Journal of Network and Systems Management (2026) 34:86
https://doi.org/10.1007/s10922-026-10064-3

Leveraging Kubernetes for Automated Deployment and Orchestration in Virtualized Wi-Fi Networks

María Canales[1] · **José Ramón Gállego**[1] · **Julia Santacruz**[1] · **José Ruiz-Mas**[1] · **Ángela Hernández-Solana**[1] · **Julián Fernández-Navajas**[1] · **Jorge Ortín**[1,2]

Received: 9 September 2025 / Revised: 2 March 2026 / Accepted: 22 April 2026
© The Author(s) 2026

Abstract
The increasing prevalence of wireless devices has led to the widespread adoption of Wi-Fi networks. This demand has driven the development of solutions that involve the coordinated management of Access Points. In recent years, Software-Defined Wireless Networks (SDWNs) have emerged as a promising approach, leveraging Software-Defined Networking (SDN) principles to enable intelligent coordination and management of APs using cost-effective hardware. However, while SDWNs have primarily focused on managing the radio interface, less attention has been devoted to SDN-based management of wired network segments or the automated deployment of virtualized Wi-Fi infrastructures. In this work, we present a fully virtualized Wi-Fi network solution that integrates SDN and Network Function Virtualization (NFV) principles across both wireless and wired segments. Our architecture features SDN-based data plane management, two SDN controllers, virtualized APs, and a virtualized gateway, all implemented as containerized components. Automated deployment and orchestration are facilitated using Kubernetes. This infrastructure provides a robust foundation for implementing advanced end-to-end functionalities, such as network slicing, thereby preparing Wi-Fi networks for seamless integration into future 5G and 6G ecosystems.

Keywords Wi-Fi networks · Software defined networks · Kubernetes · Orchestration · Overlay networks

✉ María Canales
mcanales@unizar.es

✉ José Ramón Gállego
jrgalleg@unizar.es

1 Aragón Institute for Engineering Research (I3A), University of Zaragoza, 50018 Zaragoza, Spain

2 Centro Universitario de la Defensa, 50090 Zaragoza, Spain

Published online: 06 May 2026

1 Introduction

In recent years, there has been a significant increase in the use of wireless devices such as smartphones, laptops, and tablets. This surge has driven the widespread deployment of wireless networks in various settings, including business centers, airports, malls, campuses, and even entire cities. Among these networks, Wi-Fi (IEEE 802.11) has emerged as the default technology for connecting wireless devices.

This trend has fostered the development of solutions involving coordinated Wi-Fi Access Points (APs), commonly referred to as “Enterprise Wi-Fi". However, such solutions are often proprietary, closed, and expensive, rendering them unaffordable for many organizations.

To address these limitations, the scientific community has explored proposals for inter-AP coordination, enabling advanced features such as load balancing, frequency planning, and power control while leveraging low-cost hardware and open-source software. Notably, there has been growing interest in adapting concepts from Software-Defined Networking (SDN) [1]–such as the abstraction of flows– for application in wireless networks. This approach facilitates monitoring the wireless environment and managing programmable APs, enabling the implementation of intelligent algorithms through centralized SDN controllers. This area of research is known as Software-Defined Wireless Networks (SDWNs) [2–6]. As part of this effort, we introduced a comprehensive architecture that integrates handoff mechanisms with monitoring tools and other functionalities. This solution provides intelligent features using low-cost commercial APs, offering a practical and scalable approach to wireless network management [4, 5].

In line with 5G standards, both Software-Defined Networking and Network Function Virtualization (NFV) are technologies that can drive the evolution of Wi-Fi networks toward solutions offering enhanced coordination and deployment flexibility [7–10]. Much of the SDWN research effort has focused on managing radio interfaces, assuming that all core network elements reside within the same physical network. However, the management of the wired portion of the data using SDN, particularly in scenarios where network elements (such as APs and routers) are distributed across different physical locations, has not been addressed in these works.

Furthermore, NFV enhances network flexibility by virtualizing functions through virtual machines or containers. This allows for the customization of infrastructure and adaptation to the dynamic nature of user demands and service requirements. The integration of orchestration systems, such as Kubernetes, for managing and orchestrating virtualized infrastructure necessitates a detailed analysis of the specific characteristics of wireless networks. These include dependence on radio technology, hardware limitations, connectivity intermittency, and unique radio resource constraints. These factors may require specialized virtual infrastructure and tailored controllers. While the adoption of NFV is widespread in the context of 5G and 6G networks, to the best of the authors’ knowledge, there is currently no solution addressing the automated deployment of a virtualized Wi-Fi network via a cloud orchestration platform.

In this work, we propose a fully virtualized Wi-Fi network solution that includes the deployment of APs and the default gateway, where SDN management of the data

plane is performed in both the wireless and wired segments. This solution is automatically deployed through orchestration using Kubernetes.

The main novel contributions of this paper are as follows:

- SDN-based management of the data plane in the wired portion of the network, across different physical networks through an overlay network, including mobility management resulting from handovers within the Wi-Fi access network.
- Full virtualization of the Wi-Fi network, encompassing two SDN controllers (wireless and wired), APs, and a Router/DHCP implemented as containers.
- Automated deployment and orchestration of this network using Kubernetes.

This infrastructure provides a solid foundation for integrating new end-to-end functionalities, such as the implementation of network slicing within the network, a feature we are already analyzing in the radio interface [11].

The rest of the paper is organized as follows: Sect. 2 provides a summary of the main related work. Section 3 describes the complete SDWN architecture, including both the data and control planes. The automated deployment process based on Kubernetes is detailed in Sect. 4. Section 5 offers an in-depth analysis of use cases in a real-world deployment, and finally, the main conclusions are presented in Sect. 6.

2 Related Work

2.1 Virtual APs and SDWN Architectures

SDN is a network paradigm design that separates the control plane from the data plane, enabling centralized management, programmability, and automation. This approach enhances agility, scalability, and efficiency by allowing software-driven control over network resources. An SDN architecture typically consists of three key components: the application plane, the control plane, and the data plane. At the heart of SDN is a centralized control plane, which uses a logically centralized controller to oversee the entire network. The controller communicates with data plane devices using standardized protocols like OpenFlow. Control and data planes are decoupled, with the control plane handling decision-making processes and the data plane focusing solely on forwarding packets. In addition, SDN introduces programmability by abstracting the underlying hardware and exposing the network's behavior through APIs: northbound APIs to enable communication between the control plane and application plane, allowing applications to define high-level policies, and southbound APIs to facilitate communication between the control plane and data plane, ensuring efficient execution of instructions on network devices [1].

Suresh et al. [2] propose Odin, an SDWN framework designed for enterprise WLANs that leverages virtual APs to simplify application development and abstract the station (STA) mobility across multiple physical APs. This is achieved through the introduction of the Light Virtual AP (LVAP) abstraction. When a STA connects for the first time, instead of associating with a physical AP, a central SDN controller creates a unique LVAP for the STA. As explained in Sect. 3, an LVAP consists

of four fields: a virtual Service Set Identifier (SSID), a virtual Basic Service Set Identifier (BSSID), the STA's actual MAC address, and its IP address. When the STA moves out of an AP's range, the SDN controller dynamically migrates the LVAP to another AP. Since the STA retains the same BSSID and IP address, traditional reassociation and reauthentication processes are not required, ensuring seamless handover. However, Odin has two main limitations: first, it assumes that all APs operate on the same channel, preventing effective channel planning. Second, it faces scalability challenges, as broadcast beacons cannot be used; instead, the AP must send unicast beacons with a specific MAC address to each STA. Additionally, the AP must be capable of generating the Wi-Fi ACKs corresponding to each STA.

To overcome these limitations and following this Virtual AP paradigm, several architectural proposals for SDWN have been proposed in the last years. The EmPOWER architecture introduced in [12] integrates multiple Radio Access Technologies (RATs) and provides a set of programming abstractions to model key aspects of wireless networks. This architecture was further utilized in [13], where a joint algorithm for mobility management and rate adaptation in multicast communications over 802.11 networks was proposed. Additionally, it was expanded in [14] by developing an architecture compliant with 4G and 5G networks, but without addressing its Wi-Fi scalability issues. Isolani et al. [15] extended the EmPOWER architecture with an interactive management framework based on in-band network telemetry (INT), enabling fine-grained visibility of SD-RAN deployments at the controller. Similarly, [16] introduced a Wi-Fi architecture designed to minimize packet-level delay violations for specific target services. In this approach, all APs (named Base Stations (BSs) in the paper) are configured to use the same MAC address, while virtual APs are created to manage individual services. We also introduced an open-source framework detailed in [5] and derived from [4] that integrates handoff mechanisms and multichannel support with monitoring tools and other intelligent functionalities. In [6], a similar virtual AP solution based on SDN is proposed. This solution fully supports multichannel migrations via the IEEE 802.11h Channel Switch Announcement, without imposing limitations on the channel utilization by the APs.

The LVAP abstraction dictates that each LVAP supports only a single STA, necessitating that physical APs host multiple LVAPs. Since each LVAP triggers the creation of a BSS, this means that a physical AP must host multiple BSSs simultaneously, which increases overhead on the wireless medium. In contrast, the BIGAP architecture presented in [3] utilizes a single global BSSID for the entire Extended Service Set (ESS), encompassing all APs. From the STA's perspective, the entire ESS, including all APs, is viewed as a single BSS or a large AP. This approach improves scalability with respect to both the number of serving STAs and AP density, as it reduces the wireless signaling overhead required to manage LVAPs.

Each of these proposals employs a central controller responsible for network management. However, all of them assume that network elements, such as APs and routers, reside within the same physical network. They do not address the SDN-based management of the wired segment, which becomes critical when these elements are distributed across different physical locations and constitutes one of the main contributions of this work.

2.2 NFV and Orchestration

NFV combined with SDN is driving the evolution of 5G standards toward solutions that provide improved coordination and greater deployment flexibility [7–10]. In this context, the integration of orchestration systems, such as Kubernetes, for managing and orchestrating virtualized infrastructure in 5G networks has been receiving significant attention in recent years:

Makris et al. [17] propose a framework for service orchestration over wireless network slices, providing insights into the benchmarking of softwarized 5G Radio Access Networks in real testbeds. Similarly, Kassis et al. [18] develop an integrated deployment prototype for the automatic and dynamic integration between a Virtual Network Embedding solution (VNE) and Kube5G service platform. Vittal et al. [19] also introduce a zero-touch emulation framework for network slicing management in a 5G Core testbed, to enable autonomous orchestration in an NFV environment. The authors in [20] discuss a Kubernetes-based Network Functions (KNF) deployment of an open-source 5G core network, free5GC, using Open Source Mano (OSM) and adopting a cloud-native approach. The study highlights the advantages of a containerized approach, particularly in terms of availability. The authors in [21] present an open-source containerized implementation of a 5G Standalone (SA) network, built on the principles of 5G cloud network functions, Docker containers, and Linux virtualization. Their implementation supports both minimalist and basic deployment models and adheres to 3GPP Release-16 specifications. Core network components are developed using the OpenAirInterface (OAI) 5G Core (5GC) network platform and deployed via Docker Compose. Scotece et al. [22] utilize container-based technologies and Kubernetes to design and evaluate an innovative, low-cost, ready-to-deploy solution named 5G-Kube, tailored for softwarized 5G core networks. Their evaluation focuses on two distinct use cases demonstrating the adaptability of 5G-Kube to 5G Core and Kubernetes deployment scenarios: Industry 4.0 and Smart Cities. Spantideas et al. [23] detail a framework for smart mission-critical service management, including architectural specifications and experimental results for deployments over 5G networks and beyond. In [24], the authors provide insights gained from building an end-to-end software-based 5G testbed, offering guidance to help researchers address the technical challenges associated with such an undertaking. They introduce two different 5GC testbed designs: one using two desktop PCs and another utilizing a single high-performance server. Key lessons learned during the installation process are highlighted, and potential research opportunities enabled by the testbed are discussed. Bonati et al. [25] present 5G-CT, an automation framework built on Red Hat OpenShift that utilizes the GitOps workflow to automatically deploy and test softwarized end-to-end 5G and Open Radio Access Network (O-RAN)-compliant systems within seconds, requiring no human intervention. They integrate 5G-CT with an over-the-air Software Defined Radio (SDR) testbed and demonstrate its application in testing open-source protocol stacks for cellular networks. Andrade et al. [26] evaluated resource allocation in private 5G networks using open-source cloud-native tools to achieve effective isolation for 5G network slices. Results from a hospital video conferencing scenario indicate that CPU constraints improve prioritized slice performance, while memory limits have limited impact.

These management principles, focusing on automated lifecycle management and operational resilience, are becoming the standard for 5G core orchestration. Silva et al. [27] survey the transition toward virtualized wireless communications identifying the integration of Wi-Fi into B5G systems as a primary research challenge. In this context, while network virtualization through NFV and orchestration tools such as Kubernetes is widespread in 5G networks, its adoption in Wi-Fi remains limited. OpenSDWN [28] is a platform that leverages per-client LVAPs, programmable datapath rules (WDTX), and virtual middleboxes (vMBs) to provide service differentiation, fine-grained transmission control, and seamless mobility within a logically centralized control plane. However, OpenSDWN's mobility support is limited to access points within the same physical or IP network and it does not address automated deployment or orchestration. In contrast, our work introduces dynamic tunneling across distinct IP subnets to ensure transparent handovers even when access points belong to different networks, and incorporates Kubernetes-based automated deployment and orchestration –two additional key contributions of this work.

3 SDWN Architecture

This section provides a detailed description of our proposed SDWN architecture. As stated earlier, this work builds upon the SDWN framework introduced in [4] and [5] with the inclusion of the BIGAP concept [3]. In those works, a proposal was made to integrate coordination mechanisms that enhance the capabilities of centrally managed Wi-Fi APs. As illustrated in Fig. 1, that architecture includes monitoring tools and additional functionalities, enabling intelligent network management while utilizing low-cost commercial APs. By leveraging comprehensive network information collected by central controllers, those proposals demonstrated the potential for making informed resource allocation decisions. To ensure proper operation, it was essential to use two independent networks: one for control and the other for data.

In line with the SDN concept, the design decouples the control plane from the data plane, as shown in Figs. 1 and 2. The control plane manages radio functionality and configures switching and routing within the data plane. To achieve this, two central controllers are utilized: a standard OpenFlow controller (based on Ryu [29]) for configuring the wired portion of the data path, and a custom-developed controller, Odin [4, 5], for managing the wireless segment. The OpenFlow controller oversees the virtual switches deployed along the data path, while the Odin controller interacts with Odin Agents located at the APs to manage all radio functionalities and custom wireless management applications.

For completeness, the main features of the previous proposal, including the Wireless Controller (Odin) and the LVAP concept, are summarized in this section. However, the main contribution of this work lies in the SDN-based management of the data plane within the wired segment of the network, eliminating the need for dedicated network infrastructure or exclusive servers. Instead, it leverages overlay networks and containerized deployments to achieve greater flexibility and efficiency. This includes mobility management to support handovers within the Wi-Fi access network, as implemented through the SDN Controller.

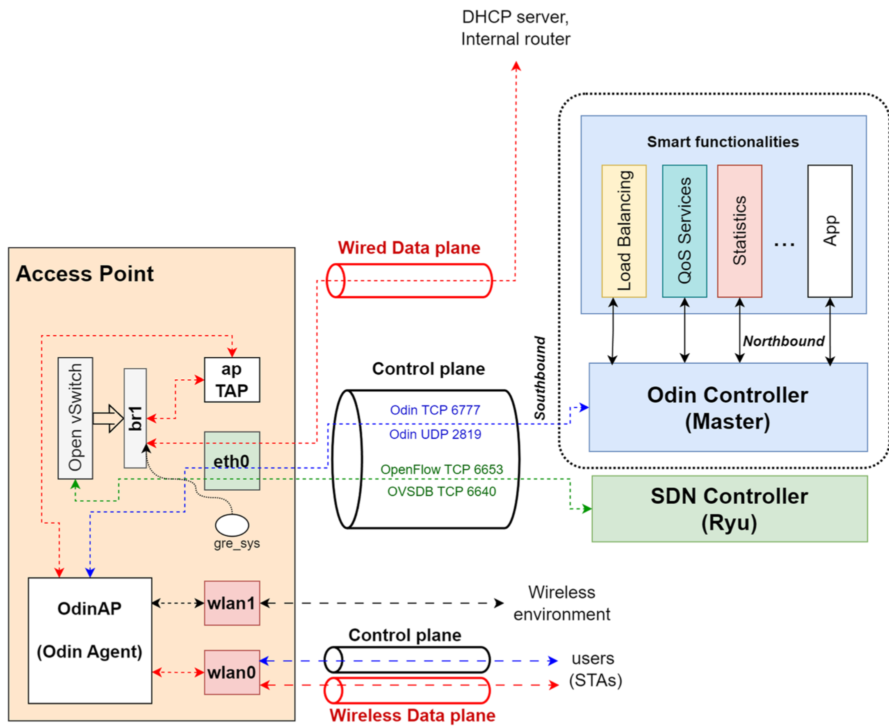

Fig. 1 SDWN architecture. Functional components and protocols. Basic virtual AP and controllers

3.1 Data Plane

Data plane establishment requires that legacy STAs associate with any available AP corresponding to the requested SSID by following standard wireless procedures. The custom-designed virtual APs, implemented through a standalone user-space application written in C (OdinAP) [5], enable seamless STA associations by exchanging the necessary information with their central controller. As shown in Figs. 1 and 3, the virtual AP uses the primary interface *wlan0* to communicate with STAs (data and signaling interchange), while the auxiliary interface *wlan1* enables the Odin Agent to monitor wireless channels and assist the controller in resource management. Once the STA is associated, the wireless data path is established, allowing configuration of the complete path, including DHCP-based IP address allocation and the wired connection between the AP and the Internet Router. The Router serves as both a DHCP server and a Network Address Translator (NAT), providing Internet access.

To enable deployment in any location, regardless of the underlying network infrastructure, a tunneled wired data path is established using an overlay SDN approach (E-LAN), as illustrated in Fig. 3. This figure highlights the connected interfaces: a virtual TAP interface (*tap0*) in the AP, which handles traffic for STAs to and from the OdinAP process, and an “internal port" (*router*) in the default gateway (DHCP/Router). GRE (Generic Routing Encapsulation) tunnels configured over the physical interfaces (*eth0* in the figure) are employed, although alternative tunneling methods

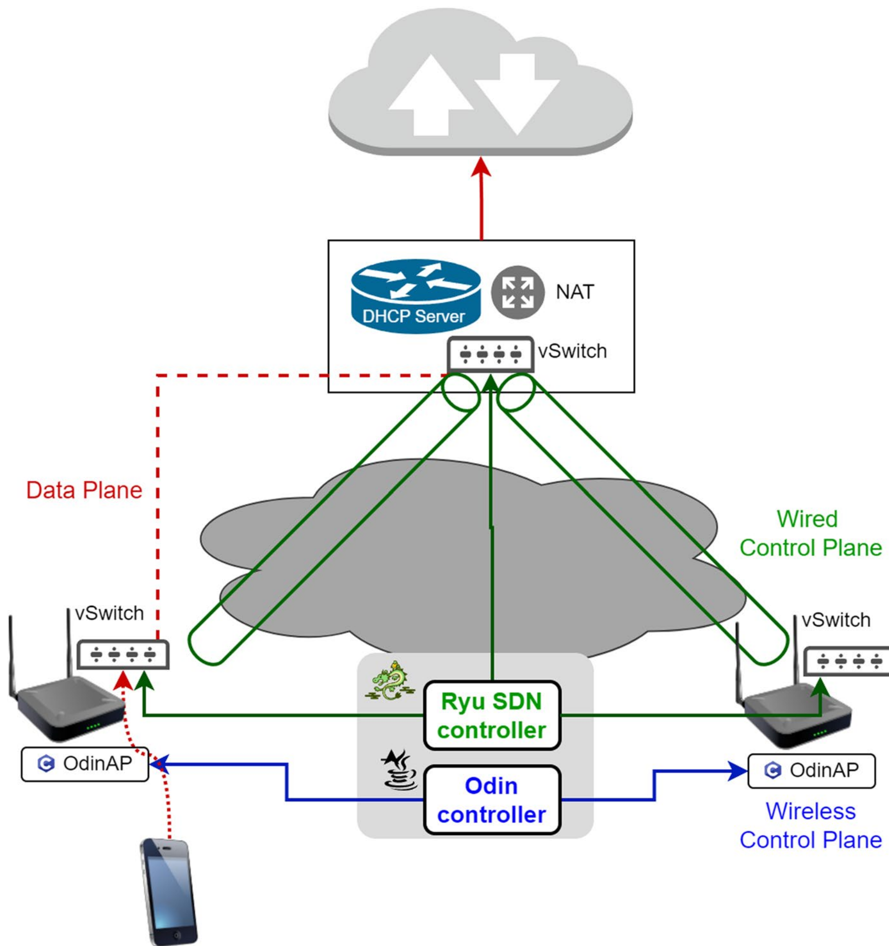

Fig. 2 SDWN architecture. Control and data planes

can also be utilized. Additionally, Fig. 3 depicts an encapsulated Ethernet frame as an example of traffic addressed to an external web server. According to the E-LAN overlay network, this traffic follows the physical routing path between the nodes hosting the AP at IP_{node1} and the default Router at IP_{node2}.

3.2 Control Plane

3.2.1 Odin Wireless Controller and LVAP

As mentioned, the SDWN approach introduces programmability through a centralized controller, enabling virtual AP management and the monitoring of key statistics. This centralized coordination optimizes resource management across APs. By exposing a northbound API, the approach allows network applications to programmatically orchestrate the underlying wireless network, translating API calls into commands

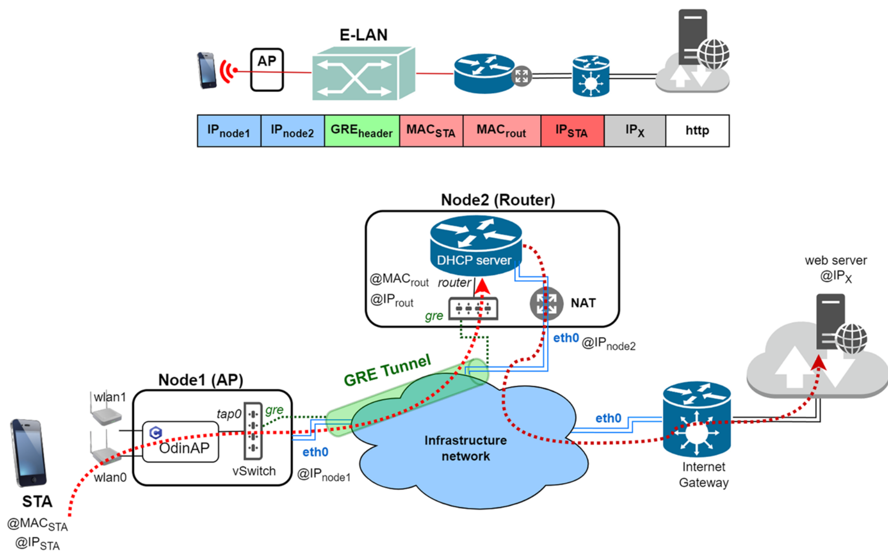

Fig. 3 Data plane. Uplink traffic from STA to internet

executed on network devices via the southbound API. The controller provides a comprehensive view of the network, including clients and APs, empowering Odin applications to manage them effectively.

Although OpenFlow has become the de facto standard for SDN architectures, it does not seamlessly integrate with the IEEE 802.11 MAC layer, as its functionality is primarily designed for programming flow tables on Ethernet-based switches. For instance, OpenFlow lacks the ability to match wireless frames, collect wireless medium measurements, report per-frame receiver-side statistics, or configure transmission parameters for specific frames or flows in the Wi-Fi datapath. To address these gaps, the Odin protocol facilitates communication between the controller and APs. This protocol handles all control and management exchanges, leveraging TCP connections for tasks like transmitting association-related information, while UDP connections manage periodic reports and notifications.

Odin agents, running on wireless APs, provide the essential hooks for the controller –and by extension, network applications– to orchestrate the Wi-Fi network and retrieve relevant metrics. Time-critical operations of the Wi-Fi MAC protocol, such as IEEE 802.11 acknowledgments, are still handled by the hardware of the Wi-Fi NIC. In contrast, non-time-critical functions, including client association management, are implemented in software on both the controller and the agents, enabling a distributed Wi-Fi split-MAC architecture. These agents also perform frame matching to support a publish-subscribe system, where network applications can subscribe to per-frame events.

For wireless network applications to make informed control decisions, they require not only per-frame statistics but also broader wireless medium measurements (e.g., detecting interference from non-Wi-Fi devices operating in the same spectrum). Odin applications operate reactively or proactively, leveraging cross-layer measure-

ments to implement network programming through the northbound API provided by the controller.

The architecture leverages the LVAP abstraction, initially proposed in [2]. The controller assigns each user a virtual AP, which consists of a virtual SSID, a virtual BSSID, the station's real MAC address, and a unique IP address. In [5], this LVAP was extended to enhance the integration of diverse STAs and optimize resource management in WLANs with Wi-Fi APs featuring heterogeneous characteristics, such as different 802.11 variants, capabilities, and security schemes (Fig. 5). The LVAP follows the user during mobility across physical APs without requiring any modifications to the user's terminal running the 802.11 standard. When a STA joins the network and associates with an initial serving AP, the Odin controller uses its LVAP to track the STA. During handovers, this information is seamlessly transferred between APs. Since the STA retains the same BSSID and IP address, traditional reassociation and reauthentication processes are not required, ensuring seamless mobility.

Therefore, the architecture uses the LVAP abstraction as state information associated with the STA. However, in this proposal, to reduce wireless medium overhead, we do not create a BSS for each LVAP. Instead, we implement a solution based on the BIGAP concept [3]. BIGAP employs a single global BSSID for the entire ESS, encompassing all APs. From the STA's perspective, the entire ESS, including all APs, appears as a single BSS or one large AP. To achieve this, BIGAP assigns different RF channels to all co-located APs. During the handover process, BIGAP leverages the 802.11 DFS functionality, leading the STA to believe that the serving AP is performing an RF channel switch. As long as the STA perceives only one AP, it will not initiate a roaming decision, enabling the network to manage clients in a coordinated manner.

To understand the critical role of the Odin controller in maintaining the wireless data path, which is subsequently linked to wired data path management, the key procedures are summarized below.

(a) *Wireless data path establishment after association*
The Odin agent in the AP follows the standardized 802.11 procedures for STA association and authentication, which establishes the wireless data path. During the wireless signaling exchange, the Odin agent communicates custom signaling messages with the Odin controller to create the corresponding LVAP and register the STA in the network, associating it with its serving AP. Since the LVAP information includes the STA's allocated IP address, signaling between the STA and the DHCP server is also required. This traffic must traverse the wired data path, which has not yet been configured. The necessary OpenFlow rules in the corresponding vSwitches (AP and Router) are created as data traffic traverses the network, thus involving the SDN controller, as explained later in b). Since DHCP messages are treated as data traffic, these packets establish the wired data path. However, the Odin Agent's intervention is still required to finalize the LVAP creation by capturing the corresponding DHCP ACK message for the STA and informing the Odin Controller accordingly. The entire procedure is illustrated in Fig. 4, which also includes the necessary OpenFlow exchange with the SDN controller. Steps 1a, 3, and 4a correspond to the standard 802.11

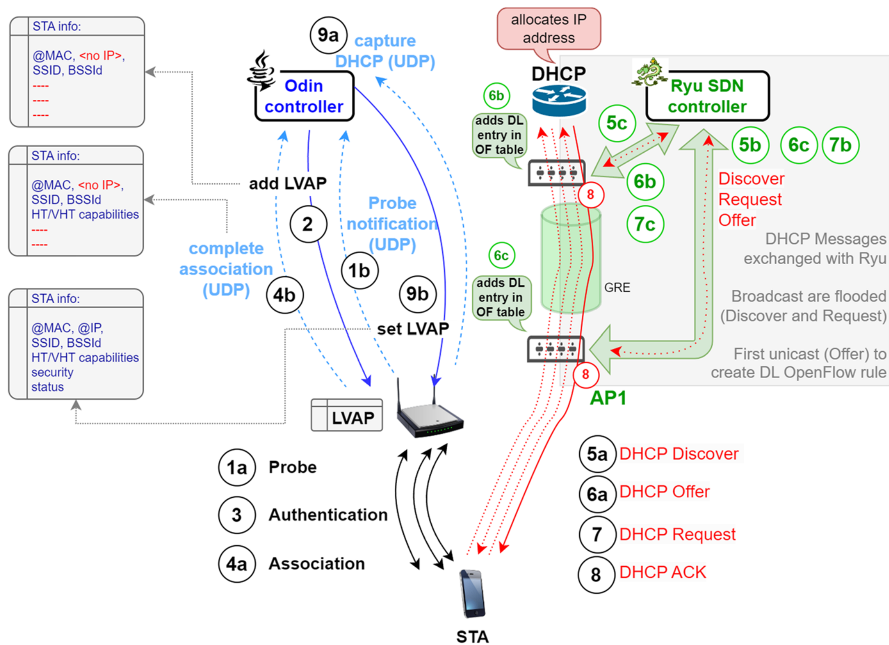

Fig. 4 STA association. Setting full LVAP and TCP/IP configuration (DHCP exchange). (Example Ryu messages in Fig. 6)

association process. While signaling occurs over the wireless medium, the Odin Agent in the AP communicates with its controller to register the STA and generate the LVAP. During the Probe phase (1a), a UDP notification sent to the Odin Controller (1b) initializes the LVAP, which is then created in the AP via the 'add LVAP' command (2) over the Odin TCP connection. Once authentication and association are completed (both reported to the Controller in 4b), the LVAP is updated but remains incomplete until the STA receives an IP address. At this stage, as the wireless data path is operational, the STA begins the DHCP configuration process. However, during the DHCP exchange, the wired segment is not yet established, as specific unicast entries in the OpenFlow table are missing. Consequently, the virtual switches rely on the SDN controller to properly direct the traffic. The next section provides a detailed explanation of this process, along with the SDN controller's role. Ultimately, once the DHCP ACK message reaches the STA –directly forwarded through the newly created downlink OpenFlow rules (8)– the Odin Agent informs the Odin Controller to finalize the LVAP (9b). At this point, the STA can transmit IP traffic using its assigned IP address. As unicast uplink data traverses the virtual switches, the SDN controller facilitates the creation of the corresponding unicast entries in the OpenFlow tables, thereby fully establishing the bidirectional data path.

(b) *Seamless handover*

As outlined in [4], the monitoring and handoff managers in the Odin controller are responsible for deciding which AP the STA should migrate to based on measurements and coordinating the communication of this decision to the STA

–by generating Channel Switch Announcement (CSA) beacons on the current serving AP– with the migration of the LVAP abstraction to the new AP and the sending of unicast welcome beacons to the STA from the new AP. As a result, the STA perceives that it remains connected to the original serving AP and simply performs a channel switch, while the new serving AP seamlessly takes over STA management upon its transition to the new channel, ensuring a smooth handoff. These procedures are summarized in Fig. 5 which illustrates the Odin protocol messages involved between two APs and the controller: As the STA moves (1) away from AP1 (channel A) and detects a weak signal, it sends a PUBLISH message to the controller (2), which requests neighboring APs (3) to listen on channel A. Those detecting the STA reply with a Scan Response (4), allowing the controller's resource management algorithm (5) to select AP2. The controller then instructs AP1 (6) to send CSAs (7), prompting the STA to switch channels after the specified beacons. Simultaneously, it adds the LVAP in AP2 (8) to start broadcasting beacons and removes it from AP1. As the STA receives AP2's beacons on channel B, it perceives the handover as a seamless channel switch within AP1. The explained wireless handoff, controlled by Odin, ensures transparency in the wireless data path, while the reconfiguration of the wired segment is still pending. The SDN Controller, described next, manages this reconfiguration while minimizing handoff latency.

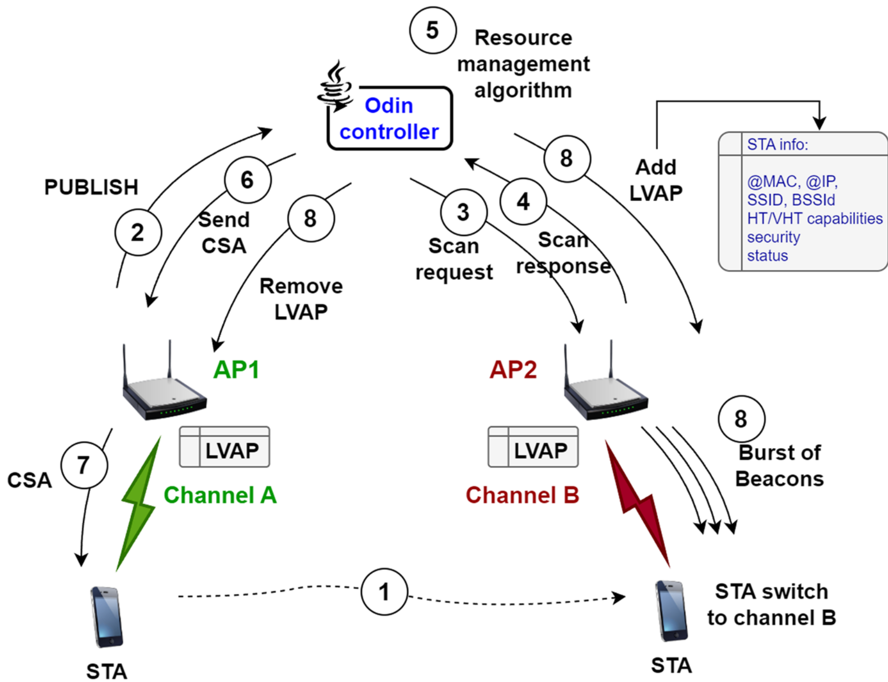

Fig. 5 Inter-channel seamless handoff process by LVAP migration [30]

3.2.2 Ryu-Based SDN Controller

As previously mentioned, the E-LAN in the wired segment of the data path is established and maintained through the interconnection of virtual switches, which are configured by an SDN controller. This controller is a custom-developed application built on the Ryu framework. Although the wireless and wired data paths are controlled separately (by the Odin and SDN controllers, respectively), some coordination is necessary. This coordination is achieved seamlessly by capturing specific data frames without requiring explicit signaling between the controllers.

The primary procedures initiated by the controller include the following:

(a) *Physical connectivity on deployment*
During the initialization of the APs and the Router, the corresponding virtual switches (vSwitch) connect to the SDN controller and complete the initial OpenFlow handshake. This process allows the controller to identify the IP addresses of the nodes hosting the virtual functions and initiate the creation of GRE tunnels, which are subsequently added as ports to the virtual switches using the OVSDB management protocol. Furthermore, the initialization scripts running within the AP or Router add the local ports (*tap0* for the AP or the internal *router* interface for the Router) to their respective virtual switches. As a result, physical connectivity between each AP and the Router is seamlessly established through the virtual switches. One of the critical design considerations is the precise identification of tunnel endpoints (IP addresses of the nodes) required for creating the GRE tunnels. Since the control plane and data plane are decoupled, the transport networks for signaling and data operate with distinct IP addressing schemes. As detailed later in Sect. 4, automating the deployment process requires resolving these addresses through an appropriate service name resolution mechanism. The adopted scheme associates the internal local ports of the deployed virtual switches with the hostnames of the physical nodes. During the OpenFlow handshake, specifically through “Port Description Reply" messages, the SDN controller identifies the hostname associated with the `OFPP_LOCAL` port. This enables the controller to determine the IP address of the physical node, regardless of the source IP address (control plane) used in the OpenFlow message.

(b) *Flow rules for STA on STA association*
Once the interconnection of switches is established, the data path between a STA and the Router/DHCP is dynamically created upon STA access via flow rules triggered by the controller. As explained before, during the association process, the Odin Agent in the serving AP communicates with its controller using the Odin Protocol to properly create the corresponding LVAP. However, completing this process requires the DHCP server to allocate an IP address to the STA, which is then included as part of the LVAP. To achieve this, the exchange of DHCP messages in the data path plays a fundamental role. These messages not only trigger the creation of flow rules in the virtual switches (with initial messages routed to the controller) but also provide the allocated IP address necessary to fully configure the LVAP. The procedure is briefly illustrated in Fig. 4. Virtual switches rely on existing OpenFlow table entries to forward traffic. When a rule

is missing, packets are sent to the SDN controller (table-miss entry) for instructions. In response, the controller creates the necessary table entries to handle future unicast traffic. Broadcast packets, such as DHCP Discover and DHCP Request, are always forwarded to the controller, whereas unicast packets trigger the creation of OpenFlow rules for subsequent communication. The DHCP Offer, as the first downlink packet from the Router to the STA, reaches the SDN controller, which then instructs the vSwitches to create the corresponding entry. This allows the DHCP ACK to be forwarded directly. Figure 6 details the OpenFlow message exchange between the vSwitch in the Router and the Ryu SDN controller: The DHCP Discover is received in an `OFP_PACKET_IN` message, matching the input port. This allows the SDN controller to learn the STA's MAC address and its corresponding port. The controller then instructs the vSwitch to flood the DHCP message by sending it back as an `OFP_PACKET_OUT`. The DHCP Offer is received in an `OFP_PACKET_IN` message, also matching the input port. This enables the controller to learn the Router's MAC address and its associated port. Since the packet is unicast and destined for the STA (whose MAC

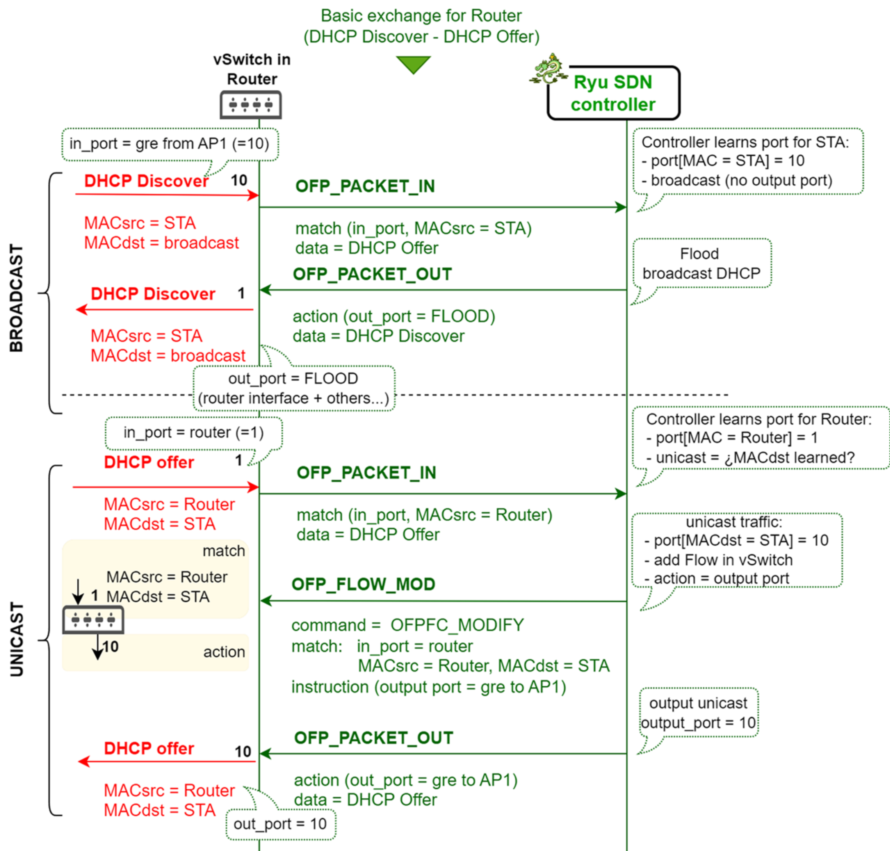

Fig. 6 OpenFlow messages with Ryu triggered by DHCP discover (broadcast) and offer (unicast). Wired downlink data-path establishment in the router vSwitch

address was previously learned), the controller performs two actions: it installs a new OpenFlow rule in the vSwitch table via OFP_FLOW_MOD and forwards the DHCP Offer according to this rule using OFP_PACKET_OUT. The DHCP Request broadcast message (not included in Fig. 6) follows the same OpenFlow message exchange and forwarding process as the DHCP Discover. However, the DHCP ACK is forwarded directly by the vSwitch, as the OpenFlow entry created during the DHCP Offer exchange already enables its transmission. In other words, the DHCP Offer message is responsible for establishing the downlink wired data path. In addition, as explained with the Odin Controller functionality, and illustrated in Fig. 4, in the absence of explicit signaling between the SDN and Odin controllers, the Odin Agent in the AP where the STA is associating performs two critical tasks: it encapsulates the DHCP ACK message in a 802.11 frame and forwards it to the STA, and it simultaneously sends the message to the Odin controller via UDP as part of the Odin signaling protocol.

(c) *Updating flow rules for STA during handoff*
Similarly, during handover, to minimize delays caused bySimilarly, during handover, to minimize delays caused by reestablishing the wired data path, the SDN controller must be notified to update flow rules accordingly. When a STA associates with a new AP, a new flow rule is naturally created in the corresponding vSwitch table. However, the Router retains pre-existing rules for the old AP. Instead of waiting for these rules to expire before creating new ones –causing unnecessary handoff delays– the proposed implementation explicitly triggers flow rule modifications. As explained in Sect. 3.2.1, while channel switching and LVAP migration ensure transparency in the wireless data path, the wired segment still requires reconfiguration. Without intervention, the wired data path is established dynamically by adding OpenFlow rules when data traffic traverses vSwitches. Uplink data packets arriving through new input vSwitch ports trigger rule creation: broadcast traffic always forwarded to the SDN controller allows it to learn the new location of source MAC addresses. Then, when unicast traffic does not match existing rules and is forwarded to the SDN controller, it updates the flow table accordingly. In contrast, downlink traffic (Router to STAs) continues to follow outdated OpenFlow rules, forwarding packets to the old AP until the rules expire, increasing handoff delays. To address this issue, the new serving AP generates gratuitous ARP packets on behalf of the STA following the handoff (specifically, once the LVAP is migrated and welcome beacons are transmitted for the STA). These broadcast packets are forwarded to the SDN controller, which instructs the relevant vSwitches to flood the network. Since broadcast ARP packets are flooded across the network, each vSwitch involved in the data path interacts with the SDN controller, as there are no predefined OpenFlow rules for broadcast traffic. Figure 7 depicts the signaling and data forwarding process: First, AP2 generates the ARP request (1), and the corresponding vSwitch forwards it to Ryu (2a), which then instructs the vSwitch to flood the packet (2b). When the ARP request reaches the vSwitch in the Router, it is again sent to Ryu (3a) and subsequently flooded (3b). Finally, AP1 follows the same process, sending the request to Ryu (4a) before it is flooded (4b). This normal operation has been enhanced to update the pre-existing flow rule in the Router's

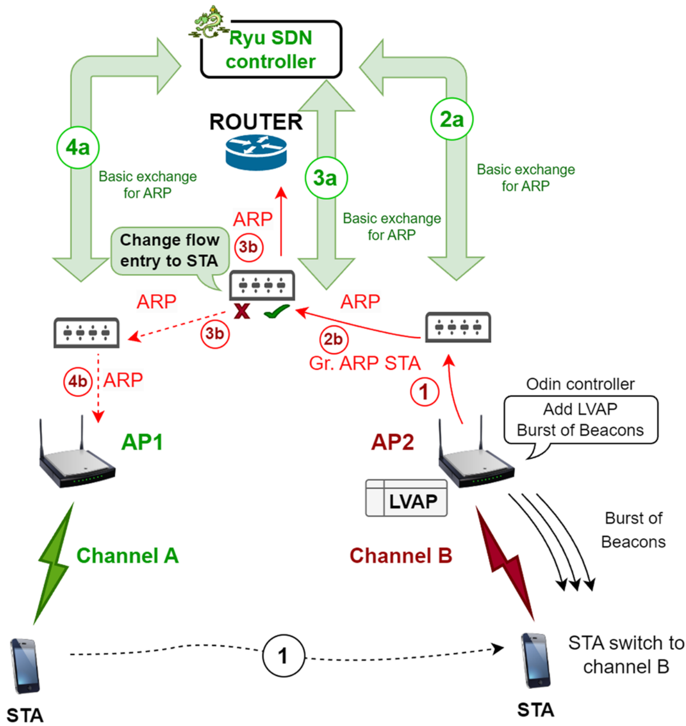

Fig. 7 Gratuitous ARP to update the downlink wired data path during handoff (Ryu messages in Fig. 8)

vSwitch, where a downlink data path has already been established but now needs to be updated. The detailed message exchanges and operations are shown in Fig. 8. Like normal broadcast packets, the gratuitous ARP is sent as data within an OFP_PACKET_IN message, including the match field to specify the input port and the STA's source MAC address. Under normal operation, new rules are not created; the SDN controller simply returns the packet in the corresponding OFP_PACKET_OUT message, instructing the vSwitch to flood it (FLOOD output port in the action field). However, OFP_PACKET_IN allows the identification of the STA's updated MAC address location (and therefore its new position) if a different input port is detected. In this case, the controller instructs the relevant vSwitch to proactively update its OpenFlow table with the new forwarding path. To achieve this, an OFP_FLOW_MOD message with the OFPFC_MODIFY command is sent, instructing the switch to update any existing OpenFlow rule that has the STA's MAC address as the destination by assigning a new output port. As a result, even in the absence of unicast traffic, the flow table is updated, thereby reducing handoff delays before data traffic forwarding resumes.

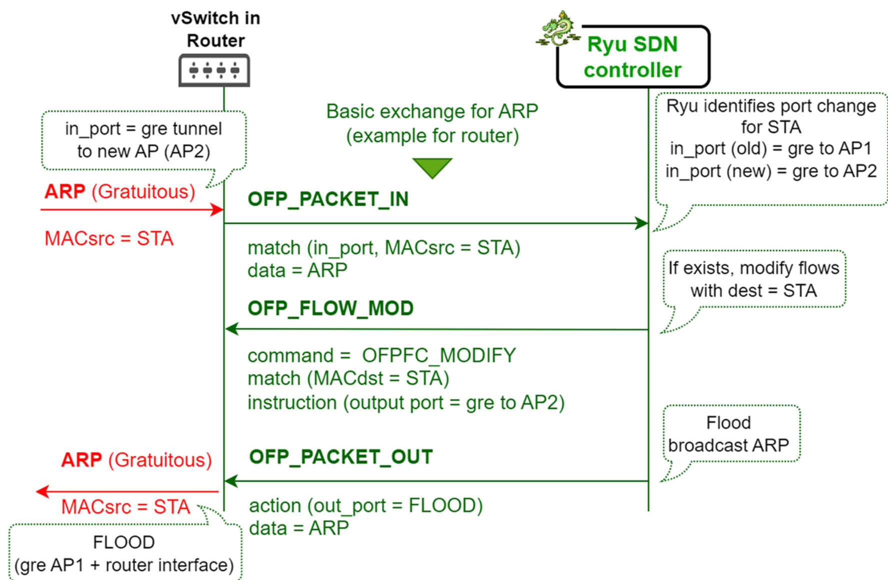

Fig. 8 OpenFlow messages with Ryu triggered by gratuitous ARP. Wired downlink data-path update in the router vSwitch

4 Kubernetes Deployment

Aligning with current trends in network deployment, a Cloud-native Network Function (CNF) approach is adopted in the design of the proposed SDWN architecture. Kubernetes is the natural choice for orchestrating network functions and applications deployed with Docker containers. This section outlines the design decisions made in this work for deploying the previously described SDWN architecture using a Kubernetes infrastructure. As detailed in Sect. 5, the Rancher K3s distribution–a lightweight Kubernetes variant– has been selected for its suitability in IoT and Edge environments. K3s simplifies both the configuration and usage of the tool, making it an ideal choice for this study's work environment as a non-production-oriented proof of concept. Figure 9 illustrates the basic architecture of a K3s cluster (depicted here with two nodes), where the master node, responsible for the control plane, can also function as a worker node and run containerized applications. Additionally, it demonstrates the integration of a Docker registry as an image repository for running containers. Kubernetes' control plane autonomously schedules and manages the lifecycle of applications according to user-defined specifications, provided through the user interface via configuration files (.yaml) and kubectl commands.

On the other hand, Fig. 10 shows the basic deployment of the proposed SDWN architecture in a 3-node Kubernetes cluster including all the required Kubernetes Objects (Pods, Services and ConfigMaps) that are described next. This figure serves as a reference for the proposal described in this work, illustrating the developed solutions and defining the proof-of-concept scenario considered later.

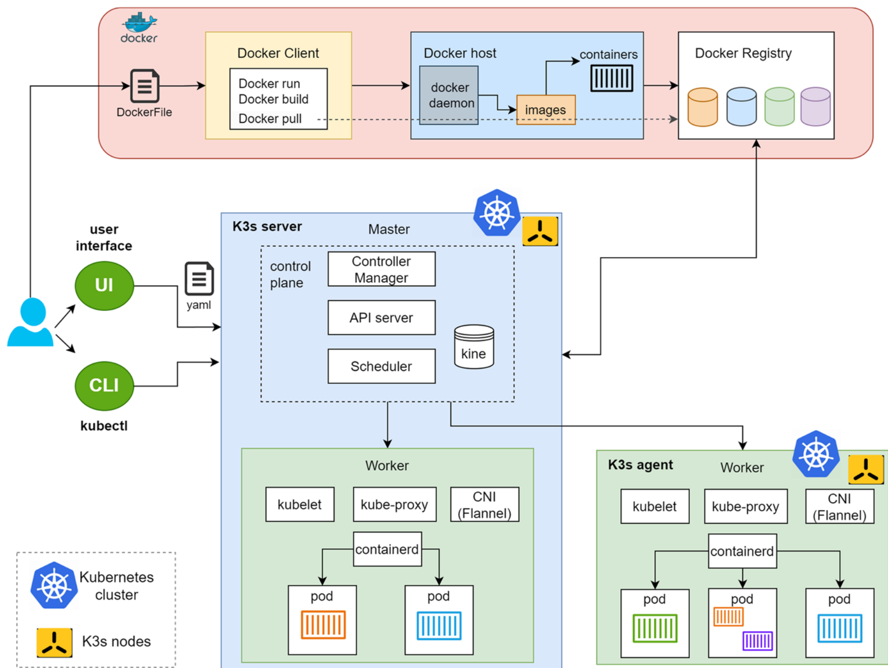

Fig. 9 Kubernetes architecture (K3s distribution) and integration with Docker registry

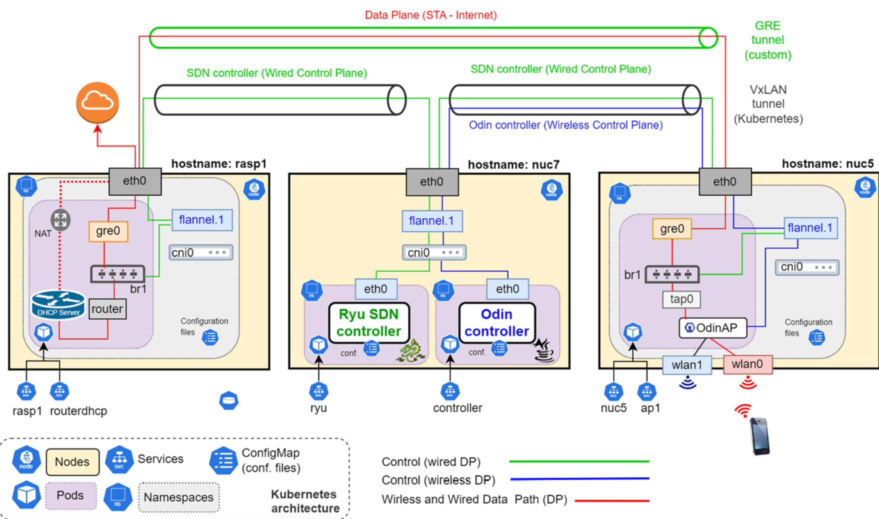

Fig. 10 Deployment of the SDWN architecture in a 3-node Kubernetes cluster

4.1 Docker Images for Containers

The control and data plane functionalities described in Sect. 3 are implemented to run as Docker containers. These containers are built from Docker images tailored to the specific requirements of the custom-developed software. To achieve this, multiple Dockerfiles are created, each containing the necessary instructions for building the source code (see red box in Fig. 9). Additionally, to ensure compatibility across different types of hardware (e.g., Ubuntu Linux servers or Raspberry Pi), a multi-platform build is performed, packaging variants for both ARM64 and AMD64 architectures within the same image. The resulting images are pushed to a Docker Hub repository (cenit1/newlanuz) to facilitate their retrieval by Kubernetes:

1. **cenit1/newlanuz:ap:** It contains both the Open vSwitch (OVS) framework for managing a virtual switch (br1) in the wired data path and the Odin Agent, a standalone user-space application called OdinAP. OdinAP, written in C, implements standard AP functionality (similar to hostapd) along with custom-developed control features, such as LVAP management and monitoring. The setup of the OVS bridge and the initiation of OdinAP are triggered by a shell script (start.sh). Additionally, a configuration file (conf.cnf) is required to parameterize OdinAP's execution. These input files will be handled as Kubernetes ConfigMap objects, as will be explained later in section 4.2.3.
2. **cenit1/newlanuz:routerdhcp:** Using ISC DHCP software and Netfilter iptables, it implements both DHCP server functionality and NAT translation. Like the AP, it includes the Open vSwitch (OVS) framework for managing a virtual switch (br1) in the wired data path. The setup of the OVS bridge, along with the initiation of the DHCP daemon and the configuration of iptables rules, are all triggered by a shell script (router-start.sh), which is managed as a Kubernetes ConfigMap object.
3. **cenit1/newlanuz:ryu**: Starting with a Docker base image that includes the Ryu software, additional dependencies are installed to incorporate specific Python libraries, such as dnspython, for handling DNS queries in Kubernetes services. A custom Ryu application, built on top of the existing SimpleSwitch code provided by the Ryu framework, is responsible for establishing GRE tunnels with OVSDB during deployment and triggering the creation of OpenFlow rules in response to STA associations and handovers. This application is not included in the Docker image but is provided as an input file, managed as a Kubernetes ConfigMap object, to be executed as the primary Ryu application.
4. **cenit1/newlanuz:register:** An additional Python-based server, co-located with the Odin Controller, enables APs to register as authorized Odin Agents. This registration mechanism, absent in the original SDWN architecture, has been introduced to provide flexible authorization of Odin Agents with minimal modifications to the Odin Controller. Upon activation, Odin Agents register using their dynamically assigned IP addresses, based on the specific deployment. In turn, the Odin Controller can access this register to monitor and manage the activation of the agents. With real-time registration, the allocated addresses do not need to be predetermined, simplifying the automated deployment process.

5. **cenit1/newlanuz:controller:** Developed in Java, it manages radio functionalities by communicating with the APs through the Odin protocol, enabling the allocation and movement of LVAPs associated with STAs, gathering statistics, and receiving northbound requests from developed applications. An input configuration file (poolfile) specifying the selected applications is again managed as a Kubernetes ConfigMap object.

4.2 Pods and Services

Kubernetes deploys containers within its fundamental compute unit, known as a Pod (purple boxes in Fig. 10). Although a Pod can run multiple containers, this approach considers one container per Pod, making the process of running a container equivalent to running a Pod. Pods are deployed within the Kubernetes Cluster, which comprises the nodes forming the computing infrastructure (K3s server and K3s agent in Fig. 9).

Kubernetes offers multiple options for defining how containers are executed within a Pod. In this section, we outline the design choices that are suitable for our scenario:

4.2.1 Assigning Pods to Selected Nodes

The Kubernetes scheduler automatically assigns Pods to worker nodes (green boxes in Fig. 9) based on available resources. However, there are several ways to restrict a Pod to specific nodes. One method is by using the **nodeName field** in the Pod specification (.yaml file). When the nodeName field is set, the scheduler bypasses the Pod, and the kubelet on the designated node attempts to place the Pod on that node. In the SDWN scenario, this approach is useful for positioning APs according to the planned coverage, while also allowing for their dynamic activation or deactivation based on demand. Conversely, for Router/DHCP and controller applications, the initial placement can be more flexible but may require adjustments to meet latency or other performance-related criteria. As an example, Fig. 11 describes the configuration parameters for scheduling an AP, including the mandatory location in a specific node. In the reference deployment shown in Fig. 10, the Router and AP Pods operate on separate nodes as required, while both Odin and SDN controllers can be hosted on the same compute node.

4.2.2 Host Network and Security Context

In Kubernetes, the **hostNetwork** is a configuration option, when set to true, that allows a container to share the network namespace of its host node. When hostNetwork is enabled for a Pod, it means that the Pod's containers will use the networking stack of the underlying host machine instead of having their own isolated network stack. Although regular containerized applications are typically isolated within their own network namespaces, certain network functions may require access to the host's networking stack. This is the case for APs, which need direct access to the wireless card, or for virtual routers and address translators, which rely on forwarding capabilities and iptables rules. This is illustrated in Fig. 10, where the Pod running the AP or

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: Pod-ap1
  labels:
    app: ap1
    loc: nuc6
spec:
  # mandatory placement
  nodeName: nuc6
  hostNetwork: true
  dnsPolicy: ClusterFirstWithHostNet
  containers:
  - name: mi-ap1
    image: cenit1/newlanuz:ap
    imagePullPolicy: Always
    securityContext:
      privileged: true
    env:
    - name: start_script
      valueFrom:
        configMapKeyRef:
          name: my-config-start
          key: start.sh
    - name: stop_script
      valueFrom:
        configMapKeyRef:
          name: my-config-stop
          key: stop.sh
    volumeMounts:
    - name: volumen-config-ap1
      mountPath: /OdinAPLVAP/conf.cnf
      subPath: conf.cnf
    command:
      - "/bin/sh"
      - "-c"
      - |
        sleep 5
        bash -c "$start_script" & \
        sleep infinity
  initContainers:
  - name: init-dhcp
    image: busybox:1.28
    command: ['sh', '-c', "until nslookup
↪   routerdhcp.default.svc.cluster.local; do echo waiting for myservice; sleep
↪   2; done"]
  - name: wait-server
    image: busybox:1.28
    command: ['sh', '-c', "until nslookup register.default.svc.cluster.local;
↪   do echo waiting for myservice; sleep 2; done"]
  - name: post-ip
    image: cenit1/newlanuz:regv0
    imagePullPolicy: Always
    command: ["sh", "-c", "python ./register-client.py pyser 12000 POST"]
    workingDir: /pyapps
    volumeMounts:
    - name: vol-config-pycli
      mountPath: /pyapps/register-client.py
      subPath: register-client.py
  volumes:
  - name: volumen-config-ap1
    configMap:
      name: my-config-ap1
  - name: vol-config-pycli
    configMap:
      name: my-config-pycli
```

Fig. 11 Description file for Pod running an AP in node nuc6: Pod-ap1.yaml

the Router (represented by the purple box in both the rasp1 and nuc5 nodes) shares the network namespace of the host node (gray box). In contrast, the Ryu and Odin controllers are isolated within their own network namespace (depicted as a purple box overlapping with the gray box). Additionally, managing network interfaces or accessing filtering and translation rules requires not only sharing the network stack but also granting privileged access to the containers. Kubernetes allows this security to be managed through the Security Context in the Pod specifications. To achieve this, containers running the APs and the Router are configured with **.spec.Context. privileged: true**, as shown in the example in Fig. 11.

4.2.3 Flexibly Introducing Configuration Parameters

A Kubernetes ConfigMap is an API object used to store non-confidential data in key-value pairs. Pods can consume **ConfigMaps** as environment variables, command-line arguments, or as configuration files in a volume. A ConfigMap enables decoupling the environment-specific configuration from container images, making applications easily portable.

In our SDWN scenario, the various containerized applications described in Sect. 4.1 require specific configuration files to define parameters such as radio capabilities and frequency channels for APs, Python code for the Ryu-based SDN controller, and more. All these files are managed via ConfigMaps and Volumes, as shown in the examples in Figs. 11 and 12.

4.2.4 Services and Domain Name Resolution

In Kubernetes, a Service is a method for exposing a network application that is running as one or more Pods in your cluster. Kubernetes creates DNS records for these Services so they can be accessed with consistent DNS names instead of IP addresses.

Although the implemented network functions are not applications intended to be exposed via services, they can still benefit from Kubernetes' domain name resolution. For instance, the APs need to know the IP addresses of the SDN and Odin controllers to establish connections. Similarly, the SDN controller requires the IP addresses of the nodes hosting the Pods running the APs and the Router to identify the GRE tunnel endpoints for the wired data path. By querying Kubernetes DNS for service names, deployment can be automated without needing prior knowledge of the specific Kubernetes CIDR (control plane) or the physical infrastructure addressing for the GRE tunnels.

The proposed architecture adopts **Headless Services** (defined by **.spec.type: ClusterIP: None**). Unlike regular services, Kubernetes does not perform load balancing or proxying for Headless Services. Instead, they provide the individual Pod IP addresses as endpoints through internal DNS records, managed by the cluster's DNS service. Moreover, for containers running with hostNetwork enabled, the IP address corresponds to the node hosting the Pod, which, as explained, is beneficial for establishing the data path.

Additionally, by using **initContainers** (specialized containers that run before app containers in a Pod), it is possible to define a specific activation sequence, providing

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: Pod-ryu
  labels:
    app: ryu
spec:
  # optional specific placement
  nodeName: nuc7
  dnsPolicy: ClusterFirst
  containers:
    - name: mi-ryu
      image: cenit1/newlanuz:ryu
      imagePullPolicy: Always
      command:
      - "/bin/bash"
      - "-c"
      - |
         sleep 5
         ryu-manager ryu.app.SDWN_switch && \
         sleep infinity
      workingDir: /root/ryu
      volumeMounts:
      - name: config-volume
        mountPath: /usr/lib/python3/dist-packages/ryu/app/SDWN.py
        subPath: SDWN.py
  initContainers:
  - name: wait-router
    image: busybox:1.28
    command: ['sh', '-c', "until nslookup
↪   routerdhcp.default.svc.cluster.local; do echo waiting for myservice; sleep
↪   2; done"]
   volumes:
  - name: config-volume
    configMap:
      name: my-config-ryu
```

Fig. 12 Description file for Pod running SDN controller: Pod-ryu.yaml

fine-grained control over the automated deployment process. Because init containers run to completion before any app containers start, init containers offer a mechanism to block or delay app container startup until a set of preconditions are met. A Service is created when the application container in the corresponding Pod starts running, which ensures that the corresponding domain name lookup succeeds. Thus, by waiting for the activation of the service of a first Pod (nslookup loop), we can execute the application of a second one in order.

In our SDWN deployment, a specific activation order is essential to ensure the proper functioning of all components. The Odin Agents must start and register in the system before the controller initializes, allowing the latter to properly activate selected applications, such as Mobility Management for seamless handover. Additionally, to enable full data path activation –including the wired segment upon STA association– GRE tunnels over the physical infrastructure must be preemptively established in the corresponding vSwitches. As a result, the required activation order is: i) Router/DHCP, ii) Register for Odin Agents and Ryu SDN Controller, iii) any AP and iv) Odin Controller.

Services are associated with Pods using label selectors. In the SDWN scenario, two distinct labels are defined: the **app** label, which identifies the Pods running specific container applications (such as the Router, Odin controller, or Ryu controller),

and the **loc** label, which identifies the nodes hosting Pods with hostNetwork enabled (APs and Router). When both the application and the hosting node are relevant for the same Pod (e.g. the Router), both labels can be included in the Pod's definition without causing any conflicts.

As a result, the defined services are:

- **routerdhcp.default.svc.cluster.local**: A service to respond to DNS queries with the IP address of the Pod running the default Router and DHCP server, which corresponds to the IP Address of the node hosting the Pod (hostNetwork: true). It is associated with an **app label** selector (routerdhcp).
- **ryu.default.svc.cluster.local**: A service to respond to DNS queries with the IP address of the SDN controller, which corresponds to the IP Address from the Pod network CIDR allocated by Kubernetes. It is assoctiated with an **app label** selector (ryu).
- **register.default.svc.cluster.local**: A service to respond to DNS queries with the IP address of the Register, which corresponds to the IP Address from the Pod network CIDR allocated by Kubernetes. When configuring the Pod specification, this Pod must be co-located with the Odin controller. It is assoctiated with an **app label** selector (register).
- **controller.default.svc.cluster.local**: A service to respond to DNS queries with the IP address of the Odin controller, which corresponds to the IP Address from the Pod network CIDR allocated by Kubernetes. It is assoctiated with an **app label** selector (controller).
- **<node-name>.default.svc.cluster.local**, for any node in the Kubernetes Cluster that host an AP or the Router. The corresponding DNS Response provides the IP address of the node (hostNetwork: true). It is associated with a **loc label** selector (<node-name>). Its usage is essential for automating the creation of GRE tunnels upon activation of the virtual switches (br1), enabling the SDN controller to resolve domain name lookups into the IP addresses of the physical nodes hosting each vSwitch.
- **ap1.default.svc.cluster.local**, for any AP in the system. This service is necessary to allow the Odin Controller to wait for any registered Odin Agent, as intended in the original SDWN architecture.

Figure 13 shows two example.yaml files defining the services (i) for the Pod running the Router/DHCP (routerdhcp), using the 'app' selector and (ii) the nuc6 node hosting any Pod, using the 'loc' selector. In addition, the Pod description file in Fig. 12 includes the use of an initContainer to wait for the Router activation by resolving the domain name of the routerdhcp service identified with the corresponding 'app' selector.

4.3 Network Model

Deploying virtualized functions with Kubernetes requires careful attention to connectivity to ensure seamless interaction between the components. Kubernetes uses its own network model, allowing direct communication between Pods within a Cluster,

Fig. 13 Description file for service router-dhcp (selector 'app') and node nuc5 (selector 'loc'): service-routerdhcp.yaml and service-nuc6.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: routerdhcp
spec:
  selector:
    app: routerdhcp
  clusterIP: None
  type: ClusterIP

apiVersion: v1
kind: Service
metadata:
  name: nuc6
spec:
  selector:
    loc: nuc6
  clusterIP: None
  type: ClusterIP
```

as well as between nodes and the Pods running on them, without requiring NAT. A Container Network Interface (CNI) plugin is required to implement the Kubernetes network model. A CNI plugin compatible with the cluster and suited to specific needs must be used. Various plugins, both open-source and closed-source, are available within the broader Kubernetes ecosystem.

While this model is effective for standard application deployments, where the primary goals include accessibility, replicability, and load balancing (e.g., a web service handling fluctuating demand levels), deploying virtual network functions may require another approach. This involves assessing the feasibility of the standard configuration and identifying the adaptations needed to meet specific connectivity requirements. With this in mind, a detailed analysis of the control and data planes has been performed, as described below.

Control plane communications between Odin Agents on the APs and the Odin Controller are seamless to the user, requiring only proper IP addressing and routing. Kubernetes ensures this through any available CNI, such as the **default Flannel CNI plugin** in the K3s distribution. Flannel's CNI uses VxLAN (Virtual Extensible Local Area Network) to encapsulate Pod network traffic, enabling seamless communication across nodes by creating an overlay network. However, the **data path** from the STAs, through the custom-developed APs, and across the virtual switches to the Router in the overlay E-LAN model, does not align with any existing CNI specifications, requiring a custom configuration. Additionally, the current data path may need future modifications and adaptations for new use cases, demanding greater flexibility and control without relying on the Kubernetes network model. For this reason, the implementation of the SDN controller and the corresponding virtual switches within the APs play a crucial role in this specific scenario, becoming a **defining feature** of the proposed architecture.

As shown in the reference Fig. 10, communications within the control plane utilize the default Flannel CNI provided by K3s. Ryu and Odin, as isolated Pods, leverage their respective eth0 interfaces (and the IP addresses allocated from the CIDR range). In contrast, the AP and the Router, residing in the network namespace of the hosting physical node, do not have an eth0 interface with an assigned IP address. Instead, they directly use the flannel.1 interface of the node. The figure also outlines the wireless and wired data path from the STA to the Router. For example, uplink

traffic traverses the *wlan0* interface on the serving AP, where the OdinAP process forwards it to the wired segment through the internal *tap0* interface. Leveraging the configured flow rules in the virtual switches (br1) on both the AP and the Router, the traffic flows through the GRE tunnel and reaches the Router's internal interface, which provides Internet connectivity. It is important to note that the tunnel is established before traffic follows the data path, thanks to the automation provided by Ryu upon vSwitch activation (initial handshake described in Sect. 3.2.2a). By identifying the nodes hosting the corresponding br1 vSwitches and retrieving the names of the related Kubernetes services (those matching the label 'loc' nodename), the IP addresses of the GRE endpoints can be determined.

4.4 Automatic Deployment

The first step in deploying the SDWN architecture is to set up a Kubernetes cluster. This involves installing the master node and the required agents to host potential deployment Pods. To enable a lightweight deployment on both large servers and resource-constrained hardware, such as Raspberry Pi, the K3s distribution is utilized. To minimize the number of nodes in the infrastructure, the master node not only handles control functions but can also serve as a compute node. By default, nodes hosting APs run only AP-related Pods, while NUCs, virtual machines, or servers can host both Router and controller Pods. Due to the required hostNetwork setting for Router and APs, these Pods cannot be scheduled on the same node. Additionally, APs must be scheduled according to the coverage plan, requiring the use of the appropriate selector (nodeName) in the Pod definition. This planning also involves configuring specific wireless parameters, such as channel allocation for each AP, registering applications with the Odin controller (e.g., the MobilityManager for seamless handovers), and setting up private addressing and address translation rules within the internal E-LAN network, including DHCP configuration and iptables masquerading.

Leveraging Kubernetes' management and control planes, the whole process can be easily automated. Once the cluster is installed in the infrastructure and configuration files are created, the required Kubernetes objects (ConfigMaps, Services and Pods) can be created and scheduled. Within a deployment directory containing all the necessary files (e.g.,.yaml files and source files for ConfigMaps), a deployment script can quickly configure the desired scenario by executing the following kubectl commands:

1. Creating ConfigMap objects for input configuration files (example):
   `kubectl create configmap my-config-ap1 –from-file=./configs/ap1/conf.cnf`
2. Creating Headless Services to associate 'A' DNS Resource Records to the IP addresses of the required Pods (example):
   `kubectl apply -f service/my-service-routerdhcp.yaml`
3. Create and scheduling the Pods (example):
   `kubectl apply -f Pods/Pod-ap1.yaml`

As previously mentioned, several example.yaml files illustrate the proper configuration of Pods and Services. These include settings for nodeName, ConfigMaps, Volumes, and initContainers to ensure the correct activation order, as well as selectors and labels for the required services. Figure 11 shows the description file for a Pod running an AP, whereas Fig. 12 shows the equivalent for a Pod running the SDN Controller. Additionally, Fig. 13 provides the configuration files for services related to DNS records for the SDN controller (ryu) and the physical node hosting the AP (nuc6).

Regarding the activation order described in Sect. 4.2.4 by using initContainers, Fig. 11 illustrates how initContainers in an AP ensure the correct activation sequence. First, the process waits for the routerdhcp service to start, ensuring that the GRE tunnel from the vSwitch is properly created. Next, it waits for the Odin Agent registration to be active, guaranteeing that, according to the third initContainer, the agent registers correctly. Finally, once these prerequisites are met, the main application container running the OdinAP software can initialize properly. In addition, Fig. 12 illustrates how the SDN controller waits for the Router activation, ensuring its corresponding vSwitch is the first to interact with the controller. Thus, once the APs activate, the GRE tunnels towards the Router can be created. Following the same strategy, the Pod running the Odin Controller requires two initContainers: one to wait for the activation of the Odin Agents (by performing an nslookup for the corresponding AP services, such as ap1, ap2, etc.), and another to verify their registration in the system (by properly querying the Register).

5 Use Case Analysis

The primary contribution of this work is the container-based redesign of the SDWN architecture and its seamless integration into Kubernetes, enabling automated deployment while preserving the previously demonstrated functionality. To validate the proposal, a laboratory framework has been set up as a proof of concept, allowing legacy terminals to connect to the Internet. Several tests have been conducted to analyze the accuracy of the newly designed procedures, primarily aimed at ensuring data path maintenance across both wireless and wired segments. To this end, two fundamental use cases have been considered: the establishment of the data path upon terminal association and the seamless handover triggered by mobility.

5.1 Scenario

To evaluate the deployed architecture, a 4-node Kubernetes cluster was set up using an infrastructure comprising three Intel NUCs and a Raspberry Pi 3, as shown in Fig. 14. The K3s cluster follows the same configuration as Fig. 10, with an additional AP. The K3s master node is hosted on one of the NUCs (hostname:nuc7), which is also capable of running application Pods. The nodeName attribute is used to assign specific locations for each Pod. The two APs are deployed in Pods on two separate NUCs (hostname:nuc6 and hostname:nuc5), while both Odin and Ryu-based controllers are scheduled on the master-node NUC (hostname:nuc7). The Raspberry Pi

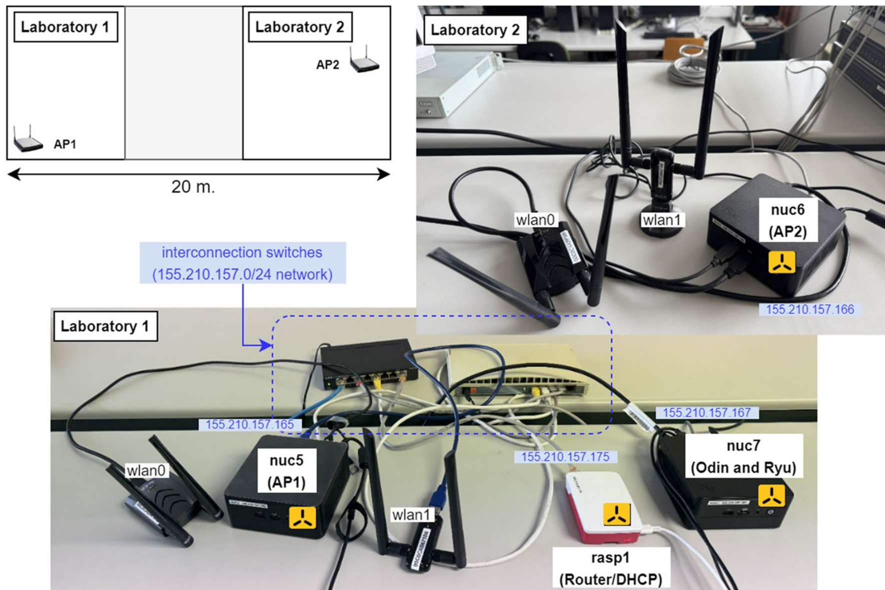

Fig. 14 Laboratory setup for the SDWN deployment. K3s nodes in two separate laboratories. Seamles handover triggered by mobility between labs

(hostname:rasp1) is responsible for hosting the Router and DHCP server. Table 1 summarizes the detailed configuration of the Pods running the SDWN functionalities in this scenario, including the necessary Docker images, Services, and ConfigMaps. It also outlines the activation order, as determined by the required InitContainers, to ensure proper operation. Table 2 presents the network configuration for both the physical infrastructure (Kubernetes nodes) and the SDWN architecture. The control plane utilizes the default Flannel network, where pods sharing the host network communicate through the flannel.1 interface, while isolated pods use the eth0 interface.

Meanwhile, the E-LAN corresponding to the data plane, which connects STAs to their Router, is configured via DHCP within the 192.168.137.0-131/24 range. The established GRE tunnels encapsulate this traffic, linking the OpenFlow ports shown in the figure.

Although this laboratory deployment was designed as a proof of concept for the containerized version of the SDWN Architecture, its ability to provide connectivity to standardized terminals, such as mobile phones and laptops, ensures a realistic validation of its functionality. Accordingly, the ongoing results stem from the interaction of an Apple laptop operating as an STA and connected to the configured WiFi network. The in-depth evaluation of the proposal, presented in the following section, focuses on its ability to sustain the complete data path, with particular emphasis on the newly introduced tunnel-based wired segment, during a seamless handover.

Table 1 Summarized parameters for pods and services in the Kubernetes-based SDWN deployment

Pod [1]	Container[2]	Docker image	Services (label)	ConfigMap (files)	Usage
pod-router-dhcp	my-router (app) (1)	cenit1/newlanuz:routerdhcp	routerdhcp (app)	router-start.sh	Router/DHCP
nodeName: rasp1	–	–	rasp1 (loc)	router-stop.sh	–
pod-ryu	init-dhcp (init) (1)	busybox:1.28	–	–	wait for Router
nodeName: nuc7	my-ryu (app) (2)	cenit1/newlanuz:ryu	ryu (app)	SDWN.py	SDN Controller
pod-register	my-reg (app) (1)	cenit1/newlanuz:register	register (app)	register-server.py	Register for Odin Agents
nodeName: nuc7	–	–	–	–	–
pod-ap1/pod-ap2	init-dhcp (init) (1)	busybox:1.28		–	wait for Router
nodeName: nuc5/nuc6	wait-register (init) (2)	busybox:1.28	–	–	wait for Register
	post-ip (init) (3)	cenit1/newlanuz:register	–	register-client.py	POST IP address
	my-ap1 (app) (4)	cenit1/newlanuz:ap	ap1/ap2(app)	start.sh/stop.sh	AP
	–	–	nuc5/nuc6 (loc)	conf.cnf	–
pod-controller	init-aps (init) (1)	busybox:1.28	–	–	wait for ap1 and ap2
nodeName: nuc7	get-ip (init) (5)	cenit1/newlanuz:register	–	register-client.py	GET IPs for agents
	my-controller (app) (6)	cenit1/newlanuz:controller	controller (app)	poolfile	Odin Controller

[1] nodeName specification included to schedule Pods in selected nodes

[2] activation order in parenthesis

5.2 Use Case: Data Path Establishment After Association

To validate the functionality of the SDWN architecture once deployed in Kubernetes, we tested the basic association of a legacy STA. This evaluation focuses not on the precise timing of the procedures but rather on ensuring the correct configuration of the STA and the successful acquisition of connectivity. Figure 15 illustrates the complete process by which an Apple laptop acquires Internet connectivity through its WiFi card. The figure is based on a combination of Wireshark traces captured from the wireless interface of AP1 and the transport network. The latter includes VxLAN-tunneled traffic encapsulating both Odin protocol and OpenFlow messages, as well as GRE-tunneled data traffic.

As explained in Sect. 3.2.1a, the Odin Agent in the AP interacts with the Odin Controller during the association process to register the STA in the system, creating the appropriate LVAP (802.11 signaling and Odin Protocol exchange, as shown in the upper part of the figure). Initially, the LVAP does not include an IP address for the STA until the TCP/IP configuration via DHCP is completed. This process occurs within the wired data path, traversing the previously established GRE tunnels, but requires the intervention of the SDN Controller –always necessary for broadcast packets and until OpenFlow rules are established for unicast packets.

The uplink DHCP messages generated by the STA (DHCP Discover and DHCP Request) are broadcast. As a result, both virtual switches (br1) in AP1 and the Router forward them to Ryu, which then instructs their flooding using `OFP_PACKET_OUT` messages with the FLOOD action.

Conversely, the downlink DHCP Offer and DHCP ACK messages are unicast. When the DHCP Offer is sent to Ryu –first from the Router and then from AP1– it triggers the creation of the Downlink OpenFlow rule. This occurs in two stages:

- In the Router, the rule matches the internal input port *router*, the Router's MAC_{src}, and the STA's MAC_{dst}, instructing the output port to forward traffic through the GRE tunnel to AP1.
- In AP1, the rule matches the input port of the GRE tunnel from the Router, the Router's MAC_{src}, and the STA's MAC_{dst}, forwarding the traffic to the internal *tap0* interface.

Once the OpenFlow rules are created after the DHCP Offer, the DHCP ACK is forwarded directly without being sent to Ryu. When this message reaches the AP, the Odin Agent announces the assigned IP address to the Odin Controller (as shown in the 'DHCP capture' in Fig. 15), completing the LVAP with the 'set LVAP' Odin command.

Once the STA has a valid TCP/IP configuration –including its IP address and other network parameters such as the domain name server and default gateway– IP data traffic from the Apple laptop is observed in the captures. Regarding the data path, as previously explained, the downlink OpenFlow rules are created with the DHCP Offer message, but no valid unicast traffic has been transmitted yet. At this point, when the first unicast uplink IP datagram is generated, the interaction with Ryu triggers the cre-

Table 2 Network parameters (Kubernetes cluster)

Infrastructure network	155.210.157.0/24
K3s agent (AP1) [nuc5]	155.210.157.165
K3s agent (AP2) [nuc6]	155.210.157.166
K3s master (controllers) [nuc7]	155.210.157.167
K3s agent (Router)	155.210.157.175 [rasp1]
Gateway to Internet	155.210.157.254
IP addresses allocated in Pods	Flannel CIDR 10.42.0.0/16
AP1 (hostNetwork=true)	10.42.1.0 (flannel.1 interface)
AP2 (hostNetwork=true)	10.42.3.0 (flannel.1 interface)
Odin controller (isolated Pod)	10.42.0.167 (eth0)
Ryu controller (isolated Pod)	10.42.0.169 (eth0)
Router (hostNetwork=true)	10.42.6.0 (flannel.1 interface)
Configured GRE tunnels	
Open vSwitch ‘br1’ in AP1	
Interface gre to Router	OpenFlow port = 10
Open vSwitch ‘br1’ in AP2	
Interface gre to Router	OpenFlow port = 10
Open vSwitch ‘br1’ in Router	
Interface gre to AP1	OpenFlow port = 12
Interface gre to AP2	OpenFlow port = 13
Network addressing in the E-LAN (DHCP)	
default Router	192.168.137.131
Address range for STAs	192.198.137.0-130/24
domain name server	8.8.8.8

ation of the uplink OpenFlow rules. This is exemplified by the DNS Query observed in the captures, illustrated at the bottom of Fig. 15.

Similar to the downlink OpenFlow rule creation, this process occurs in two stages, as the messages first traverse AP1 and then the Router:

- In AP1, the rule matches the input port of internal *tap0* interface from the OdinAP, the STA’s MAC_{src}, and the Router’s MAC_{dst}, forwarding the traffic to the GRE tunnel to the Router.
- In the Router, the rule matches the GRE tunnel from the AP1, the STA’s MAC_{src}, and the Router’s MAC_{dst}, instructing the output port to forward traffic through the internal interface *router*.

5.3 Use Case: Seamless Handover Triggered by Mobility

To demonstrate the flexibility of the proposal in dynamically adapting the data path to mobility across both wireless and wired segments, a handover test is performed. An STA (the Apple laptop) is initially associated with one AP (AP1, in channel 149) but, due to mobility, needs to transition to a new AP (AP2, in channel 157). To assess the

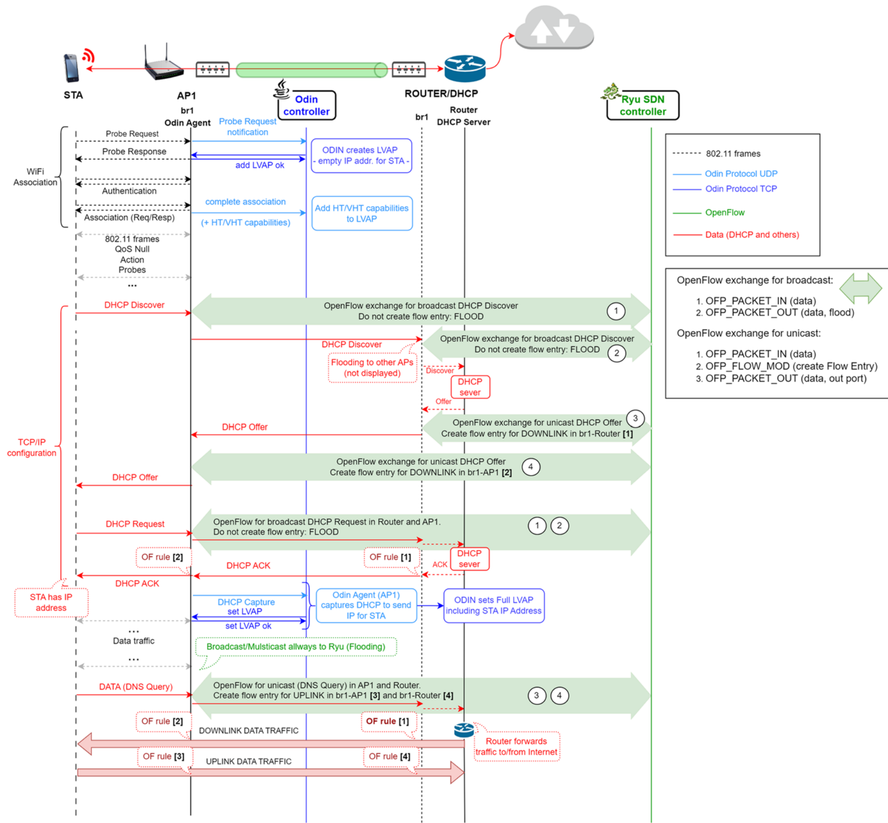

Fig. 15 Association process and TCP/IP configuration for an STA: LVAP management and wired data path establishment through OpenFlow rules (downlink and uplink)

impact on the data path, controlled downlink or uplink traffic is generated between the Router and the STA using iperf3 with the server (-s argument) being the Router or the STA, respectively:

- `iperf3 -s`
- `iperf3 -c` <@IP$_{server}$>`-l 400 -t 3600 -b 1 M -u`

This corresponds to UDP traffic of 1 Mbps with a packet size of 400 bytes and an inter-packet interval of 3.125 ms.

Previous analyses validated the effectiveness of the original SDWN architecture in reducing handoff delays [4, 5]. However, those results primarily focused on the advanced functionalities developed in the Odin Controller (such as proactive handoff and load balancing) and on analyzing the wireless data path, including LVAP management, channel switching, and wireless configuration parameters. Table 3 summarizes the key configuration parameters considered in the tests, particularly those related to the observed delays in channel switching.

Table 3 WiFi configuration parameters (802.11ac, 5 GHz)

Default beacon interval	100 ms
Unicast Beacon interval with CSA	2 ms
Number of CSA Beacons	3
Number of Unicast Beacons	3
Unicast Beacon interval in handoff	2 ms
Number of Gratuitous ARP after handover	1
Channel in AP 1	149
Channel in AP 2	157

Fig. 16 Handoff process during downlink traffic. UDP iperf3 packets from router (client) to STA (server)

In contrast, this study demonstrates how the novel Kubernetes deployment preserves connectivity during handover while keeping the delay within bounds. Particular emphasis is placed on evaluating the reconfiguration of the wired data path, leveraging the control plane features implemented in the Ryu-based SDN controller.

The following figures illustrate the key signaling events during the handoff process, encompassing both the wireless and wired data paths, while focusing on measuring the timing related to the re-establishment of the wired data path. The figures are derived from the combination of observed Wireshark traces captured on the wireless interfaces of AP1 and AP2, as well as within the transport network. The latter includes encapsulated traffic in both VxLAN and GRE tunnels. VxLAN carries control traffic transmitted through the Kubernetes Flannel network (Odin –TCP/UDP– and Ryu signaling –OpenFlow), while GRE transports data traffic (ARP and iperf3). Figure 16 illustrates the signaling and data traffic during handoff in a down-

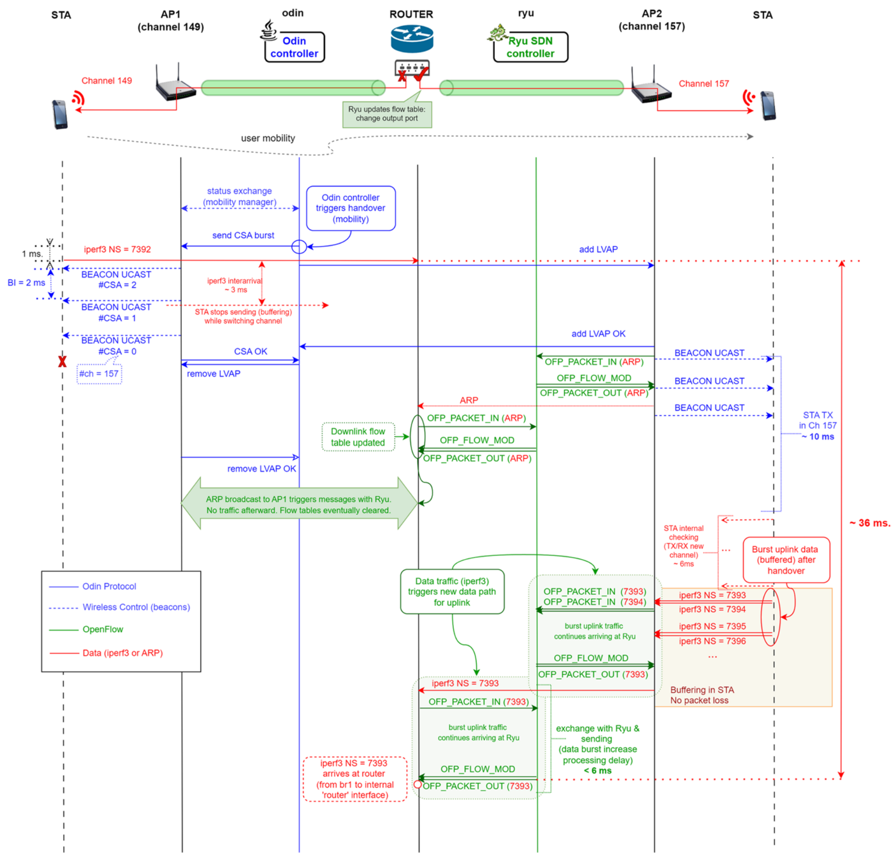

Fig. 17 Handoff process during uplink traffic. UDP iperf3 packets from STA (client) to router (server)

link transmission (with the STA acting as the iperf3 server), whereas Fig. 17 presents the equivalent scenario during handoff in an uplink transmission (with the STA acting as the iperf3 client).

The signaling messages (Odin protocol and OpenFlow) depicted in these figures follow the procedures outlined in both Figs. 5 and 7, as directly observed in the captured traffic. The primary differences noted relate to the direction of data traffic, which directly impacts the updates to the OpenFlow tables in the respective data path, as will be detailed subsequently.

The wireless handoff delay is primarily determined by the STA's channel switching delay, which depends on the wireless cards and the configured AP parameters (such as the Beacon Interval and the number of transmitted CSAs). Consequently, the analysis centers on evaluating the interaction between Odin and Ryu procedures and their impact on data forwarding.

The time values shown in the figures are not intended to have statistical significance, but rather to illustrate the correct functioning of the implemented deployment.

Figure 16 shows the handoff process during downlink data traffic. In this particular case, a handoff delay of 21 ms. has been measured, with 5 iperf3 packets lost. These values are specific to this scenario, but illustrate the involved events explained next.

As it can be observed, the process begins with the Odin controller signaling AP1 to send a Channel Switch Announcement to the STA via unicast beacons, while simultaneously triggering the migration of the LVAP to AP2 using the 'add LVAP' command. Once the CSA signaling is complete and the LVAP has been added to the new AP, it is removed from AP1, registering the STA with AP2.

When the AP2 correctly adds the LVAP, it starts sending unicast beacons to announce its presence to the STA, in order to speed the switching without waiting a regular broadcast beacon. At the same time, to speed the data path relocation, it sends a gratuitous ARP on behalf of the STA to trigger the OpenFlow messages exchange from the internal virtual switches along the wired path, which aid the Ryu controller to identify a downlink port change in the OpenFlow tables. The specific message exchange, as detailed in Fig. 8, includes an `OFP_PACKET_IN` message carrying the ARP request and the Ryu responses: an `OFP_FLOW_MOD` message to instruct the switch to modify its flow table, and an `OFP_PACKET_OUT` message to return the ARP request to be sent by the switch. As with any broadcast packet, no entries for unicast flows are added. In this case, the `OFP_FLOW_MOD` message only instructs the switch to modify the tables that include the MAC address as the destination to specify a new output port. It is important to note that standard broadcast messages do not trigger the transmission of this OpenFlow message, as shown in Fig. 4. This message is specifically intended to expedite the handover process, as previously explained in Sect. 3.2.2c and illustrated in Fig. 8

According to Fig. 16, the STA disconnects from AP1 once the CSA counter reaches 0. During the beacon transmission, the AP stops sending traffic and buffers the downlink packets (iperf3 sequence numbers 15523 to 15525). Before removing the LVAP, the buffered traffic is transmitted; however, since the STA is already disconnected, these packets are lost.

Meanwhile, the Router continues to send traffic to AP1 despite the LVAP being created in AP2, as the data path has not yet been updated (iperf3 sequence numbers 15526 and 15527, which are also lost). As shown in the figure, the number of lost packets depends on the data rate (iperf3 interarrival time of 3.125 ms in this test, as stated above). Packet loss in the wireless medium due to the STA's channel switch is unavoidable unless the AP buffers the downlink traffic, which has not been implemented in this scenario.

Packet loss caused by the outdated wired data path is limited to the delay in path reconfiguration. As observed in the test, this delay is primarily determined by the Ryu processing time, approximately 8 ms (4 ms for each Open vSwitch in the path: AP2 and the Router). However, as illustrated in Fig. 16, this time is shorter than the duration required for the STA to switch channels and announce its presence to the new AP via a QoS Null Data frame (approximately 13 ms after AP2 sends the unicast beacons) then being masked by this time.

Conversely, thanks to the update in the downlink data path, the Router can properly route the traffic in advance (iperf3 sequence number 15528), triggering the creation of an OpenFlow entry in AP2's br1 and allowing its transmission over the

wireless segment once the STA is connected. In this case, as the data carried in the OFP_PACKET_IN is a unicast packet (iperf3 from Router to STA), the corresponding flow entry for unicast traffic is added as specified by OFP_FLOW_MOD, enabling future iperf3 packets to be forwarded directly.

On the other hand, Fig. 17 illustrates the handoff process during uplink data traffic. In this case, a higher handoff delay of 36 ms has been measured, but no packets were lost. Again, these values are specific to this scenario, but as the test was performed under similar conditions to the downlink test, it allows us to identify the differences with the previous scenario and illustrates relevant events.

The STA now sends traffic and buffers packets when switching channels to prevent data loss. As a result, once the STA begins transmitting on the new channel, the buffered packets are sent in a burst through AP2. The size of this burst is limited by the channel switch time, during which the STA buffers data traffic. This packet burst increases the time required for the Ryu controller to process incoming OpenFlow packets, delaying the creation of the flow table entry in each br1 (in AP2 and the Router) by up to 6 ms.

In any case, even a minimal delay (less than 4 ms for the exchange of three OpenFlow messages with any br1) occurs after the STA starts transmitting and can no longer be masked by connection delay as before.

It is worth noting the observed uncontrolled channel switching delay (ranging from approximately 10 to 13 ms in this experiment), which is primarily dictated by the specific NIC hardware of the STA, making its behavior unpredictable. For instance, as observed with the Apple laptop used in this test, the STA transmits non-standard frames (ad-hoc mode) to verify its transmission and reception capabilities after switching channels –an event noted only in the uplink test, where the wireless card actually has pending traffic to send. Conversely, during downlink traffic, the STA follows a standard channel switch procedure, announcing its availability to the AP, which then transmits the downlink packets accordingly.

Comparing the traces in Figs. 16 and 17 (which share the same time scale), the handoff delay is mainly due to three different factors: i) the duration of the Odin signaling and the LVAP migration process (including processing by the Odin Agents in the APs), ii) the wireless channel switching delay and STA activation, and iii) the duration of the Ryu signaling and processing in both the controller and the Open vSwitches (br1 in AP2 and the Router). As stayed before, the factor in ii) cannot be controlled, as it is highly dependent on the specific STA. The factor in i) remains bounded with the optimization procedures for seamless handover, as demonstrated in previous studies.

Regarding the new factor in iii), one of the main contributions of this paper is that deploying the SDWN architecture in Kubernetes has not negatively impacted the total delay, maintaining the operation of the original architecture. Moreover, the innovative incorporation of the tunneled wired data path and its management with the developed Ryu-based SDN controller, introduces a bounded additional delay that is only significant when a handoff occurs during uplink traffic, but is masked in the downlink scenario.

6 Conclusion

In this work, we have presented a solution for deploying a fully virtualized Wi-Fi network, including the deployment of APs, a default Router, and a DHCP server. To achieve this, we extended the SDNWN architecture presented in [4, 5] by incorporating data plane management into the wired segment of the network through the implementation of a controller built on the Ryu framework. This approach enables the deployment to extend across multiple independent physical networks. Additionally, it manages mobility during Wi-Fi handovers by dynamically updating the necessary OpenFlow rules, which in turn adjust the GRE tunnels associated with data transport in the overlay E-LAN network within the wired segment.

Additionally, we automated network deployment and orchestration using a lightweight variant of Kubernetes, the Rancher K3s distribution, which is well suited for the scope of this study, focused on a non-production-oriented proof-of-concept deployment. Control and data plane functionalities were implemented as Docker containers, with a multi-platform build process that packages variants for both ARM64 and AMD64 architectures within the same image, supporting compatibility across different hardware platforms. Particular attention was given to connectivity to enable seamless interaction between the various virtualized functions. While Kubernetes' CNI plugin-based network model is effective for standard application deployments, the deployment of virtualized network functions required a detailed analysis of the aforementioned control and data planes to ensure the proper operation of the overlay E-LAN network responsible for data plane connectivity.

For our proof-of-concept evaluation, we deployed a four-node Kubernetes cluster. Within this cluster, we instantiated two APs, controllers based on Odin and Ryu, a Router, and a DHCP server on both ARM64 and AMD64 architectures. Using this infrastructure, we analyzed two relevant use cases–data path establishment after association and the handover triggered by mobility–by conducting a detailed examination of the traffic flows between network elements and their associated timing. The analysis confirmed that data paths were correctly established after association and consistently updated during handovers, validating the functional correctness of the proposed design in the evaluated proof-of-concept scenarios.

The results demonstrated the feasibility of the proposed approach, although some limitations persist. The proposed system currently focuses on initial deployment and connectivity. Future work will address full lifecycle management–including monitoring, event detection, and automated pod relocation– as well as the implementation of a high-level application layer to translate user requirements into network operations. Nonetheless, this study provides a foundation for exploring advanced end-to-end functionalities such as network slicing, aligning Wi-Fi networks with the requirements of future 5G and 6G ecosystems.

Acknowledgements This research was supported by Grant PID2022–136476OB-I00 funded by MICIU/AEI/10.13039/501100011033/FEDER EU, ERDF/EU, Gobierno de Aragón (reference group T31_23R).

Author Contributions M.C. and J.R.G. proposed the architecture. J.R.M., Á.H.S. and J.F.N. collaborated in refining and revising it. M.C. and J.S. implemented the architecture. All authors collaborated in trouble-

shooting and solving the challenges encountered during the implementation. M.C. and J.R.G. wrote the manuscript. All authors collaborated in reviewing and editing it.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature.

Data Availability No datasets were generated or analysed during the current study.

Declarations

Conflict of interest The authors declare no conflict of interest.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Kreutz, D., Ramos, F.M.V., Ver ssimo, P.E., Rothenberg, C.E., Azodolmolky, S., Uhlig, S.: Software-defined networking: a comprehensive survey. Proc. IEEE **103**(1), 14–76 (2015). https://doi.org/10.1109/JPROC.2014.2371999
2. Suresh, L., Schulz-Zander, J., Merz, R., Feldmann, A., Vazao, T.: Towards programmable enterprise Wlans with Odin. In: Proceedings of the First Workshop on Hot Topics in Software Defined Networks. HotSDN '12, pp. 115–120. Association for Computing Machinery, New York, NY, USA (2012). https://doi.org/10.1145/2342441.2342465
3. Zubow, A., Zehl, S., Wolisz, A.: Bigap seamless handover in high performance enterprise ieee 802.11 networks. In: NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium, pp. 445–453 (2016). https://doi.org/10.1109/NOMS.2016.7502842
4. Saldana, J., Munilla, R., Eryigit, S., Topal, O., Ruiz-Mas, J., Fern ndez-Navajas, J., Sequeira, L.: Unsticking the wi-fi client: Smarter decisions using a software defined wireless solution. IEEE Access **6**, 30917–30931 (2018). https://doi.org/10.1109/ACCESS.2018.2844088
5. Saldana, J., Ruiz-Mas, J., Fern ndez-Navajas, J., Ria o, J.L.S., Javaudin, J.-P., Bonnamy, J.-M., Le Dizes, M.: Attention to wi-fi diversity: Resource management in wlans with heterogeneous aps. IEEE Access **9**, 6961–6980 (2021). https://doi.org/10.1109/ACCESS.2021.3049180
6. Lucas Vieira, J., Mosse, D., Passos, D.: Leaf: Improving handoff flexibility of ieee 802.11 networks with an sdn-based virtual access point framework. IEEE Transactions on Network and Service Management 21(6), 6630–6642 (2024) https://doi.org/10.1109/TNSM.2024.3441390
7. Barakabitze, A.A., Ahmad, A., Mijumbi, R., Hines, A.: 5g network slicing using sdn and nfv: A survey of taxonomy, architectures and future challenges. Comput. Netw. **167**, 106984 (2020). https://doi.org/10.1016/j.comnet.2019.106984
8. Lake, D., Wang, N., Tafazolli, R., Samuel, L.: Softwarization of 5g networks implications to open platforms and standardizations. IEEE Access **9**, 88902–88930 (2021). https://doi.org/10.1109/ACCESS.2021.3071649
9. Babbar, H., Rani, S., AlZubi, A.A., Singh, A., Nasser, N., Ali, A.: Role of network slicing in software defined networking for 5g: use cases and future directions. IEEE Wirel. Commun. **29**(1), 112–118 (2022). https://doi.org/10.1109/MWC.001.2100318
10. Borsatti, D., Grasselli, C., Contoli, C., Micciullo, L., Spinacci, L., Settembre, M., Cerroni, W., Callegati, F.: Mission critical communications support with 5g and network slicing. IEEE Trans. Netw. Serv. Manag. **20**(1), 595–607 (2023). https://doi.org/10.1109/TNSM.2022.3208657

11. Hernández-Solana, Á., Ruiz-Mas, J., Canales, M., Fernández-Navajas, J., Gállego, J.R., Ibáñez-Alloza, S.: Practical challenges of implementing a hybrid ISS-QoSS RAN slicing in SDN WLAN networks. Computer Networks. **284**, 112289 (2026). https://doi.org/10.1016/j.comnet.2026.112289
12. Riggio, R., Marina, M.K., Schulz-Zander, J., Kuklinski, S., Rasheed, T.: Programming abstractions for software-defined wireless networks. IEEE Trans. Netw. Serv. Manag. **12**(2), 146–162 (2015). https://doi.org/10.1109/TNSM.2015.2417772
13. Coronado, E., Riggio, R., Villal n, J., Garrido, A.: Joint mobility management and multicast rate adaptation in software defined enterprise wlans. IEEE Trans. Netw. Serv. Manag. **15**(2), 625–637 (2018). https://doi.org/10.1109/TNSM.2018.2798296
14. Coronado, E., Khan, S.N., Riggio, R.: 5g-empower: a software-defined networking platform for 5g radio access networks. IEEE Trans. Netw. Serv. Manag. **16**(2), 715–728 (2019). https://doi.org/10.1109/TNSM.2019.2908675
15. Isolani, P.H., et al.: Sd-ran interactive management using in-band network telemetry in IEEE 802.11 networks. J. Netw. Syst. Manag. 31(11), 25 (2023) https://doi.org/10.1007/s10922-022-09692-2
16. Nakauchi, K., Shoji, Y.: Wifi network virtualization to control the connectivity of a target service. IEEE Trans. Netw. Serv. Manag. **12**(2), 308–319 (2015). https://doi.org/10.1109/TNSM.2015.2403956
17. Makris, N., Zarafetas, C., Valantasis, A., Korakis, T.: Service orchestration over wireless network slices: testbed setup and integration. IEEE Trans. Netw. Serv. Manag. **18**(1), 482–497 (2021). https://doi.org/10.1109/TNSM.2020.3045224
18. Kassis, M., Aba, M.A., Castel-Taleb, H., Elkael, M., Araldo, A., Jouaber, B.: Integrated deployment prototype for virtual network orchestration solution. In: NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium, pp. 1–3 (2022). https://doi.org/10.1109/NOMS54207.2022.9789812
19. Vittal, S., Sarkar, S., P S, P., A, A.F.: A zero touch emulation framework for network slicing management in a 5g core testbed. In: 2021 17th International Conference on Network and Service Management (CNSM), pp. 521–523 (2021). https://doi.org/10.23919/CNSM52442.2021.9615531
20. Botez, R., Pasca, A.-G., Dobrota, V.: Kubernetes-based network functions orchestration for 5g core networks with open source mano. In: 2022 International Symposium on Electronics and Telecommunications (ISETC), pp. 1–4 (2022). https://doi.org/10.1109/ISETC56213.2022.10010246
21. Tufeanu, L.-M., Martian, A., Vochin, M.-C., Paraschiv, C.-L., Li, F.Y.: Building an open source containerized 5g sa network through docker and Kubernetes. In: 2022 25th International Symposium on Wireless Personal Multimedia Communications (WPMC), pp. 381–386 (2022). https://doi.org/10.1109/WPMC55625.2022.10014753
22. Scotece, D., Noor, A., Foschini, L., Corradi, A.: 5g-kube: Complex telco core infrastructure deployment made low-cost. IEEE Commun. Mag. **61**(7), 26–30 (2023). https://doi.org/10.1109/MCOM.006.2200693
23. Spantideas, S.T., Giannopoulos, A.E., Trakadas, P.: Smart mission critical service management: architecture, deployment options, and experimental results. IEEE Trans. Netw. Serv. Manag. **22**(2), 1108–1128 (2025). https://doi.org/10.1109/TNSM.2024.3498348
24. Phan, L.-A., Pesch, D., Roedig, U., Sreenan, C.J.: Building a 5g core network testbed: Open-source solutions, lessons learned, and research directions. In: 2024 International Conference on Information Networking (ICOIN), pp. 641–646 (2024). https://doi.org/10.1109/ICOIN59985.2024.10572091
25. Bonati, L., Polese, M., D'Oro, S., Prever, P.B., Melodia, T.: 5g-ct: Automated deployment and over-the-air testing of end-to-end open radio access networks. IEEE Commun. Mag. **63**(1), 155–160 (2025). https://doi.org/10.1109/MCOM.001.2300675
26. Andrade, M., Wickboldt, J.A.: A study on 5g network slice isolation based on native cloud and edge computing tools. J. Netw. Syst. Manag. **33**(90), 42 (2025). https://doi.org/10.1007/s10922-025-09958-5
27. Silva, M., Santos, J., Curado, M.: The path towards virtualized wireless communications: a survey and research challenges. J. Netw. Syst. Manag. **32**(12), 45 (2024). https://doi.org/10.1007/s10922-023-09788-3
28. Schulz-Zander, J., Mayer, C., Ciobotaru, B., Lisicki, R., Schmid, S., Feldmann, A.: Unified programmability of virtualized network functions and software-defined wireless networks. IEEE Trans. Netw. Serv. Manag. **14**(4), 1046–1060 (2017). https://doi.org/10.1109/TNSM.2017.2744807
29. Ryu Project Team: Ryu SDN Framework. (2011). Accessed: 2025-02-18. https://ryu-sdn.org/
30. Sequeira, L., Cruz, J.L., Ruiz-Mas, J., Saldana, J., Fernandez-Navajas, J., Almodovar, J.: Building an sdn enterprise wlan based on virtual aps. IEEE Commun. Lett. **21**(2), 374–377 (2017). https://doi.org/10.1109/LCOMM.2016.2623602

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

María Canales was born in Zaragoza, Spain, in 1978. She received the Engineer of Telecommunications MS and Ph.D. degrees from the Universidad de Zaragoza, Spain, in 2001 and 2007, respectively. In 2002, she joined the Departamento de Ingeniería Electrónica y Comunicaciones, Universidad de Zaragoza, where she is currently an Associate Professor and a member of the Aragón Institute of Engineering Research (I3A). Her current research interests include wireless communications and network virtualization.

José Ramón Gállego was born in Zaragoza, Spain, in 1978. He received the Engineer of Telecommunications MS and Ph.D. degrees from the Universidad de Zaragoza, Spain, in 2001 and 2007, respectively. In 2002, he joined the Departamento de Ingeniería Electrónica y Comunicaciones, Universidad de Zaragoza, where he is currently an Associate Professor and a member of the Aragón Institute of Engineering Research. His current research interests include wireless communications and network virtualization.

Julia Santacruz was born in Zaragoza, Spain, in 2001. She received the Bachelor's degree in Telecommunications Engineering from the Universidad de Zaragoza (UZ), Spain, in 2024. In 2023, she joined the Department of Electronics Engineering and Communications of UZ with a research grant. Her research interests include network virtualization.

José Ruiz-Mas was born in Lorca, Murcia, Spain in 1965. He received the M.S. and Ph.D degrees in engineering of telecommunications from the Polytechnic University of Catalonia (UPC), Spain, in 1991 and, from University of Zaragoza (UZ), Spain, in 2001, respectively. He joined the Dept. of Electronics Engineering and Communications of UZ in 1994, where he is currently an Associate Professor and a member of the Aragón Institute of Engineering Research (I3A). His research activity includes communication networks with emphasis on wireless networks and distributed multimedia system.

Ángela Hernández-Solana was born in Zaragoza, Spain, in 1971. She received the M.S. and Ph.D. degrees in engineering of telecommunications from the Polytechnic University of Catalonia (UPC), Spain, in 1997 and 2005, respectively. In 1999, she joined the Dept. of Electronics Engineering and Communications of UZ, where she is currently a Full Professor and a member of the Aragón Institute of Engineering Research (I3A). Her current research interests include wireless communications with an emphasis on radio resource management and wireless sensor networks.

Julián Fernández-Navajas was born in Alfaro, Spain in 1969. He received the M.S. and Ph.D degrees in engineering of telecommunications from the Polytechnic University of Valencia (UPV), Spain, in 1993 and, from University of Zaragoza (UZ), Spain, in 2001, respectively. In 1994 he joined the Dept. of Electronics Eng. and Communication of UZ, where he is currently an Associate Professor and a member of the Aragón Institute of Engineering Research (I3A). His research activity lies in communication networks with emphasis on wireless networks and distributed multimedia system.

Jorge Ortín was born in Zaragoza, Spain, in 1981. He received the Engineer of Telecommunications MS and Ph.D. degrees from the Universidad de Zaragoza in 2005 and 2011 respectively. In 2008, he joined the Aragón Institute of Engineering Research, Universidad de Zaragoza, where he has participated in different projects funded by public administrations and by major industrial and mobile companies. In 2012, he joined the Universidad Carlos III of Madrid, Madrid, Spain, as a Post-Doctoral Research Fellow. From 2013, he works at Centro Universitario de la Defensa Zaragoza. Research interests include wireless communications systems.