Experimenting with open source tools to deploy a multi-service and multi-slice mobile network

Gines Garcia-Aviles[a], Marco Gramaglia[a], Pablo Serrano[a], Francesco Gringoli[b], Sergio Fuente-Pascual[a], Ignacio Labrador Pavon[c]

[a] *University Carlos III of Madrid, Spain*
[b] *CNIT/University of Brescia, Italy*
[c] *ATOS Research, Madrid, Spain*

Abstract

With network slicing, the infrastructure is divided into separate networks, each one customized to provide a specific service. Network slicing is a key technology to efficiently support services with very diverse requirements, such as the ones that should support 5G networks. While the architectural work for 5G is well advanced, and many theoretical solutions that address diverse aspects such as resource assignment or service composition exist, the experimental work lags behind. In this paper, we aim at filling this gap by describing our implementation experiences when deploying a small-scale multi-service prototype. We consider a video streaming service and an augmented reality service, each one provided over a different network slice, and extend existing open-source software solutions for a better provision of them. Our implementation showcases key features of future 5G networks, such as radio slicing with service differentiation, support for local breakout, or multi-slice orchestration with QoE-triggered optimization. With the core of our implementation being open-source, we believe that our results will prove very useful to researchers and practitioners working on this area of research.

Keywords: Network slicing, Service orchestration, 5G Networks

Email addresses: gigarcia@it.uc3m.es (Gines Garcia-Aviles), mgramagl@it.uc3m.es (Marco Gramaglia), pablo@it.uc3m.es (Pablo Serrano), francesco.gringoli@unibs.it (Francesco Gringoli), 100330783@alumnos.uc3m.es (Sergio Fuente-Pascual), ignacio.labrador@atos.net (Ignacio Labrador Pavon)

This paper is an extension of [1]

1. Introduction

While specification and documentation activities regarding the 5^{th} generation mobile systems (5G) are progressing at a rapid speed, practical experiments are progressing at a slower pace. Apart from the obvious reasons due to the relatively high development costs, another (and partly related) issue that precluded the prototype of mobile system is the unavailability of resources: mobile equipment is typically expensive, and the frequency bands are licensed. Because of this, operators were the only players that could afford experimentation with cellular systems, while the academia was restricted to technologies based on the ISM band (see e.g. [2] for a survey on experimentation with 802.11). One technology that helped this ISM-based experimentation is Software Defined Radio (SDR), which enables researchers to implement all layers of a wireless protocol stack.

The use of SDR has enabled the implementation of novel access schemes and communication paradigms (such as e.g. cognitive radio, full-duplex), but prototypes have been limited to the lower layers of the protocol stack (i.e., the MAC layer), the main reason being the lack of interest on developing complete systems over these prototype boards, given the high development costs. These costs are further exacerbated for the case of mobile systems, as these are characterized by relatively complex protocol stacks (in contrast to e.g. TCP/IP). The situation, though, has recently changed with the emergence of software platforms based on SDR that enable instantiating a complete mobile network, with Eurecom's OpenAirInterface (`OAI`) [3] and `srsLTE` [4] among the most popular initiatives. In fact, we have recently carried out an extensive performance comparison of these platforms [5], confirming their adequate performance for prototyping activities and their compatibility with commercial equipment.

The availability of these open projects has levelled up the playground, with now more players (academics, SMEs) being able to develop novel enhancements for cellular systems. This helps accelerating the development of solutions for 5G networking, which will boost the running of field trials under realistic conditions and bring better understanding about the performance of future mobile systems. For instance, thanks to the above mentioned `OAI` and `srsLTE` initiatives, we recently released `POSENS`, the first practical and completely open-source solution for end-to-end network slicing [6]. Moreover, future 5G Networks are expected to be highly heterogeneous, for instance by leveraging on cloud infrastructure as well. So, software that manages the

virtual infrastructure is of paramount importance to efficiently manage multiple services at the same time. The open source community, stemming from the work already available in the cloud computing world, pushed towards this direction, developing highly scalable orchestrators. However, the support brought by orchestration software in the network domain is still under heavy development.

Despite the above, the gap between theory and practice for 5G networking is still large. In particular, while network slicing has been acknowledged as a key technology to efficiently support services with very diverse requirements [7], there are little experimental "hands on" reports on the use of this technology in practice. In this paper, we contribute to filling this gap by reporting on the development and validation of a multi-slice 5G network prototype, each slice serving a different application, and by showcasing several features such as the reallocation of virtual network features or the use of local breakout (LB) to minimize delays [8]. We describe the hardware used, the software installed, and how the different building blocks are connected, providing in this way researchers and practitioners with a "how to" guide while reporting on our experiences and best practices for prototyping. We believe our results provide valuable information to boost the development of further 5G prototype initiatives.

2. Background

In this section, we first provide a short overview of the most relevant open source solutions for the development of mobile network prototypes. Then, we present the two services that we consider towards our implementation, which serve to illustrate the main features of the developed platform. One key feature of these services is that they can be dynamically re-orchestrated, depending on the network conditions and performance experienced (as we validate in our prototype). We finalize this section by describing the contributions of the paper and its structure.

2.1. Existing open source tools for mobile networking

In order to prototype a mobile network with network slicing capabilities, several pieces of software are needed. We can briefly summarize them as follows by drawing a rough comparison to traditional the network domains: the Radio Access Network (RAN), the Core Network (CN) and the Management and Orchestration (MANO) part.

Concerning *the RAN part*, the most relevant solutions are `OAI` [3], and the more recent `srsLTE` [4] from Software Radio Systems. While `OAI` provides an implementation of some selected 5G-NR functionality, the `srsLTE` open-source project provides at the moment a platform for LTE experimentation only, aligned with Rel. 10 and designed for maximum modularity. Even though performance of `OAI` is better (both in terms of throughput and resource footprint), the "smaller" source code of `srsLTE` makes it easier to customize.

Regarding *the CN part*, the most relevant solutions are the ones associated with the above initiatives: `srsLTE` has recently released `srsEPC`, a light-weight CN implementation including the mobility management entity (MME), home subscriber server (HSS) as well as packet and serving gateways (P-GW and S-GW, respectively), while `OAI` provides the same elements for a basic EPC solution.

The *MANO* part has received a lot of attention from both the open-source community and the enterprises, with a variety of tools such as ONAP [9] or OSM [10]. They all rely on a Virtual Infrastructure Manager (VIM), an element that can be provided with solutions such as OpenStack [11]. While these pieces of software offer the basic functionality, more advanced QoE-aware solutions are needed to efficiently support next generation networking.

While each of these solutions provides specific elements to implement a 5G mobile network, only recently [6, 12, 13] researchers started putting pieces together to implement end-to-end network slicing. Among very recent efforts available in the literature that target an objective similar to ours we have the one in [14, 15], which base their implementation on `OAI` instead of `SRSLTE`, as we do. Finally, mention some ongoing initiative such as [16, 17], that aim at providing a living lab for wireless network researchers. In this work, we limit our research to small scale, locally hosted, deployments.

However, hand-on experience on such tools is still largely missing in the literature. In this paper, we aim to fill the gap by providing details on our experience in implementing specific elements envisioned by 5G Networking on a real life testbed.

2.2. Novel services considered

The deployed network provides two services over two network slices, with a focus on aspects like QoS/QoE-aware control, and NFs virtualization and orchestration. The motivation is to feature two different network slices on the cloud infrastructure: one with a reduced latency service and another one with

a mobile broadband service. We show how an ETSI NFV MANO platform can be used to deploy, manage and orchestrate different services on different network slices, this including the dynamic re-orchestration of a particular NS forwarding graph, and the placement of certain VNFs in the appropriate host. In both cases, QoS/QoE aspects trigger the re-orchestration function. The software produced for this paper is partially based on [6] and it is mostly available on GitHub [2] [3].

The two network slices are: (*i*) a Reduced Latency slice (i.e., URLLC), used to read real-time physical measurements triggered by Quick Response (QR) labels, and (*ii*) an enhanced Mobile Broadband (eMBB) slice, which serves contextual captions to streaming media, according to the user profile and surrounding context. Both slices are deployed on the same eNB and share the same spectrum. For the CN part, each tenant (i.e., a service) runs its own instance of the protocol stack (i.e., mobility management, gateways and the upper layers), performing the QoE/QoS policies needed by each scenario. [4] In the following, we describe each slice, listing the most important orchestration-related features that have to be implemented (which will be detailed in Section 5).

2.2.1. Service 1: Video Streaming through the eMBB slice

The first service is hosted by an eMBB slice that has been designed to provide a service consisting of enriching a video streaming signal with context-based add-ons (e.g., subtitles or other graphical elements) depending on the user preferences (i.e., color, language), other conditions (e.g., hearing impairments) and surroundings (e.g., ambient noise). These profiles or environmental conditions can be understood here as QoS/QoE influence factors: a final user, for instance, can generate a trigger to explicitly request the service according to its preferences; or certain QoS metrics could automatically activate the service without an explicit user request.

The additional video features are activated by means of MANO procedures that dynamically add the necessary VNFs to the forwarding graph without interrupting the video streaming. Besides the possible real-life applications of this service, our goal is to demonstrate three different orchestration-

[2] `https://github.com/wnlUc3m`

[3] `https://github.com/bsnet`

[4] Although we deploy a multi-tenancy case where each service belongs to a different tenant, other multi-tenancy scenarios can be supported with our solution.

related functionalities:
(i) On-boarding of the network service itself, i.e., the deployment of the necessary VNFs driven by service descriptors where the operational layout and requirements are defined; *(ii) Dynamic update of the NS Forwarding Graph (FG)* depending on QoS/QoE measurements (QoS/QoE-awareness); *(iii) Placement of VNFs to specific compute nodes*, to simulate the placement of NFs in the edge or core cloud depending on the service requirements.

2.2.2. Service 2: Augmented Reality through an URLLC slice

The Reduced Latency use case consists of an Android application that performs Augmented Reality (AR) using QR codes. In a real environment, those codes may be distributed in an industrial area close to the equipment where measurements of interest are obtained (e.g., pipes flow or pressure, electric measurements, tank levels, etc.). On each QR code decoding, the mobile terminal requests the corresponding information that it displays to the user on top of the captured image. To keep delays between the User Equipment (UE) and the information server low, the latter shall be located close to the user, e.g. in the same element hosting the eNB, or in an edge cloud. To this end we deploy within this application a *local breakout* component that can route selected traffic directly towards an edge cloud. With this feature, information flows are processed locally and incur into smaller delays.

2.3. Paper contributions and structure

Thus, in this paper we describe our experience with Open Source tools in the context of a multi-service, multi-slice softwarized network. The paper is structured as follows: in Section 3, we describe the software components we employed for the (Radio) Access Network. In Section 4 we detail the same for the Core functions, while in Section 5 we discuss the software we used for the Management and Orchestration. We then discuss some evaluation results in Section 6 before concluding the paper in Section 7.

3. Access Network

The mobile network architecture employed in our deployment relies on a well-known open-source software implementations of the LTE Stack: `srsLTE` [4] an open source implementation of a UE and the eNB. In the following Sections we describe how we extended the software architecture to support the

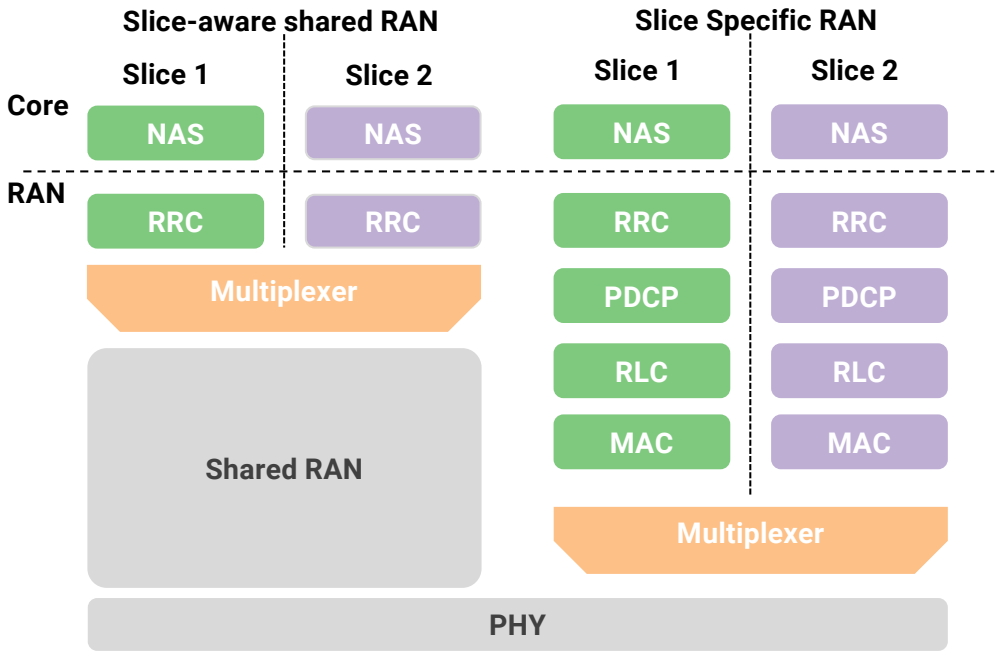

Figure 1: RAN sharing options.

two key features of our solution, namely, (*i*) support for multi-slice in the radio part, providing isolation and resource differentiation across services, and (*ii*) support for local breakout, which enables a more efficient implementation of latency-critical services.

3.1. Slice-aware RAN

While the functions in the Core Network run as VNFs (as we will describe below), because of timing considerations the network functions for the radio part run as physical network functions (PNFs). That is, the cloudification of RAN function is currently a very hot research topic [18]. Still, the porting to real systems of real cloud RAN software is in its early stages. Our implementation of the slice-aware RAN is based on `srsLTE` [4].

Although it is a full software implementation of a 3GPP RAN suite, also `srsLTE` has a limited integration in the state of the art virtualization technologies such as Virtual Machines and Containers. Similar considerations apply for the other major open source RAN platform: `OAI` [3].

The `srsLTE` suite used for this implementation defines different classes for the different tasks that have to be fulfilled in the RAN: (*i*) a common

library that performs the encoding/decoding operations, up to the MAC; (*ii*) the RLC, RRC and the PDCP layers; and (*iii*) specific modules for the UE (the UE module for the data plane of the UE) and the eNB (S1AP, that manages the connectivity for the core control plane, and GW for the GW connectivity).

In order to support a multi-slice network, software modifications to the `srsLTE` mobile network protocol stack are needed. More specifically, the RAN has been modified to provide: (*i*) some means to multiplex and de-multiplex traffic associated to the two different slices, and (*ii*) some means to control the resources assigned to each slice, so the the corresponding requirements are fulfilled, e.g., minimum throughput for a mobile broadband service. Note that the former is a fundamental enabler as the two implemented services share the same RAN, while each tenant can deploy its own core NF for each service. In the following we provide two alternative implementation for the RAN slicing solutions. In Section 3.1.1 we discuss our baseline approach for introducing network slicing in the RAN (namely the Slice-aware shared RAN in Figure 1). This implementation is at the basis of the subsequent evaluation discussion provided in Sections 6. Then, in Section 3.1.2, we discuss on a possible additional implementation (the the Slice-specific RAN in Figure 1). Although they are alternative in the same RAN deployment, they can be selected according to the kind of service. For instance, if two services require a fine-grained control on the resources available to each slice, a slice specific RAN implementation may be a better choice. Instead, a Slice-aware shared RAN can be beneficial when different services do not require differentiation in the access, but just in the core part of the network.

3.1.1. RAN slicing implementation

We implemented the so-called "Slice-aware shared RAN" scheme as it is defined in [7] and represented in Figure 1. To this aim, we modified the higher layers of the network protocol stack (i.e., PDCP and RRC) on both the eNB and UE. Namely, we provided per slice differentiation for the common c-plane and u-plane procedures, as follows: for the c-plane, we had to allow two registrations against two different Non-access Stratum (NAS) instances, so we duplicated (i.e. an instance for each slice) the UE module and the RRC layer, which triggered two NAS connectivity requests. On the other hand, the u-plane is multiplexed and de-multiplexed at the PDCP module, according to the traffic final destination address. That is, by triggering two NAS registration procedures, the UE obtains two valid IP addresses from

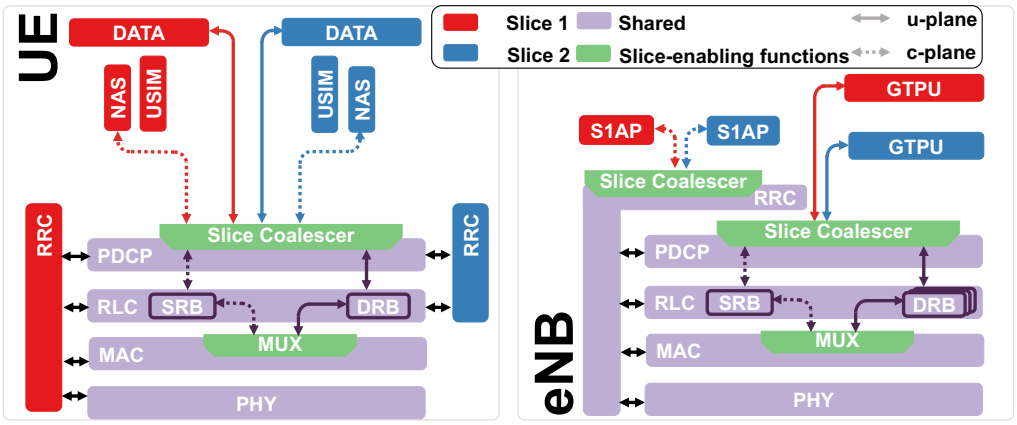

Figure 2: The Slice aware shared RAN implementation details, adapted from [6]

each slice GW and creates two virtual network TUN interfaces (one per each slice). Therefore, the UE can connect to two different slices at the same time.

A sketch of the proposed implementation is depicted in Figure 2. The network slicing enabling modules, marked in green, provide the (de)multiplexing functionality described above, discriminating the traffic (both control and user plane) belonging to each slice between the slice specific modules (among them the NAS, which is part of the core functionality explained in Section 4) and the shared ones. This allows for a differentiation of different flows in the core part, while keeping the radio shared, enabling enhanced orchestration capabilities such as the one described in Section 5. More details about the slice-aware shared RAN implementation can be found in [6].

3.1.2. An alternative implementation for RAN slicing

As shown in Figure 1, the one described above is one of the possible options for RAN slicing. Therefore, we implemented also the "Slice Specific RAN" one, by modifying the `srsLTE` platform.

Differentiating traffic at the MAC layer is a problem that has received a broad attention in the literature in the past [19]. However, the introduction of network slicing into MAC schedulers requires modifications that go beyond the metrics that are usually controlled by state of the art solutions such as the delay budget or the minimum throughput. As already discussed by [12], network slicing aware solutions need also to take into account other factors such as the amount of resources allotted to each slice, which are usually not considered.

In order to differentiate the resources given to each slice, for simplicity we decided to introduce modifications to the Scheduler that is in charge of assigning resources to the different UEs attached (in the following, we assume that different UEs are assigned to different slices). Our motivation is to demonstrate the feasibility of a slice-aware scheduler, by enforcing different resource assignment to tenants according to, e.g., some pre-defined requirements (the means to convey these requirements are out of our scope). For instance, if on a 5G radio access point two tenants are assigned with concurrent slices, but one has more stringent requirements than the other, it shall be granted a larger share of resources to fulfill them.

The scheduler is in charge of the allocation for shared time-frequency resources among users at every time instant. In `srsLTE`, this module is located in the eNB part and assigns uplink and downlink resources according to the scheduling policy. Thus, it determines to which user the shared resources (time and frequencies) for each TTI (in our case, 1 ms) should be allocated for reception of Downlink Shared Channel (DL-SCH) transmissions (We refer the reader to [4] for a thorough description of the `srsLTE` platform)

By default, `srsLTE` implements a Round Robin scheduler: commonly used in LTE networks, UEs are given resources sequentially. When all the UE have been assigned with Resource Blocks (RB) at least once, it starts over. In this way, this scheduling results very simple and does not take into account e.g. the Channel Quality Information (CQI) reported by UEs. While this has some disadvantages in terms of spectral efficiency (as UEs with instantaneous poor channel conditions are scheduled), it is easy to be implemented and provides good fairness in the terms of Resource Blocks (RBs) allocation.

We decided to modify the Round Robin scheduler to implement a weighted Round Robin, where the assignment is performed by weighting the time slot assignment following the priority assigned to each slice. For this purpose, we define a vector of weights $\mathbf{W} = [w_1, w_2, \ldots, w_n]$, where w_i represents the relative weight of slice i among the n slices served by the RAN. For instance, $\mathbf{W} = [1, 1, 1]$ means that the system is serving $n = 3$ slices with equal weight, while $\mathbf{W} = [2, 1]$ represents a system with $n = 2$ slices, with the first one obtaining twice the resources than the second.

If we let $l = \sum_1^n w_i$ and $\hat{\mathbf{W}}$ be an array that holds the cumulative sum of $\mathbf{W}$, we assign RBs in the following way. Being t_i the incremental TTI number we compute t as its modulo base l. Finally the scheduled slice i is $i = \arg\min_i \hat{w}_i - t$. In this way we assure that upon every cycle of l TTIs, we assign resources to slices according to their relative weight $\mathbf{W}$.

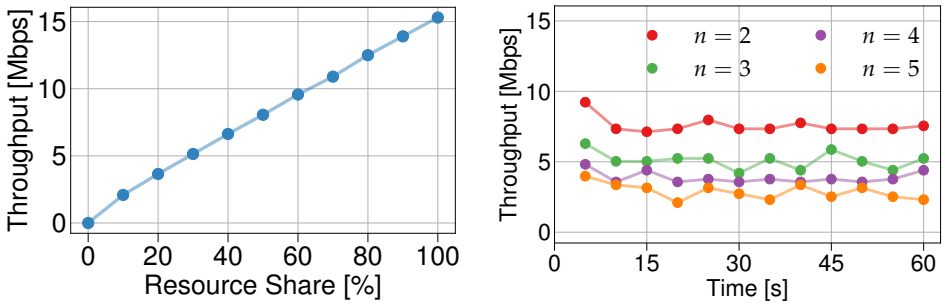

Figure 3: The throughput obtained by our implementation with variable resource shares vs share (left) and vs time (right)

We tested our implementation with the same setup we used for the evaluation described in Section 6, consisting of one UE and one eNB connected through two SMA to SMA cables. The full bandwidth achievable with this configuration is roughly 15 Mbps with good channel conditions.

We emulate the presence of other slices by leaving the resource blocks originally assigned to them blank. We then run an `Iperf` in downlink direction with TCP traffic. Figure 3 shows the correctness of our software implementation in terms of throughput for different sharing options (see Figure 3a) and along time (see Figure 3b).

In the first experiment we create 2 slices (i.e., $n = 2$) and assign an increasing share to the first slice (the one that holds the UE) by modifying the vector $\mathbf{W}$. The results, shown in Figure 3a, demonstrate the effectiveness of our implementation in correctly enforcing the amount of resources assigned to a slice. In the second experiment, we generate a variable number of slices and assign to them the same amount of resources. The results, depicted in Figure 3b, show how our implementation can maintain the throughput around the assigned level for all the duration of our tests (60s).

Although this implementation, available in our repository, is not meant to implement a sophisticated scheduling policy with strong mathematical guarantees, it is indeed a first step towards the open implementation of one of the RAN slicing options depicted in Figure 1.

3.2. Local breakout

Enforcing network slicing at the RAN level (as discussed in Section 3.1) allows to implement different per-slice data traffic management policies as

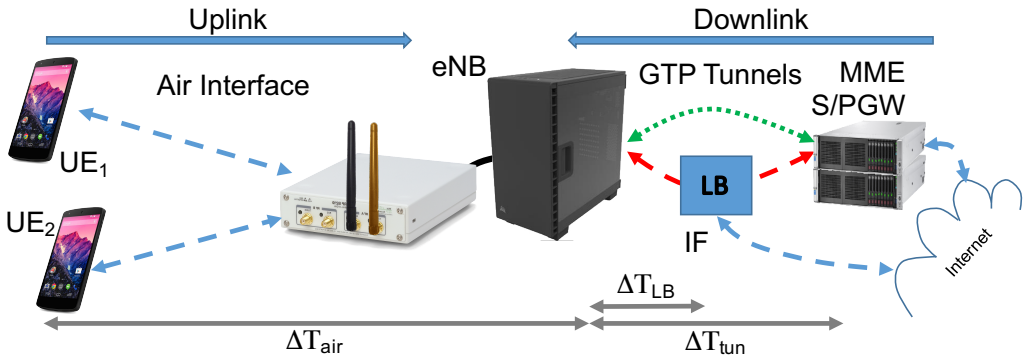

Figure 4: General scenario showing a legacy GTP tunnel (top, green dotted line) and a GTP with LB (bottom red dashed line). The LB introduces additional interface IF for routing traffic locally.

early as the traffic flows leaves the Radio. This is of particular importance when the considered slices have stringent latency requirements and their data flows shall be handled at the edge. One promising solution to this problem is Local Breakout (LB). In the following, we describe the implementation aspects of a LB solution based on the hardware platform depicted in Figure 4.

The local breakout functionality could be deployed in different manners: for instance by directly modifying the network configuration to support different Packet Data Networks (PDN) on the UE. However, deploying it as an independent VNF results a more flexible solution, for at least two reasons: *(i)* it can be maintained and upgraded separately, and *(ii)* it would work (in principle) with any eNB, as it does not require any NF-specific API. This point makes the lifecycle management of the LB software much easier, allowing to deploy it on demand and at runtime. We next present the main characteristics of our LB implementation.

3.2.1. GTP-tunnel and LB

Differently than in a Wireless Local Area Network, traffic generated by mobile users is not routed at the device that terminates the air interface: it is, in fact, delivered inside a GPRS Tunneling Protocol (GTP) tunnel to the remote Gateway controlled by the user's service provider (S/P-GW for LTE, UPF for 5G) where it starts its journey to the destination (top tunnel in green/dotted line in Figure 4). Return traffic is first received by the gateway that tunnels it to the specific eNB at which the destination

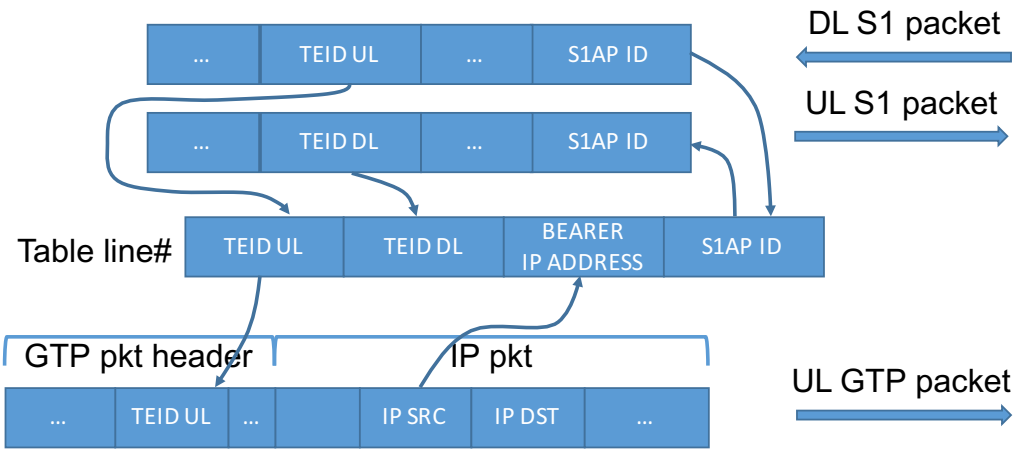

Figure 5: TEID table maintained by the LB threads: it associates TEID numbers to the IP address of each UE. It is used for crafting GTP tunnel return packets.

UE is connected. This method facilitates accounting for roaming users but introduces inefficiencies at the routing level.

To avoid these issues, a LB device can be set up to intercept GTP packets earlier (bottom tunnel in red/dashed line). We report in the following the LB architecture focusing on the mechanisms for transparently intercepting and conveniently routing selected traffic locally.

3.2.2. LB implementation

As shown in Figure 4, the LB node has to *(i)* inspect tunneled packets to extract those matching specific rules and forward them through the new interface IF; and *(ii)* push packets received from IF into the tunnel.

As uplink users' packets are embedded in UDP datagrams, the LB can easily access both the fixed length GTP header and the original IP packet. The GTP header is 8 byte long and contains the *Tunnel endpoint identifier* (TEID), a 32-bit value that identifies the bearer to which the inner IP packet is addressed and that is generated by the eNB/MME when the UE node connects to the network (or when it requests a new service). We implemented the tasks for intercepting and pushing packets into the tunnel and for determining the TEID values as three main threads that refer to a common TEID table for storing/fetching TEID values:

S1 sniffer thread. When a UE requests a new service, the eNB and the MME exchange a couple of S1 packets that carry new TEID values: one in

downlink with the TEID decided by the MME; followed by one in uplink carrying the TEID chosen by the eNB. The knowledge of the latter is fundamental to the LB for pushing downlink packets received from the IF interface into the tunnel. We show in Figure 5 how the S1 thread correlates the two S1 packets by using the common S1AP-ID field to create a new row in the TEID table with the corresponding TEID values. Because of the complexity in dissecting S1 traffic, the S1 thread forks a `tshark` process and uses a pipe for receiving the TEID data from it. It is worth noting that at this stage the IP address of the UE is not yet known: it will be discovered by the Uplink thread as we explain next.

Uplink thread. To inspect all GTP packets going to the S/PGW, the Uplink thread installs a rule in the netfilter framework of the LB kernel that matches UDP packets addressed to the remote gateway. Setting NFQUEUE [5] as target allows the Uplink thread to receive all packets and decide which must be stopped, stripped by their GTP header and injected through interface IF.

When a UE bearer transmits an uplink data packet for the first time, the thread adds the source IP address found in the packet inside the TEID table: to this end, it looks up the corresponding row by searching the TEID UL value extracted from the GTP packet, as shown at the bottom of Figure 5.

Downlink thread. This thread receives packets from the IF interface and pushes them into the GTP tunnel. It first uses the destination IP address of the packet to look up in the TEID table the value of the TEID DL field: it then crafts a new GTP packet by copying the TEID DL value in the header and concatenating the IP packet in the GTP payload. If no TEID DL value is found, the thread simply drops the packet.

4. Core Network

The transition towards 5G and the application of new concepts of network softwarization has pushed the standardization efforts towards a cleaner designed of the core network with respect to the 4G/LTE. More specifically (since Rel. 14, the last 4G one), the main CN modules implement a split between the control plane and the user plane, distinguishing between the Network Function devoted to c-plane functionality and the one that handles

[5] `https://www.netfilter.org/projects/libnetfilter_queue`

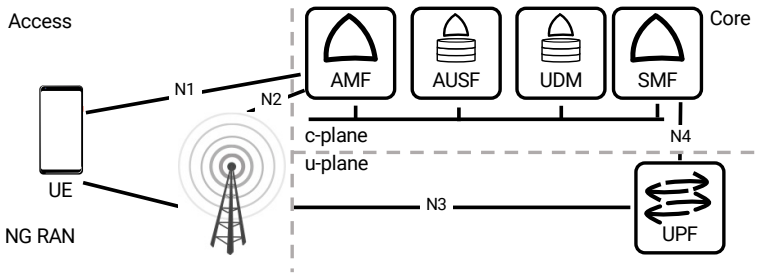

Figure 6: Sketched view of the 5G Core, as in [20]

u-plane. Therefore, in our implementation, we decided to leverage on this new design paradigm, to provide an implementation of some selected functionality of the 5G Core, which follows the c-/u-plane split discussed above. Some of the code has been adapted from existing open source projects, while other components which were not available as open source have been implemented from scratch. Also this part is publicly available in our repository.

We next describe the baseline 5G Core architecture that we have developed that includes: (*i*) the interface towards a multi-slice capable access network (both in the UE and eNB, such as the one described in Section 3.1), to support end-to-end network slicing, (*ii*) a *modularized* implementation of the Core Network, as mandated by recent 3GPP standards, and (*iii*) their interfaces towards the MANO framework.

Although there are a number of initiatives providing Core Network (CN) functionality (e.g., the ones included in `OAI` and `srsLTE`, or NextEpc[6]), in our testbed we implemented most of our solutions from scratch, to take full advantage of the novel architecture designed by 3GPP, including the following features: (*i*) a clean control/user plane split and, (*ii*) a service based architecture (SBA) for the c-plane function. Specifically, we have implemented all the CN modules illustrated in Figure 6, including the c-plane interfaces for the SBA (which are mandated by 3GPP). The implementation of the SBA allows for a consumer / producer communication that enables the modularization of the core network functions. The implementation of the Access Management Function (AMF), AUSF and User Data Management (UDM) functions is done in `Python 3` and inspired by the highly-modular design of `srsLTE`, as detailed next.

[6] `http://nextepc.org/`

4.1. User Plane Function (UPF)

This function provides the encapsulation, decapsulation, and forwarding to the Packet Data Network. Its context consists on the current rules applied to encapsulate/decapsulate packets and to forward them. This function has already been built to enable the standard OpenFlow protocol to deliver the c-plane rules (i.e., the context) to the function, which simplifies the implementation of the context extraction and installation primitives, as they are very similar to the existing OpenFlow primitives. We have implemented the UPF module building on the Open Source, OpenFlow-capable `Lagopus` switch.[7]

4.2. Session Management Function - SMF

This c-plane function controls and configures the UPF instances on the u-plane through the N4 interface. Thus, the context here also consists of the rules to encapsulate/decapsulate/forward packets, in this case for all the UPF functions controlled by the SMF. For the implementation of this module, we leverage available SDN-capable implementations, enriching them with mobile network functionality, and employing a `Ryu` Controller[8] to implement the N4 interface between the UPF and the SMF. One key feature of our implementation is that the context information is stored in a separated object, thus facilitating its extraction and installation.

4.3. Other Core functions

In our testbed we also implemented other Core network functions that implement some selected functionality for our setup. In order to provide the basic authentication functions we developed a tailored Python prototype of the Authentication Server Function (AUSF), which retrieves user information from the Unified Data Management (UDM) modules that holds all the basic user information such as the private keys or the IMEI.

While the two modules above mostly deals with the authentication procedures, we also had to implement a lightweight version of the Access and Mobility Management Function (AMF) which deals with the selection of the radio access from the core perspective. While this functionality is needed for the overall behaviour of the network, in our scenario is trivially linking the `srsLTE` software with the core modules to provide connectivity.

[7] `http://www.lagopus.org`
[8] `https://osrg.github.io/ryu/`

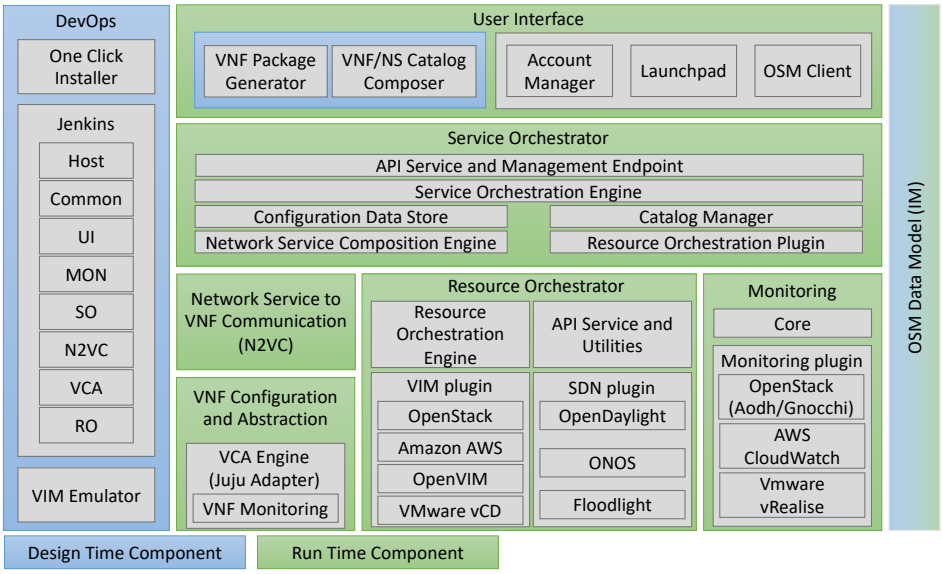

Figure 7: The OSM architecture (Adapted from [10])

5. Management and Orchestration

5.1. The MANO Engine

In order to implement the features envisioned by the testbed architecture described above, we rely on an orchestration architecture based on the Open Source MANO (OSM) orchestrator and the OpenStack VIM. OSM is one of the leading solutions for the implementation of a fully-fledged mobile network orchestrator that includes several components for the automatic LCM. Still, to provide the enhanced functionality needed, substantial changes to the OSM code and architecture are provided.

Figure 7 shows the OSM architecture, composed by several modules that carry out the different functionality needed by an orchestration. In the following, we discuss the additional modules that are required for supporting the specific features needed for our purposes.

Resource Orchestrator. This module provides the hooks towards different VIMs supported by OSM. In the context of the testbed, we use OpenStack as the main VIM, to manage the local Network Function Virtualisation Infrastructure (NFVI) deployment. However, we could leverage also on other APIs: the Amazon AWS EC2 one, to demonstrate the feasibility of a large-

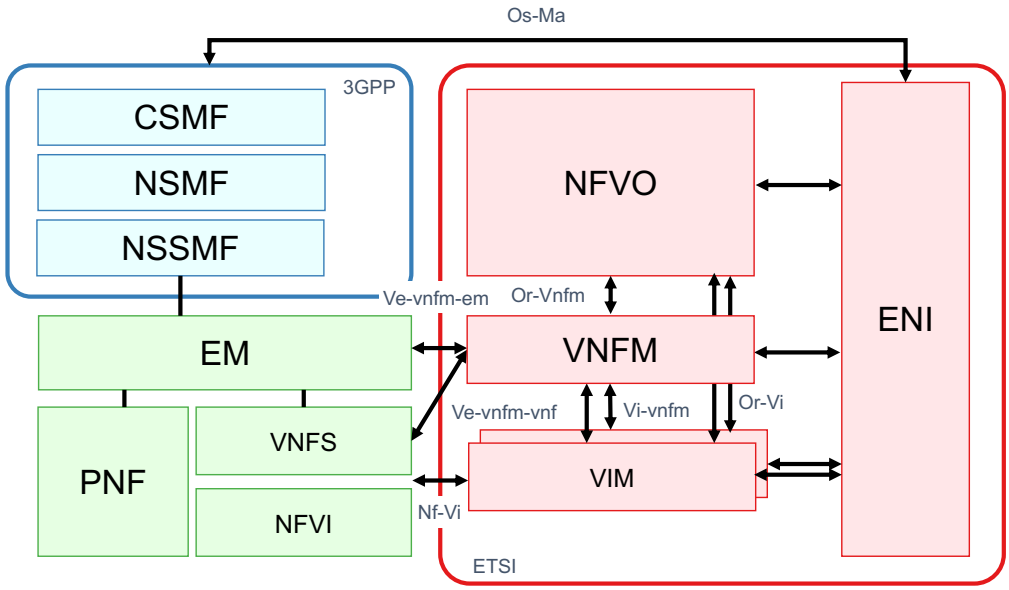

Figure 8: The relations between our implemented architecture and the ETSI - 3GPP domains

scale deployment over public clouds and the Kubernetes one, to possibly include container-based VNFs.

Network Service to VNF communications. We extend this API (that is used to configure the VNFs in a chain) to allow more specific configurations (such as the ones needed by the radio ones, that are PNFs) and support the VNF relocation.

Orchestration module. the specific algorithms that trigger the network function relocation or the local breakout reside on top of the Service Orchestration module, to use the API specifically designed for that purpose.

5.2. ETSI NFV compliancy

The diagram represented in Figure 8 shows the framework of the testbed based on the well-known ETSI NFV [21] and 3GPP MANO architectures [22] that are also building our one. We populated the specific modules of the architecture with our algorithms, as specified next.

Onboarding. This functionality encompasses all the baseline operation available in a 5G MANO system. The automatic onboarding of the two network slices is a seamless operation that allows (*i*) verticals to define the requirements associated with each slice (in this case, high bandwidth for the eMBB and very low latency for the URLLC) and (*ii*) the deployment of the VNF in the cloud to fulfil the set of requirements. This part has specific elements of novelties for the blueprinting. Specifically, we implemented it by using the YAML file descriptors that are used for the definition of VNFs in OSM. More specifically, two kinds of descriptors have to be created: one for the virtual networks (i.e., the virtual links that span one or more VM and connect VNFs among them) called Network Slice Descriptor (NSD, see Listing 1) and one for the virtual appliances that run on top of them, called Virtual Network Function Descriptor (VNFD, see Listing 2).

Listing 1: A snippet of the Network Slice Descriptor.

```
vld:
    # Networks for the VNFs
    - id: management
      name: management
      short-name: management
      type: ELAN
      mgmt-network: "true"
      vim-network-name: public
      vnfd-connection-point-ref:
        # Specify the constituent VNFs
        # member-vnf-index-ref - entry from constituent vnf
        # vnfd-id-ref - VNFD id
        # NGINX
        - vnfd-id-ref: cirros_vnfd_sl1
          member-vnf-index-ref: "1"
          vnfd-connection-point-ref: vnf-mgmt1
          ip-address: 192.168.200.185
          # 245_UPPER_LAYERS
        - vnfd-id-ref: cirros_vnfd_sl1
          member-vnf-index-ref: "1"
          vnfd-connection-point-ref: vnf-mgmt2
          ip-address: 192.168.200.186
          # 245_UPF
        - vnfd-id-ref: cirros_vnfd_sl1
          member-vnf-index-ref: "1"
          vnfd-connection-point-ref: vnf-mgmt3
          ip-address: 192.168.200.187
```

Listing 2: A snippet of the VNF Descriptor.

```
- id: upf
  name: upf
  description: UPF
  count: 1
  # Flavour of the VM to be instantiated for the VDU
  vm-flavor:
    vcpu-count: 1
    memory-mb: 512
    storage-gb: 5
  # Image including the full path
  image: "ubuntu"
  interface:
    # Specify the external interfaces
    # There can be multiple interfaces defined
    - name: eth0-mgmt3
      type: EXTERNAL
      virtual-interface:
        type: VIRTIO
      external-connection-point-ref: vnf-mgmt3
      position: 1
    - name: eth1-in10_0_0_3
      type: EXTERNAL
      virtual-interface:
        type: VIRTIO
      external-connection-point-ref: in10_0_0_3
      position: 2
    - name: eth2-in11_0_0_3
      type: EXTERNAL
      virtual-interface:
        type: VIRTIO
      external-connection-point-ref: in11_0_0_3
      position: 3
```

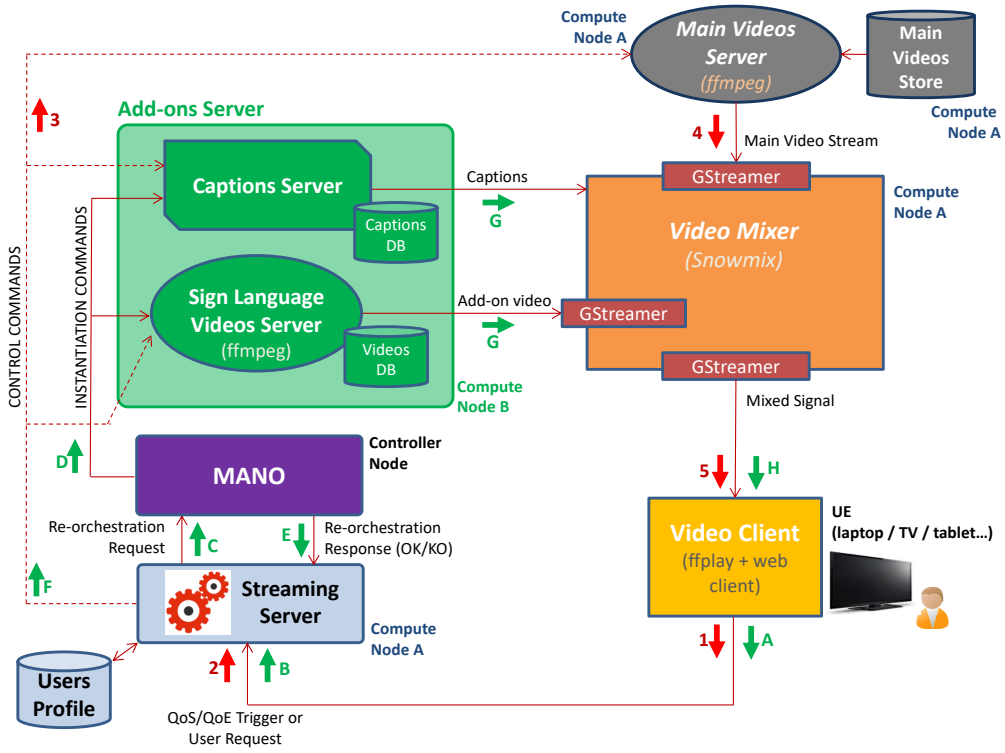

Figure 9: The Orchestration architecture

5.3. Functions beyond 3GPP

The network functions described in Section 4 provide eventually IP connectivity towards the so-called Packet Data Network (PDN) that is attached to the UPF. However, in order to provide novel 5G services (such as the ones described in Section 2.2), additional functions besides the 3GPP ones shall be instantiated and orchestrated by a MANO framework. For instance, an eMBB slice may need a Content Distribution Network to be deployed or an URLLC slice may need to leverage on Mobile Edge Computing capabilities.

5.3.1. Enriched video service

Specifically, the main feature of the eMBB network slice deployed in this work is to provide a service that may add context-based add-ons on a live video streaming signal, depending on the user profile, preferences, and certain environmental conditions. These parameters are defined here as QoS/QoE influence factors, and serve as VNFs re-orchestration triggers.

The add-ons can be subtitles, generated depending on the subscriber preferences (language, colour, size, position, etc.) or other support videos, with the sign language translation (e.g., for people with a hearing impairment). These add-ons could be explicitly requested by the user at any time during the video playback or triggered by environmental conditions (e.g., excessive ambient noise), so a good synchronization between add-ons and the main video signal is a must.

The MANO architecture and the orchestrated network functions to provide this service are depicted in Figure 9. In the Figure, the red arrows labelled from 1 to 5 show the initial interactions, where the user request the playback of a certain video from a client (yellow box on the right). This is just a regular video service providing the requested video to the user, where the Streaming Server block (bottom left) is acting just as a proxy for the Main Videos Server (where the video files are actually stored). The video stream pass through the Video Mixer block (orange box), which although it does not mix the video with any other content, it helps to prevent disruptions when add-ons are inserted.

When add-ons are requested by the user (green arrows, labelled from A to H), the Streaming Server communicates with the MANO block, which instantiates the Add-ons Server (green box). This server is split into two internal blocks: the Sign Language Videos Server (which is used to provide the sign language videos) and the Captions Server (for providing subtitles). Once the appropriate Add-on Server is instantiated, the MANO informs the

Streaming Server block, which sends control commands towards the Add-ons Server (dashed line). The selected add-on (video and/or subtitle) is injected into the Video Mixer block which delivers the mixed video signal to the final user.

The Video Mixer block is implemented using Snowmix, an open-Source and very-flexible command line tool for dynamically mixing live audio and video feeds which supports overlaying video, images, texts and graphic elements as well as mixing audio [23]. All these components have been embedded in a special-purpose VNF, deployed on one of the compute nodes. The Add-ons Server is split in two VNFs, corresponding to the two blocks in the Figure, i.e., the Captions Server and the Sign Language Videos Server. Each VNF contains a database with the necessary caption and video files, and a service running over TomCat [24]. The service exposes a REST API used by the Streaming Server block to execute the necessary control commands once the VNF instances are up and running. The Caption Server communicates with the Video Mixer block using a Snowmix-specific protocol which makes it possible to specify where and how the add-ons are placed within the video. The Sign Language Videos Server uses ffmpeg [25] to stream the add-on videos into the mixer.

The Main Videos Server is also an independent VNF, running over the same compute node as previously mentioned. It contains a database with all the possible videos the final user could access. The Streaming Server is an independent VNF which performs three primary functions:

1. It works as a server for the end user. Specifically, it embeds an HTTP server to what the final user can access (using a general purpose web browser) to request the video playbacks and the available add-ons.

2. It decodes the user's HTTP requests, implementing the logic to trigger the instantiation of the necessary add-on server (through a request to the MANO) and sending the necessary control commands to these instances, and also, to the Main Videos Server.

3. It maintains a good synchronization between the main video stream and the add-ons. Add-ons can be requested at any time by the final user, so they must appear properly synchronized with the main video and with a minor delay.

The MANO block (purple box in Figure 9) represents our implementation of the MANO framework. Although not explicitly represented in Figure 9,

it also communicates with the underlying ETSI NFVI MANO components (i.e., NFVO, VNF Manager and VIM, embedded in the OSM platform) to orchestrate and manage the virtual resources. This process is executed on the Controller Node (represented in Figure 10). Finally, regarding the End-User equipment (yellow box in the Figure), it consist of two different elements, namely, the video player and a web browser with a GUI. The former is based on the ffplay video player [25], while the later is developed for the end-user to control the service.

5.3.2. Latency-triggered re-orchestration

Both network slices are orchestrated with all their VNFs (namely the higher layers of the RAN stack, the eNB) instantiated and running in the central cloud. The MANO system continuously collects data about the network parameters of the virtual network (i.e. latency, throughput, available and used radio resources). This is especially important for the URLLC slice, which may have very stringent requirements on the E2E latency between the UE and the server running in the cloud. For this reason, the delay is constantly monitored to avoid operational glitches caused by a sudden delay increase due to external factors, such as additional congestion in the radio and transport, or internal ones, like a high number of UEs connected to the server. In these cases, the orchestration framework triggers (*i*) a relocation of the NFs and VR application to the edge cloud, to benefit from the reduced latency or (*ii*) the offloading of some selected flows through Local Breakout.

6. Validation

6.1. Testbed setup

The platform runs on commodity hardware. More specifically: an Intel 12 Core i7-3930K PC @ 3.2 GHz with 32 GB RAM and 1 TB of storage and two twin AMD FX 8320 eight-core processor PCs @ 3.5 GHz with 32 GB RAM and 1 TB of local storage.

Besides the CPU, memory and storage characteristics, the main criterion to select this hardware is to have two identical nodes, to enable the VNF live-migration functionality. Figure 10 shows the network topology. The twin PCs work as compute nodes in the architecture, i.e., they run some of the VNFs (as we describe next, not all network functions will run as VNFs). The other PC acts as both network controller and storage node. One general-purpose switch is used to interconnect the nodes through a private isolated

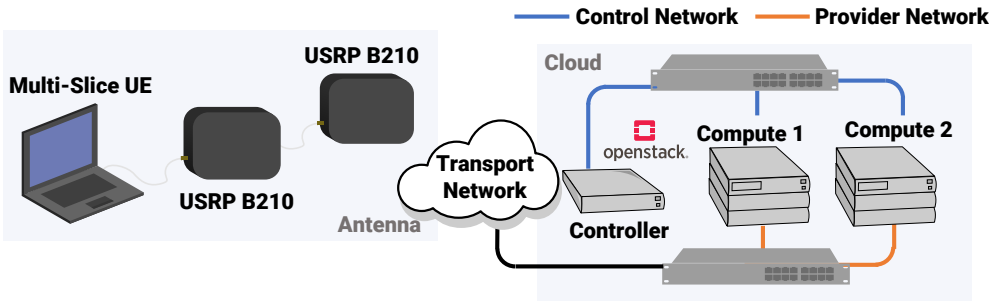

Figure 10: The deployed network setup.

Parameter	Value
Frame Type	FDD
Band	7
Downlink Frequency	2.68 Ghz
Uplink Frequency	2.56 Ghz
Number of RB in Downlink	75
Tx Gain	90 dB
Rx Gain	125 dB

Table 1: Configuration of the Radio Front-end

network, while another switch is used to connect the nodes with the Internet through an OpenVPN Server. All nodes run the Linux CentOS operating system (release 7.2).

The radio part of our deployment is based on two USRP B210 devices by Ettus Research[9], one on the UE side and the other for the eNB. As mentioned in Section 3.1, the network functions for the radio part run as physical network functions (PNFs): the eNB runs on a bare metal server and the UE runs in a laptop. As the USRP B210 features two separate antenna ports (one for receiving and one for transmitting), we cross-connected the two devices with SMA to SMA cables, adding 30 dB attenuators to avoid damages. The left part of Figure 10 represents the connected radio setup. The full configuration parameters for the radio front-end are listed in Table 1.

[9] `https://www.ettus.com/product/details/UB210-KIT`

Phase 1. Onboarding		
Time Components	What	Time
1.01	Network Slice Template (NEST)	7 minutes
1.02	Network Service Descriptor (NSD)	2 minutes
1.03	VNF package (VNFD)	5 minutes
Phase 2. Instantiation, Configuration and Activation		
Time Components	What	Time
2.01	Instantiate Network Slice (NSI)	30 seconds
2.02	Instantiate & Activate Network Service (NS)	45 seconds
2.03	Instantiate & Configure VNFs in service chain (VNF)	10 seconds
2.04	Configure other NFVI elements	10 seconds
2.05	Configure SDN infrastructure	5 seconds
Phase 3. Modification		
Time Components	What	Time
3.01	Modify Network Slice configuration	1.5 minutes
3.02	Modify Network Service configuration	2 minutes
3.03	Detect triggering condition	<1 second
3.04	Modify VNF configuration in service chain	<1 second

Table 2: Service Creation Time KPI

6.2. Service Creation Time

In this section we present the results achieved in our implementation with respect to the service creation time, one of the most important parameters targeted by the major stakeholders in the 5G environment such as 5GPP [26]. Therefore in Table 2 we summarize the timings that we measured in our setup. Intervals are obtained by taking timings of the OSM or OpenStack primitives that are used to perform the different actions, grouped by the main lifecycle management primitives (onboarding, instantiation and modification).

Onboarding. Here we measure the times needed to actually create the service by uploading the YAML descriptors detailed in Section 5. Under the Network Slice Template (NEST) we group the two sub-timings devoted for the NSD and the VNFD. In our setup, which is limited by the available hardware capabilities, we can successfully create the two services in around 7 minutes. This is completely in line with the requirements set by e.g., 5G-PPP ("from 90 hours to 90 minutes") [26].

Instantiation, Configuration and Activation. In this group we measure the timings needed for the activation of the single VNFs that build the service

function chain providing the services. Here (also due to the reduced size of the Virtual Machines involved), the timings are reduced to seconds. Also, some time is needed to configure other NFVI infrastructure such as the transport network. We remark that the timings obtained here are a subset of the Onboarding one. Onboarding requires more time for internal sanity checks performed by OSM before considering the operation completed.

Modification. As discussed in Section 5, we implemented in the Network Orchestration algorithms to perform e.g a VNF relocation (see Section 6). While the timings related to the detection of the triggering conditions and the decision to enforce a new orchestration policy are negligible (sub-seconds), the time needed to prepare the network for such re-orchestration is around a couple of minutes: similarly to onboarding, in fact, this operation requires the preparation of VMs and additional sanity checks.

6.3. VNF re-location

One of the objectives of this multi-service deployment is to showcase advanced orchestration functionality, like a service-aware adaptive allocation of functions to different network nodes using VNF mobility concepts. Regarding this, we envisioned the possibility of relocating the VNFs between different compute nodes, which are acting as edge and central cloud nodes in a real 5G network. For our particular case, we provide this feature by using the so-called "live-migration" functionality offered by OpenStack (which is our default choice for the OSM VIM). Namely, we performed two different experiments, which we detail next.

The first one, based on the so-called "block live-migration", just needs a network connection from the source node to the destination. Basically, the complete virtual machine is transmitted without service interruption. In our specific case, VNF instances can be quite big (tens of GBs once instantiated), so it would be necessary to have a very high bandwidth and dedicated network connection to achieve fast migration times. With a common network connection like the ones we have in our testbed the process takes times in the range of minutes, as discussed before. Moreover, while very high bandwidths may be achievable within the same datacenter, having them available in different geographical locations may be questionable. Furthermore, this approach also reduces the VM processing power during the migration time, so we considered this is not an acceptable approach for a future 5G network.

The second possible approach is called "shared storage live migration". As the name states, it is based on using a shared storage which is accessible from both, source and destination host. In this case, performance has been quite good (in the range of few milliseconds, see entry 3.06 in Table 2), without any service interruption.

Although the performance is quite good using the latter approach, this shared storage node is a drawback in itself. This procedure has well-known inconveniences, among others: (*i*) adding shared storage nodes requires to design the network topology according to this; (*ii*) the shared storage node becomes single point of failure; (*iii*) the network itself becomes a single point of failure; (*iv*) the network security shall be improved as the shared storage should be placed in a separate secured network, and (*v*) even with the good performance results experienced in our demo case, the network latency could impact performance, especially for certain scenarios which require very low latency. However, while this aspect goes beyond the purposes of this paper, we think these problems may be solved in different ways. They range from reducing the size of the VMs to be migrated to the usage of containers [27] or unikernels [28]. Another possibility is the improvement of the shared storage option to tackle the known weaknesses, e.g., adding redundancy to the single points of failure and improving the security.

6.4. Low latency through LB

We tested our LB implementation by running the corresponding VNF in the slice providing the low latency service. We attached to our setup the multi-slice UE (UE1) presented in Section 3.1 and an additional single slice UE (UE2) that connected to the low latency slice only.

In one scenario, we co-located the AR server with the gateway, and compared it with another scenario where the AR server was placed together with the LB VNF. An additional delay of 100 ms was added between the eNB and the gateway, to emulate the distances between edge and central cloud.

The box and whiskers plot in Figure 11 shows the delay for a UE to server and UE to UE communications. The latter is not related to the provided URLLC service, but it is useful to assess the performance of e.g., machine to machine communications. As expected, the latency with the LB VNF activated is much less than the normal case, with a median value of around 30ms and 135ms for the UE to Server case with or without LB respectively. The gap is even higher for the UE to UE case, with median latency figures of 55ms and 255ms respectively.

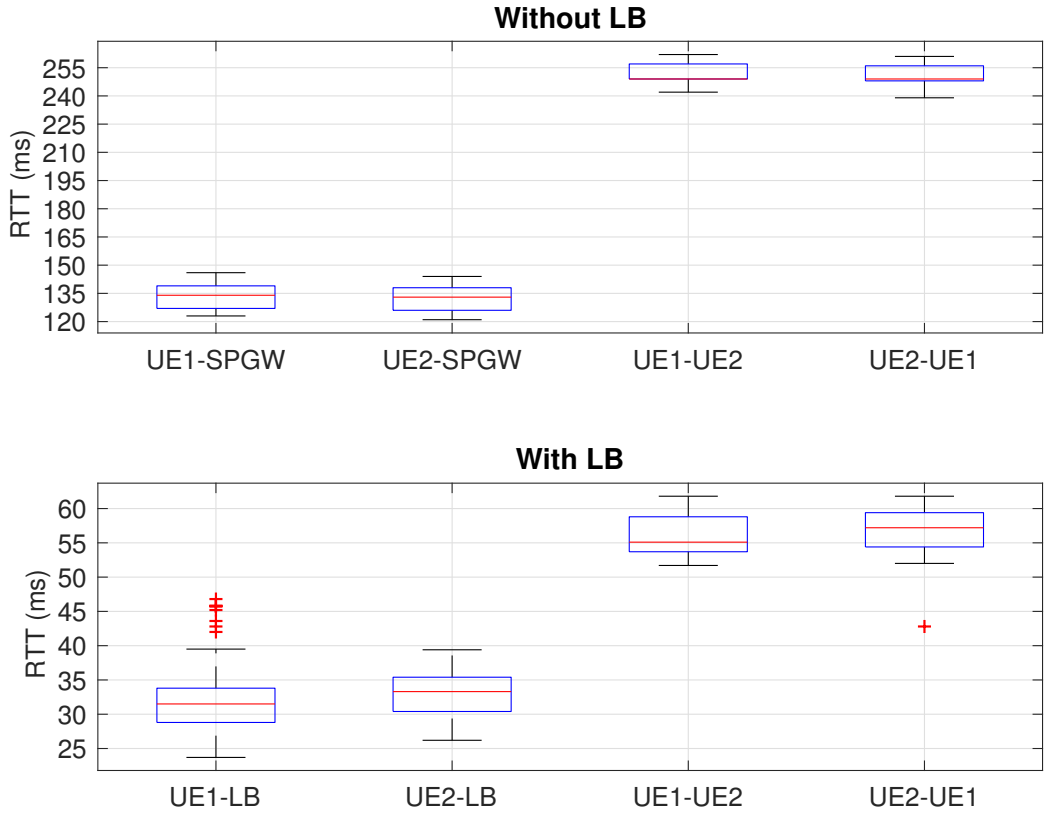

Figure 11: Latency evaluated for the URLLC scenario.

7. Conclusion

The availability of open-source software and cheaper network hardware opened the door for a more flexible experimentation of novel communication paradigm such as *network slicing*, one of the fundamental building block of the 5G ecosystem. Therefore, in this paper we described our hands-on experience gained from the implementation of some distinctive functionality of 5G Networking, (*i*) multi-slice orchestration, (*ii*) RAN slicing and (*iii*) local breakout. All our software is based on open source components, and most of it is also released as open source on public repositories. We believe that our results together with the tools that we release will prove very useful to researchers and practitioners working on this area of research.

Acknowledgement

This work was supported by the H2020 5G-MoNArch (grant agreement no. 761445), 5G EVE (grant agreement No. 815074) and 5G-Tours (grant agreement No. 856950) Projects.

[1] M. Gramaglia, I. L. Pavón, F. Gringoli, G. Garcia-Aviles, P. Serrano, Design and validation of a multi-service 5g network with qoe-aware orchestration, in: Proceedings of the 12th International Workshop on Wireless Network Testbeds, Experimental Evaluation & Characterization, WiNTECH@MobiCom 2018, New Delhi, India, November 2, 2018, 2018, pp. 11–18. doi:10.1145/3267204.3267216.

[2] P. Serrano, P. Salvador, V. Mancuso, Y. Grunenberger, Experimenting with commodity 802.11 hardware: Overview and future directions, IEEE Communications Surveys Tutorials 17 (2) (2015) 671–699. doi:10.1109/COMST.2015.2417493.

[3] N. Nikaein, M. K. Marina, S. Manickam, A. Dawson, R. Knopp, C. Bonnet, Openairinterface: A flexible platform for 5g research, SIGCOMM Comput. Commun. Rev. 44 (5) (2014) 33–38. doi:10.1145/2677046.2677053.
URL http://doi.acm.org/10.1145/2677046.2677053

[4] I. Gomez-Miguelez, A. Garcia-Saavedra, P. D. Sutton, P. Serrano, C. Cano, D. J. Leith, srslte: An open-source platform for lte evolution and experimentation, in: Proceedings of the Tenth ACM International Workshop on Wireless Network Testbeds, Experimental Evaluation, and Characterization, WiNTECH '16, ACM, New York, NY, USA, 2016, pp. 25–32. doi:10.1145/2980159.2980163.
URL http://doi.acm.org/10.1145/2980159.2980163

[5] F. Gringoli, P. Patras, C. Donato, P. Serrano, Y. Grunenberger, Performance assessment of open software platforms for 5g prototyping, IEEE Wireless Communications 25 (5) (2018) 10–15. doi:10.1109/MWC.2018.1800049.

[6] G. Garcia-Aviles, M. Gramaglia, P. Serrano, A. Banchs, Posens: A practical open source solution for end-to-end network slicing, IEEE Wireless Communications 25 (5) (2018) 30–37. doi:10.1109/MWC.2018.1800050.

[7] P. Rost, C. Mannweiler, D. S. Michalopoulos, C. Sartori, V. Sciancalepore, N. Sastry, O. Holland, S. Tayade, B. Han, D. Bega, D. Aziz, H. Bakker, Network slicing to enable scalability and flexibility in 5g mobile networks, IEEE Communications Magazine 55 (5) (2017) 72–79. doi:10.1109/MCOM.2017.1600920.

[8] S. Lee, J. Kim, Local breakout of mobile access network traffic by mobile edge computing, in: 2016 International Conference on Information and Communication Technology Convergence (ICTC), 2016, pp. 741–743. doi:10.1109/ICTC.2016.7763283.

[9] ONAP project, `https://www.onap.org/`.

[10] OSM project, `https://osm.etsi.org/`.

[11] OpenStack project, `https://www.openstack.org/`.

[12] X. Foukas, M. K. Marina, K. Kontovasilis, Orion: Ran slicing for a flexible and cost-effective multi-service mobile network architecture, in: Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking, MobiCom '17, ACM, New York, NY, USA, 2017, pp. 127–140. doi:10.1145/3117811.3117831.
URL `http://doi.acm.org/10.1145/3117811.3117831`

[13] Mosaic5G ecosystem, `http://mosaic-5g.io/`.

[14] X. Foukas, F. Sardis, F. Foster, M. K. Marina, M. A. Lema, M. Dohler, Experience building a prototype 5g testbed, in: Proceedings of the Workshop on Experimentation and Measurements in 5G, EM-5G'18, ACM, New York, NY, USA, 2018, pp. 13–18. doi:10.1145/3286680.3286683.
URL `http://doi.acm.org/10.1145/3286680.3286683`

[15] C. Huang, C. Ho, N. Nikaein, R. Cheng, Design and prototype of a virtualized 5g infrastructure supporting network slicing, in: 2018 IEEE 23rd International Conference on Digital Signal Processing (DSP), 2018, pp. 1–5. doi:10.1109/ICDSP.2018.8631816.

[16] The POWDER testbed, `https://powderwireless.net/`.

[17] The COSMOS lab, `https://www.cosmos-lab.org/`.

[18] A. Garcia-Saavedra, X. Costa-Perez, D. J. Leith, G. Iosifidis, Fluidran: Optimized vran/mec orchestration, in: IEEE INFOCOM 2018 - IEEE Conference on Computer Communications, 2018, pp. 2366–2374. doi:10.1109/INFOCOM.2018.8486243.

[19] Y. Zaki, T. Weerawardane, C. Gorg, A. Timm-Giel, Multi-qos-aware fair scheduling for lte, in: 2011 IEEE 73rd Vehicular Technology Conference (VTC Spring), 2011, pp. 1–5. doi:10.1109/VETECS.2011.5956352.

[20] 3GPP, System architecture for the 5G System (5GS), Technical Specification (TS) 23.501, 3rd Generation Partnership Project (3GPP), version 15.5.0 (03 2019).
URL https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=3144

[21] ETSI, Network Functions Virtualisation (NFV);Use Cases, Tech. Rep. GS NFV 001, v1.1.1 (10 2013).

[22] 3GPP, Management and orchestration; Concepts, use cases and requirements, Technical Specification (TS) 28.530, 3rd Generation Partnership Project (3GPP), version 15.1.0 (12 2018).
URL https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=3273

[23] Snowmix - the swiss army knife of open source live video mixing., https://snowmix.sourceforge.io/.

[24] Apache Tomcat project, https://tomcat.apache.org/.

[25] FFMPG project, https://ffmpeg.org/.

[26] 5G-PPP vision and mission, https://5g-ppp.eu/about-us/.

[27] K.-T. Seo, H.-S. Hwang, I.-Y. Moon, O.-Y. Kwon, B.-J. Kim, Performance comparison analysis of linux container and virtual machine for building cloud, Science & Engineering Research Support soCiety, 2014. doi:10.14257/astl.2014.66.25.
URL https://doi.org/10.14257/astl.2014.66.25

[28] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, J. Crowcroft, Unikernels: Library operating

systems for the cloud, in: Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, ACM, New York, NY, USA, 2013, pp. 461–472. doi:10.1145/2451116.2451167.
URL http://doi.acm.org/10.1145/2451116.2451167