slice. *(currently not implemented, need modification to wireless MAC)*
    - **wl_uplink_rate** (Type: number): Aggregated wireless uplink rate for all the clients to wireless node of the slice. *(currently not implemented, need modification to wireless MAC)*
3. **VirtualWiFi** (Type: list/array): List of components from the *VirtualWiFi* abstraction module. This module is used for creating virtual radio interfaces for 802.11. This module contains two types of components: *radio* and *bss*. A *bss* must be attached to a *radio* to function.

    (a) **radio** (Type: dictionary/object): Radio configuration profile for 802.11 AP, mostly derived from *hostapd*.

    - **name** (Type: string): Name of the radio interface used as reference by the tenant. The actual name of the interface is managed by the framework to differentiate between tenants on the same resource node.
    - **channel** (Type: number): 802.11 frequency channel. The available channels are 1 to 14 in 2.4GHz. The availability of 5GHz channels varies depending on the country code.

- **hwmode** (Type: string): 802.11 mode (a/b/g/n/ac/ad). Currently, only 802.11a and 802.11b/g are available on *alix3d2*-based access points.
- **txpower** (Type: number): Radio transmission power in dBm.
- **country** (Type: string): Country code based on ISO 3166-1 alpha-2. Canada is CA and the United States is US. Certain restrictions in frequency channels apply depending on the country code.
- **bss_limit** (Type: number): Number of BSS that can be attached on this radio interface. The minimum value is 1 while the maximum is 4 for *alix3d2*-based access points (can be increased with RF card upgrade).
- **bss_shared** (Type: number): This is the number of BSS that can be used by other tenants on this radio interface. In other words, this parameter allows the tenants to control whether they want to share the radio with other tenants. The number of BSS available to other tenants cannot exceed the maximum number of BSS specified by *bss_limit*.

(b) **bss** (Type: dictionary/object): Basic service set (BSS) configuration profile for 802.11 AP, mostly derived from *hostapd*. Must be attached to a *radio*. This also creates a virtual radio interface.

- **name** (Type: string): Name of the BSS used as reference by the tenant. This name is internal but is sometimes the same as SSID field.
- **radio** (Type: string): Name of radio interface (typically *radio0* or *radio1*) on which the BSS is attached to.
- **ssid** (Type: string): Service set identifier. Maximum 32 characters. They are managed by the framework to prevent conflict between tenants. Thus, it can be different from the *name* field.
- **if_name** (Type: string): Name of the virtual radio interface created by this BSS.
- **macaddr** (Type: string): MAC address of the virtual radio interface. Typically, this field is not available to tenant and is automatically allocated.
- **encryption_type** (Type: string): Encryption type for client association to BSS. Currently available types are *wep-open*, *wep-shared*, *psk* and *psk2*.
- **key** (Type: string): Key (password) for encryption. Note that different encryption types have different key length requirements.

- **auth_server** (Type: string): RADIUS authentication server IP.
- **auth_port** (Type: number): RADIUS authentication server port number.
- **auth_secret** (Type: string): RADIUS authentication server key.
- **acct_server** (Type: string): RADIUS accounting server IP.
- **acct_port** (Type: number): RADIUS accounting server port number.
- **acct_secret** (Type: string): RADIUS accounting server key.
- **nas_id** (Type: string): Optional NAS-Identifier string for RADIUS messages. Currently left as default.

4. **VirtualInterfaces** (Type: list/array): List of components from the *VirtualInterfaces* abstraction module. This module is used for creating virtual network interfaces.

   (a) **capsulator** (Type: dictionary/object): Capsulator over-IP tunneling interface.

   - **name** (Type: string): Name of tunneling border interface.
   - **attach_to** (Type: string): Name of interface on which tunnel is attached to.
   - **forward_to** (Type: string): IP address of tunnel endpoint.
   - **tunnel_tag** (Type: number): Tunnel tag ID.
   - **is_virtual** (Type: Boolean): Create a virtual interface matching with *name* if set to TRUE. Otherwise, makes the existing interface specified *name* the border interface.

   (b) **veth** (Type: dictionary/object): Virtual Ethernet interface.

   - **name** (Type: string): Name of virtual Ethernet interface.
   - **attach_to** (Type: string): Name of interface on which *veth* is attached to.

5. **VirtualBridges** (Type: list/array): List of components from the *VirtualBridges* abstraction module. This module is used for creating virtual bridges.

   (a) **ovs** (Type: dictionary/object): Open vSwitch (OVS) bridge that transforms the bridge into an OpenFlow-enabled virtual switch.

   - **name** (Type: string): Name of bridge interface.

- **interfaces** (Type: list/array): List of string representing interface names attached to this bridge.
- **default_ovs** (Type: Boolean): Flag marking whether this is a separate instance of OVS or the default OVS provided by SAVI. If the default OVS is selected, the default virtual network interface *veth0* is automatically included. By default this flag is set to *False*.
- **bridge_settings** (Type: dictionary/object): Additional OVS settings to the bridge. These attributes are optional and are derived from *ovs-vsctl*.
  - **controller** (Type: string): OpenFlow controller ip in the format "tcp:x.x.x.x".
  - **dpid** (Type: string): OpenFlow datapath identifier for this bridge
  - **fail_mode** (Type: string): Failsafe mode without OpenFlow controller. Use *standalone* for a default controller by OVS when an external controller is not present. Use *secure* to disable the bridge when an external controller is not present.
  - **ip** (Type: string): The IP address of the OVS bridge interface.
- **port_settings** (Type: dictionary/object): Additional port settings to the bridge. *(currently not used)*

(b) **linux_bridge** (Type: dictionary/object): Standard Linux bridge.

- **name** (Type: string): Name of bridge interface.
- **interfaces** (Type: list/array): List of string representing interface names attached to this bridge.
- **bridge_settings** (Type: dictionary/object): Additional Linux-bridge settings. These attributes are optional and are derived from *brctl*.
  - **ageing** (Type: number): Number of seconds a MAC address is kept in the forwaring database after last packet seen.
  - **stp** (Type: string): Activate spanning tree protocol (on/off).
  - **bridge_priority** (Type: number): Priority of bridge in STP. Lowest priority bridge is set as the root STP bridge.
  - **forward_delay** (Type: number): Number of seconds spent by the bridge in listening/learning state before initiating the forwarding.

  - **hello_time** (Type: number): Number of seconds between Hello packets sent by the bridge for STP.
  - **max_age** (Type: number): Time-out if no Hello messages are received from a bridge.
  - **ip** (Type: string): The IP address of the Linux bridge interface.
- **port_settings** (Type: dictionary/object): Additional *brctl* settings to the bridge.
  - **priority** (Type: string): Format is "[port] [priority]" and set a 8-bit value to be used as metric in STP.

A.3 Aurora-Manager MySQL Resource Database

The MySQL resource database on Aurora-Manager mostly contains resources attributes (relatively static) and relationship between resources (dynamic). It is separated into multiple tables. There are three *resource tables*: *ap*, *ap-slice* and *wnet*. In addition, there are two *tag tables* used to attach location tags on *ap* and tenant tags on *ap-slice*. As a side, *primary* keys must be unique for MySQL entries within a given table.

1. **ap**: The *ap* tables contain attribute information of physical access point resources, which are physical resource nodes (*wnode*). Most of these attributes, with the exception of the memory status, do not change during operation.
   - **name** (Primary key): String for name of the *ap*. (Example: *mg-ap-1*)
   - **region**: OpenStack region of the *ap* node. For McGill, it is *MG-EDGE-1*.
   - **firmware**: Firmware for the wireless AP. Currently, only *OpenWrt* is supported.
   - **version**: Place holder for version control of resource node.
   - **supported_protocol**: String specifying which 802.11 protocols are supported. The current *alix3d2* hardware supports "a/b/g".
   - **number_radio**: Number of physical radio cards available on the resource node. Currently 1 or 2.
   - **number_radio_free**: Number of radio cards currently inactive on the resource node.

```
+--------------------+--------------+------+-----+---------+-------+
| Field              | Type         | Null | Key | Default | Extra |
+--------------------+--------------+------+-----+---------+-------+
| name               | varchar(255) | NO   | PRI | NULL    |       |
| region             | varchar(255) | YES  |     | NULL    |       |
| firmware           | varchar(255) | YES  |     | NULL    |       |
| version            | double       | YES  |     | NULL    |       |
| supported_protocol | varchar(255) | YES  |     | NULL    |       |
| number_radio       | int(11)      | NO   |     | NULL    |       |
| number_radio_free  | int(11)      | NO   |     | NULL    |       |
+--------------------+--------------+------+-----+---------+-------+
```

2. **ap_slice**: The *ap_slice* table contains the attributes for the *wslice* instances for 802.11 access points. It mainly contains information about its status and relationship with other entities, notably *tenant* and *wnet*.

   - **ap_slice_id** (Primary key): UUID for *ap-slice* resource. ID format is 36 characters (including dashes) of the form "xxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx".
   - **tenant_id**: Name of the tenant that owns this slice.
   - **project_id**: Name of the project in which this slice belongs to.
   - **physical_ap**: Name of the physical *ap* on which this slice is attached to.
   - **wnet_id**: UUID of the *wnet* resource on which this slice is attached to.
   - **status**: Status of the slice can take possible values: PENDING, ACTIVE, FAILED, DOWN, DELETING and DELETED (see Subsection 4.2.4).

```
+----------------+--------------+------+-----+---------+-------+
| Field          | Type         | Null | Key | Default | Extra |
+----------------+--------------+------+-----+---------+-------+
| ap_slice_id    | varchar(36)  | NO   | PRI | NULL    |       |
| tenant_id      | varchar(255) | YES  |     | NULL    |       |
| project_id     | varchar(255) | YES  |     | NULL    |       |
| physical_ap    | varchar(255) | YES  |     | NULL    |       |
| wnet_id        | varchar(36)  | YES  |     | NULL    |       |
| status         | enum         | NO   |     | NULL    |       |
+----------------+--------------+------+-----+---------+-------+
```

3. **wnet**: The *wnet* tables contain attribute information of virtual wireless network resources. It is not used in the current implementation.

- **wnet_id** (Primary key): UUID of the *wnet* resource. ID format is 36 characters (including dashes) of the form "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx".
- **name**: Name of the *wnet* resource
- **tenant_id**: Name of the tenant that owns this wnet.
- **project_id**: Name of the project in which this wnet belongs to.

```
+--------------+--------------+------+-----+---------+-------+
| Field        | Type         | Null | Key | Default | Extra |
+--------------+--------------+------+-----+---------+-------+
| wnet_id      | varchar(36)  | YES  | PRI | NULL    |       |
| name         | varchar(255) | YES  |     | NULL    |       |
| tenant_id    | varchar(255) | YES  |     | NULL    |       |
| project_id   | varchar(255) | YES  |     | NULL    |       |
+--------------+--------------+------+-----+---------+-------+
```

4. **location_tag**: The *location_tag* tables contain relational information about the location tags attached to each *ap*. The general rule is that each *ap* can have multiple location tags. Each location tag can link to multiple *ap*. The *ap-slice* can be indirectly linked to its location tags through the *physical_ap* field. As such, both *name* and *ap_name* can be duplicate but the combination cannot be duplicate (joint primary key).

   - **name** (joint primary key): Name of the location tag.
   - **ap_name** (joint primary key): Name of the access point resource node.

```
+-------------+--------------+------+-----+---------+-------+
| Field       | Type         | Null | Key | Default | Extra |
+-------------+--------------+------+-----+---------+-------+
| name        | varchar(255) | NO   | PRI |         |       |
| ap_name     | varchar(255) | NO   | PRI |         |       |
+-------------+--------------+------+-----+---------+-------+
```

5. **tenant_tag**: The *tenant_tag* tables contain relational information about the tenant-added tags attached to each *ap-slice*. The general rule is that each *ap-slice* can have multiple (or none) tenant tags. Each tenant tag can link to multiple *ap*. The *tenant_id* field of *ap-slice* must be used to differentiate between the same tag used by different tenants. Both *name* and *ap_slice_id* can be duplicate but the combination cannot be duplicate (joint primary key).

- **name** (joint primary key): Name of the tenant tag.
- **ap_slice_id** (joint primary key): UUID of the *ap-slice* resource. ID format is 36 characters (including dashes) of the form "xxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx".

```
+-------------+--------------+------+-----+---------+-------+
| Field       | Type         | Null | Key | Default | Extra |
+-------------+--------------+------+-----+---------+-------+
| name        | varchar(255) | NO   | PRI |         |       |
| ap_slice_id | varchar(36)  | NO   | PRI |         |       |
+-------------+--------------+------+-----+---------+-------+
```

A.4 Aurora-Manager JSON Configuration Database

The JSON configuration database residing in Aurora-Manager is a database in which the blueprint configuration of each wireless slice (in this implementation *ap-slice*) is stored in a separate JSON file. Each tenant has a folder named by its *tenant_id*. Each slice file is named by its *slice_id* and is put into a tenant folder. As such, each slice can be accessed using the file path "[tenant_id]/[slice_id].json". The blueprint files are stored in the same format specified in Appendix A.2 and can be directly used in slice creation commands to Aurora-Agent.

This database is more an *archive* and as such is not designed to be searchable. Instead, searches should be performed using the MySQL resource database. Then, the *slice_id* obtained can be directly used to form the file path of the slice configuration and access it. If the need for searching particular module components inside blueprint file arises, future extensions can consider creating relational MySQL tables for modules and resource components inside slices.

A.5 Aurora-AP Local Database

The local database reside on each Aurora-Agent. As mentioned in Subsection 4.2.4, the local database only resides in temporary memory as long as Aurora-Agent is active. As such, it only contains information about wireless slices that are currently active (or believed to be active) on the resource node. The agent calls different modules and plug-ins to orchestrate and build the slice on behalf of the clients. Separate database entries are created for *each* resource component. Each entry is a dictionary that can be identified by four fields:

tenant_id, *slice_id*, *flavor*, *attributes*. These resource components entries are linked inside two Python dictionaries. The first *slice dictionary* groups the components by tenant and slice. The other *component dictionary* groups the components by abstraction module and plug-in flavor.

The slice dictionary is used to restrict the components within the scope of a single tenant and provide configuration isolation between them. For instance, a virtual interface component from one tenant cannot be attached to a virtual bridge component owned by another tenant. The component dictionary is used when specific plug-ins need to be restarted. In that case, the agent can obtain a list of all resource components that use that plug-in across all tenants. The local database is illustrated in Figure A.2.

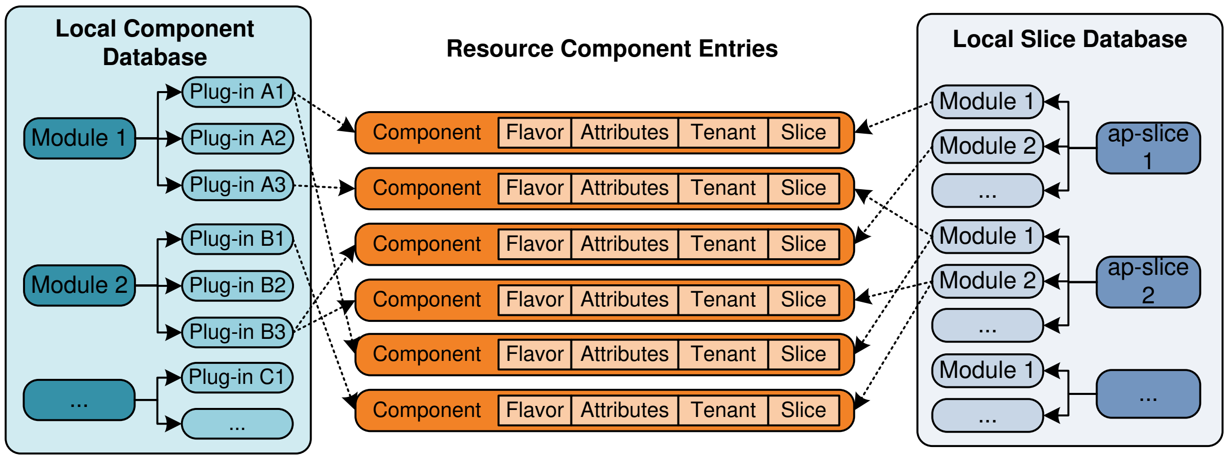

Fig. A.2: Local Agent Database Component Entries

Finally, the local agent database also loads a physical *ap* metadata file that describes the physical capabilities of the *ap* resource node (i.e, number of radios, firmware version, etc.) during the initialization of the agent. Some of these attributes could potentially be automatically detected on the resource node. Parts of the *ap* metadata are also found inside the resource database on Aurora-Manager. Currently, no automated synchronization for the *ap* metadata between the manager and the agents is implemented. These metadata files are written in JSON with the following sample format:

```
{
    "firmware":"OpenWRT",
    "firmware_version":"r37630",
    "aurora_version":"0.1",
    "memory_mb":"256",
    "wifi_radio":
    {
        "number_radio":"1",
        "number_radio_free":"1",
        "radio_list":[]
    }
}
```

A.6 Aurora Sample Blueprint

The sample blueprint given in this appendix section is a blueprint for the OpenFlow Wireless setup described in Subsection 5.3.1. It can be written in a format that aggregates the initialization of multiple *ap-slice* for a single tenant into a single blueprint. Even though this blueprint aggregation is optional and is not used when slices greatly differ from each other, it is useful in cases where a large quantity of similar slices must be created. By default, if only a single entry for an attribute is specified, that entry is applied to all *ap-slice* created using this blueprint. Otherwise, the number of entries must match the number of *ap-slice* defined on the blueprint.

1. **Tenant A Blueprint (Aggregated)**:

```
{
    "ap":["mg-ap-1","mg-ap-2","mg-ap-3"],
    "command":"create_slice",
    "user":"tenantA",
    "config":
    {
        "VirtualInterfaces": [
            {
                "flavor":"capsulator",
                "attributes":{
                    "attach_to":["eth0"],
                    "forward_to":["10.5.255.16"],
                    "name":["s1-tap0"],
                    "tunnel_tag":["1","2","3"],
                    "is_virtual":[true]
                }
            },
            {
                "flavor":"veth",
                "attributes":{
                    "attach_to":["wlan0"],
                    "name":["vwlan0"]
                }
            }
        ],
        "VirtualBridges": [
            {
                "flavor":"ovs",
                "attributes":{
                    "name":["ovs-1"],
                    "interfaces": [[
                        "s1-tap0",
                        "vwlan0"
                    ]],
                    "bridge_settings": {
                        "controller":["tcp:10.5.255.16"],
                        "dpid":[
                            "0000000000000001",
                            "0000000000000002",
                            "0000000000000003"
                        ]
                    },
                    "port_settings":{}
                }
            }
        ]
    }
}
```

2. **Tenant B Blueprint (Aggregated)**:

```
{
    "ap":["mg-ap-1","mg-ap-2","mg-ap-3"],
    "command":"create_slice",
    "user":"tenantB",
    "config":
    {
        "VirtualInterfaces": [
            {
                "flavor":"capsulator",
                "attributes":{
                    "attach_to":["eth0"],
                    "forward_to":["10.5.255.13"],
                    "name":["s2-tap0"],
                    "tunnel_tag":["1","2","3"],
                    "is_virtual":[true]
                }
            },
            {
                "flavor":"veth",
                "attributes":{
                    "attach_to":["wlan0-1"],
                    "name":["vwlan0-1"]
                }
            }
        ],
        "VirtualBridges": [
            {
                "flavor":"ovs",
                "attributes":{
                    "name":["ovs-2"],
                    "interfaces": [[
                        "s2-tap0",
                        "vwlan0-1"
                    ]],
                    "bridge_settings": {
                        "controller":["tcp:10.5.255.13"],
                        "dpid":[
                            "0000000000000001",
                            "0000000000000002",
                            "0000000000000003"
                        ]
                    },
                    "port_settings":{}
                }
            }
        ]
    }
}
```

References

[1] H. Wen, P. K. Tiwary, and T. Le-Ngoc, “Current Trends and Perspectives in Wireless Virtualization,” in *Proc. of 2013 IEEE MoWNet*, Aug. 2013.

[2] H. Wen, P. K. Tiwary, and T. Le-Ngoc, *Wireless Virtualization, SpringerBriefs in Computer Science*. New York, NY: Springer, 2013.

[3] “Network Functions Virtualisation: An Introduction, Benefits, Enablers, Challenges & Call for Action.” White Paper, SDN and OpenFlow World Congress, Oct. 2012.

[4] A. Attar, H. Li, and V. Leung, “Green last mile: how fiber-connected massively distributed antenna systems can save energy,” *Wireless Communications, IEEE*, vol. 18, pp. 66–74, 2011.

[5] Y. Zaki, L. Zhao, C. Grg, and A. Timm-Giel, “A Novel LTE Wireless Virtualization Framework,” in *Proc. of MONAMI’10*, 2010.

[6] P. Bosch, A. Duminuco, F. Pianese, and T. Wood, “Telco clouds and Virtual Telco: Consolidation, convergence, and beyond,” in *Proc. of the 2011 IFIP/IEEE IM*, May 2011.

[7] K.-K. Yap, R. Sherwood, M. Kobayashi, T.-Y. Huang, M. Chan, N. Handigol, N. McKeown, and G. Parulkar, “Blueprint for introducing innovation into wireless mobile networks,” in *Proc. of VISA’10*, 2010.

[8] S. Paul, J. Pan, and R. Jain, “Architectures for the future networks and the next generation Internet: a survey,” *Computer Communications*, vol. 34, pp. 2–42, Jan. 2011.

[9] J. Sachs and S. Baucke, “Virtual radio: a framework for configurable radio networks,” in *Proc. of WICON’08*, Nov. 2008.

[10] “Network Sharing: Architecture and Functional Description.” 3GPP Technical Specification 23.251 Version 11.4.0 Release 11, Jan. 2013.

[11] B. Naudts, M. Kind, F.-J. Westphal, S. Verbrugge, D. Colle, and M. Pickavet, "Techno-economic analysis of software defined networking as architecture for the virtualization of a mobile network," in *Proc. of EWSDN'12*, Oct. 2012.

[12] A. Wang, M. Iyer, R. Dutta, G. Rouskas, and I. Baldine, "Network Virtualization: Technologies, Perspectives, and Frontiers," *Journal of Lightwave Technology*, vol. 31, pp. 523–537, Aug. 2012.

[13] "GENI Spiral 2 Overview." Technical Report, Group-GENI, [Online]. Available: `http://groups.geni.net/geni/attachment/wiki/SpiralTwo/GENIS2Ovrvw060310.pdf`.

[14] A. Leon-Garcia, "NSERC Strategic Network on Smart Applications on Virtual Infrastructure." [Online]. Available: `http://www.savinetwork.ca/wp-content/uploads/Al-Leon-Garcia-SAVI-Introduction.pdf`.

[15] H. Bannazadeh and A. Leon-Garcia, "Virtualized Application Networking Infrastructure," in *Proc. of TridentCom'10*, May 2010.

[16] "Open source software for building private and public clouds." [Online]. Available: `http://www.openstack.org/`.

[17] J.-M. Kang, H. Bannazadeh, and A. Leon-Garcia, "SAVI Testbed: Control and Management of Converged Virtual ICT Resources," in *Proc. of 2013 International Symposium on Integrated Network Management*, May 2013.

[18] M. Hoffmann and M. Staufer, "Network Virtualization for Future Mobile Networks: General Architecture and Applications," in *Proc. of the 2011 IEEE ICC Workshops*, 2011.

[19] "OpenStack Neutron." [Online]. Available: `https://wiki.openstack.org/wiki/Neutron`.

[20] "Technical document on wireless virtualization." Technical Report, Group-GENI, [Online]. Available: `http://groups.geni.net/geni/attachment/wiki/OldGPGDesignDocuments/GDD-06-17.pdf`.

[21] R. Mahindra, G. Bhanage, G. Hadjichristofi, I. Seskar, D. Raychaudhuri, and Y. Zhang, "Space Versus Time Separation For Wireless Virtualization On An Indoor Grid," in *Proc. of 2008 Next Generation Internet Networks*, 2008.

[22] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling innovation in campus networks," *SIGCOMM Computer Communication Review*, vol. 38, 2008.

[23] "Open vSwitch An Open Virtual Switch." [Online]. Available: http://openvswitch.org/.

[24] "OpenFlow Switch Specification Version 1.0.0." [Online]. Available: http://www.openflow.org/documents/openflow-spec-v1.0.0.pdf, 2009.

[25] "OpenFlow Switch Specification Version 1.1.0." [Online]. Available: http://www.openflow.org/documents/openflow-spec-v1.1.0.pdf, 2011.

[26] "OpenFlow Switch Specification Version 1.2." [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.2.pdf, 2011.

[27] "OpenFlow Switch Specification Version 1.3.0." [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf, 2012.

[28] R. Sherwood, G. Gibb, and K.-K. Yap, "FlowVisor: A Network Virtualization Layer." Technical Report.

[29] A. Curtis, J. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Yalagandula, "DevoFlow: scaling flow management for high performance networks," in *Proc. of ACM SIGCOMM*, 2011.

[30] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "Nox: towards an operating system for networks," *SIGCOMM Computer Communications Review*, vol. 38, pp. 105–110, July 2008.

[31] P. Dely, J. Vestin, A. Kassler, N. Bayer, H. Einsiedler, and C. Peylo, "CloudMAC—An OpenFlow based architecture for 802.11 MAC layer processing in the cloud," in *Proc. of 2012 IEEE Globecom Workshops*, Dec. 2012.

[32] P. Gurusanthosh, A. Rostami, and R. Manivasakan, "SDMA: A Semi-Distributed Mobility Anchoring in LTE Networks," in *Proc. of 2013 IEEE MoWNet*, Aug. 2013.

[33] B. Aboba, "Virtual Access Points." IEEE 802.11-03/154rl. [Online]. Available: https://mentor.ieee.org/802.11/dcn/03/11-03-0154-00-000i-virtual-access-points.doc, 2003.

[34] G. Bhanage, D. Vete, I. Seskar, and D. Raychaudhuri, "SplitAP: Leveraging Wireless Network Virtualization for Flexible Sharing of WLANs," in *Proc. of 2010 IEEE Globecom*, Dec. 2010.

[35] Y. Dong, X. Yang, J. Li, K. Li, K. Tian, and H. Guan, “High Performance Network Virtualization with SR-IOV,” in *Proc. of the IEEE International Symposium on High Performance Computer Architecture*, 2010.

[36] S. Tripathi, N. Droux, T. Srinivasan, and K. Belgaied, “Crossbow: From hardware virtualized NICs to virtualized networks,” in *Proc. of the 1st ACM workshop on Virtualized infrastructure systems and architectures*, 2009.

[37] L. Xia, S. Kumar, X. Yang, P. Gopalakrishnan, Y. Liu, S. Schoenberg, and X. Guo, “Virtual WiFi: Bring Virtualization from Wired to Wireless,” in *Proc. of VEE’11*, Mar. 2011.

[38] Y. Al-Hazmi and H. de Meer, “Virtualization of 802.11 interfaces for Wireless Mesh Networks,” in *Proc. of WONS’11*, Jan. 2011.

[39] G. Aljabari and E. Eren, “Virtualization of Wireless LAN Infrastructures,” in *Proc. of 6th IEEE International Conferenxce on Intellignet Data Acquistion and Advanced Computer Systems*, 2011.

[40] P. Salvador, S. Paris, C. Pisa, P. Patras, Y. Grunenberger, X. Perez-Costa, and J. Gozdecki, “A Modular, Flexible and Virtualizable Framework for IEEE 802.11,” in *Proc. of 2012 Future Network and Mobile Summit*, 2012.

[41] G. Bhanage, I. Seskar, R. Mahindra, and D. Raychaudhuri, “Virtual Basestation: Architecture for an Open Shared WiMAX Framework,” in *Proc. of VISA’10*, 2010.

[42] G. Bhanage, R. Daya, I. Seskar, and D. Raychaudhuri, “VNTS: A Virtual Network Traffic Shaper for Air Time Fairness in 802.16e Systems,” in *Proc. of IEEE International Conference on Communications*, May 2010.

[43] R. Kokku, R. Mahindra, H. Zhang, and S. Rangarajan, “NVS: a virtualization substrate for WiMAX networks,” in *Proc. of MobiCom’10*, 2010.

[44] P. Serrano, P. Patras, X. Perez-Costa, B. Gloss, and D. Chieng, “A MAC Layer Abstraction for Heterogeneous Carrier Grade Mesh Networks,” in *Proc. of the ICT-Mobile Summit*, 2009.

[45] “GNU Radio Overview.” [Online]. Available: `http://gnuradio.org/redmine/projects/gnuradio`.

[46] K. Tan, H. Liu, J. Zhang, Y. Zhang, J. Fang, and G. M. Voelker, “Sora: High-Performance Software Radio Using General-Purpose Multi-Core Processors,” *Communications of the ACM*, vol. 54, pp. 99–107, Jan. 2011.

[47] M. Bansal, J. Mehlman, S. Katti, and P. Levis, "OpenRadio: A Programmable Wireless Dataplane," in *Proc. of HotSDN'12*, Aug. 2012.

[48] Y. He, J. Fang, J. Zhang, H. Shen, K. Tan, and Y. Zhang, "MPAP: virtualization architecture for heterogenous wireless APs," *SIGCOMM Computer Communication Review*, vol. 41, Jan. 2011.

[49] K. Tan, H. Shen, J. Zhang, and Y. Zhang, "Enabling Flexible Spectrum Access with Spectrum Virtualization," in *Proc. of DySPAN'12*, Oct. 2012.

[50] S. Hong, J. Mehlman, and S. Katti, "Picasso: flexible RF and spectrum slicing," in *Proc. of the ACM SIGCOMM*, Aug. 2012.

[51] G. J. Popek and R. P. Goldberg, "Formal Requirements for Virtualizable Third Generation Architectures," *Communications of the ACM*, vol. 17, pp. 412–421, 1974.

[52] K.-H. Kim, S.-J. Lee, and P. Congdon, "On Cloud-Centric Network Architecture for Multi-Dimensional Mobility," in *Proc. of MCC'12*, Aug. 2012.

[53] F. Boccardi, O. Aydin, U. Doetsch, T. Fahldieck, and H.-P. Mayer, "User-Centric Architectures: Enabling CoMP Via Hardware Virtualization," in *Proc. of PIMRC'12*, 2012.

[54] "PC Engines alix3d2 product file." [Online]. Available: `http://www.pcengines.ch/alix3d2.htm`.

[55] "WARP: Wireless Open Access Research Platform." [Online]. Available: `http://warpproject.org/trac`.

[56] "Python Programming Language." [Online]. Available: `http://www.python.org/`.

[57] "MySQL :: The world's most popular open source database." [Online]. Available: `http://www.mysql.com/`.

[58] "Capsulator - OpenFlow Wiki." [Online]. Available: `http://archive.openflow.org/wk/index.php/Capsulator`.

[59] "OpenWrt Wireless Freedom." [Online]. Available: `https://openwrt.org/`.

[60] "OpenStack Keystone." [Online]. Available: `https://wiki.openstack.org/wiki/Keystone`.

[61] "OpenStack API Quick Start." [Online]. Available: `http://docs.openstack.org/content/`.

[62] "RabbitMQ - Messaging that just works." [Online]. Available: http://www.rabbitmq.com/.

[63] "Keystone Middleware Architecture." [Online]. Available: http://docs.openstack.org/developer/keystone/middlewarearchitecture.html.

[64] "The tc manpage." [Online]. Available: http://lartc.org/manpages/tc.txt.