Deployment and configuration of MEC apps with Simu5G

Alessandro Noferi, Giovanni Nardini, Giovanni Stea, and Antonio Virdis

Department of Information Engineering, University of Pisa, Pisa, Italy
a.noferi1@studenti.unipi.it, {name.surname}@unipi.it

Abstract

Multi-access Edge Computing (MEC) is expected to act as the enabler for the integration of 5G (and future 6G) communication technologies with cloud-computing-based capabilities at the edge of the network. This will enable low-latency and context-aware applications for users of such mobile networks. In this paper we describe the implementation of a MEC model for the Simu5G simulator and illustrate how to configure the environment to evaluate MEC applications in both simulation and real-time emulation modes.

1 Introduction

Next-generation mobile networks will need to provide users innovative services with stringent Quality of Service (QoS) requirements, such as reduced latency and high bandwidth. To this aim, providing cloud-computing-like capabilities at the edge of the network is crucial. Multi-access Edge Computing (MEC) is an international standard developed by the European Telecommunications Standards Institute (ETSI) that provides computing infrastructures at the edge of the network, by placing nodes, called *MEC hosts*, as close as possible to end-users. Specific Application Programming Interfaces (APIs) towards the underlying network allows the MEC system to retrieve information about the status of the network and its users in real time. Such information can be used to offer new context-aware services to the end-users.

In order to assess the performance of new MEC-based services, it is key for researchers and developers to have tools for rapid prototyping of applications in a controllable environment that includes realistic models of both the mobile network and the MEC system.

In this paper we present an ETSI-compliant implementation of the MEC system, which is included in Simu5G, an OMNeT++-based simulator for the data plane of 5G networks [8]. We discuss our modelling choices and describe the necessary configurations for testing MEC applications (MEC apps, hereafter) using the proposed framework. In particular, we illustrate an exemplary configuration for simulating a MEC-enabled 5G deployment, as well as the guidelines to run an emulation testbed exploiting OMNeT++/INET real-time and emulation capabilities. The latter features allow one to run MEC app prototypes, coded using any programming language, on top of a realistic 5G environment.

The rest of the paper is organized as follows: Section 2 describes the main components of the ETSI MEC architecture, whereas Section 3 provides an overview of Simu5G. Section 4 discusses our MEC implementation within Simu5G. Section 5 illustrates the guidelines to test MEC applications within Simu5G, in both simulated and emulated mode. Section 6 concludes the paper.

2 ETSI MEC Architecture

According to ETSI [6], a MEC system is organized in two layers, namely the MEC System Level and the MEC Host Level, as shown in Figure 1.

The MEC system level is responsible for maintaining an overall view of the MEC hosts in the MEC system and managing the lifecycle of the MEC apps, i.e instantiation, relocation and

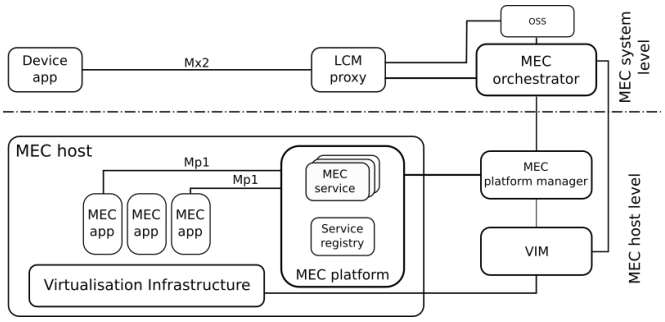

Figure 1: MEC architecture

termination. The core of the MEC system level is the MEC orchestrator, which receives requests (after a granting from the Operations Support Systems (OSS)) to instantiate or terminate applications from a Device application running on users' devices via the Lifecycle Management (LCM) proxy. Then, the MEC orchestrator selects the MEC host where the application should be placed based on the required constraints, such as latency, available resources and required MEC services.

In the MEC host level, the MEC host contains the virtualization infrastructure where MEC apps run as virtual machines. MEC services are deployed in the MEC platform and are consumed by the MEC app through standard MEC APIs, built upon RESTful APIs [1]. Such MEC services can be discovered by querying the Service registry, again via a RESTful API. The MEC platform manager and the Virtualization infrastructure manager entities manage the status of the MEC host and are queried by the MEC orchestrator to retrieve information (e.g. available resources, MEC services, etc.) during the MEC app lifecycle operations.

Communications among MEC system entities occur via standardized reference points, like Mp1 between MEC app and the MEC platform, and Mx2 between Device app and LCM proxy.

Within a 5G network, MEC hosts can be deployed at different location in a flexible way, e.g. at the base station — or an aggregation of them— or in the core network [9].

3 Overview of Simu5G

This section provides a brief description of the main components of Simu5G, with special focus on the modules involved in the MEC architecture.

Simu5G [8] is the evolution of the popular SimuLTE [10] 4G network simulator that incorporates 5G New Radio access. It models the user plane of both the Core Network (CN) and the Radio Access Network (RAN). As depicted in Figure 2, within Simu5G a 5G network is composed of User Equipments (UEs), gNodeBs (gNBs) and User Plane Functions (UPFs) modules. The latter implements the GPRS tunneling protocol (GTP) enabling the routing of packets from the RAN to Internet through the CN.

UEs and gNBs communicate via the layer-2 New Radio (NR) protocol stack, implemented along with the physical layer in the *NrNic* module. The internal design of the NIC for both the gNB and UE is depicted in Figure 3 and it is composed of one submodule for each layer of the NR protocol stack. On the UE side, the NIC also contains submodules for the LTE protocol stack, so as to allow both mixed 4G/5G and E-UTRA/NR Dual Connectivity scenarios.

The topmost part of the TCP/IP stack, i.e. from the application layer to the IP included, is provided by the INET library, and so are the UE mobility models. One can also use other mobility libraries available for OMNeT++. gNBs communicate with each other via the X2 interface, e.g. for handover and interference-coordination mechanisms.

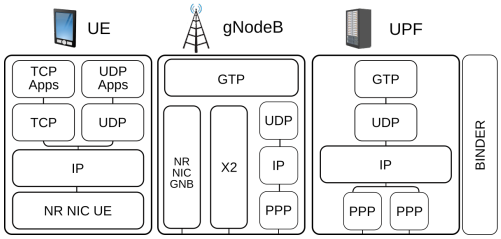

Figure 2: Simu5G main modules

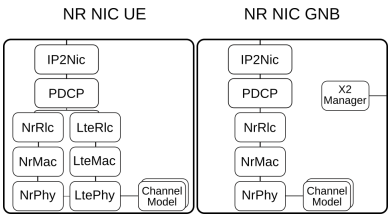

Figure 3: NIC modules

Control-plane functions and protocols are generally not modeled. Instead, a special module, called *Binder* takes care of maintaining network-wide data structures (e.g., which UE is associated with which gNB, etc.). Network nodes (e.g., UEs and gNBs) can query the Binder via direct method calls, thus mimicking control-plane functions without the need to setup complex protocol state machines.

Simu5G can also be run as a real-time 5G network emulator by leveraging OMNeT++'s real-time event scheduler and INET's external interfaces. This enables an external application to exchange packets with a module in an emulated 5G network, or to use the 5G network as a transport network between two external applications.

4 Modelling MEC within Simu5G

The proposed MEC model aims to provide MEC developers a tool for rapid prototyping of MEC apps, which can exploit MEC services made available by Simu5G. Application endpoints (e.g. Mp1 and Mx2 interfaces) and MEC service communication have been implemented according to the ETSI specifications [3, 7, 1]. Since Simu5G can run as a real-time network emulator, one can also interface `real` MEC apps with it. This way, one can test MEC apps in a computation/communication framework that includes 5G transport, a MEC infrastructure, and the MEC services provided by the former to the latter.

In this section, we describe how MEC entities have been modelled.

4.1 MEC system-level modelling

Figure 4 shows our models for the LCM proxy and the MEC orchestrator. The LCM proxy is a compound module including the TCP/IP stack and one application module that provides the RESTful API consumed by the Device app (either internal or external to the simulator, through an INET external interface) to trigger instantiation and termination of MEC apps via the Mx2 reference point. The MEC orchestrator is a simple module connected to the LCM proxy via gates. In scenarios with multiple MEC systems, the MEC orchestrator manages a subset of the MEC hosts in the simulation, hence it must specify such subset using the `mecHostsList` parameter, which is a space-separated string specifying the names of the MEC hosts under its control. The LCM proxy and the MEC orchestrator communicate via OMNeT++ messages, whereas interactions with MEC host-level entities are realized via direct method calls (see Figure 4). Such interfaces are not fully compliant with the standard. On one hand, the reason is that they have not been completely standardized by ETSI at the time of writing. On the other hand, this allows us to model all the functionalities required by the application endpoints without overly complicating the system.

Upon receiving a MEC app instantiation request from the LCM proxy (which in turn receives the request by a Device app), the MEC orchestrator selects the most suitable MEC host among those associated with its MEC system, according to the MEC app requirements. The

latter are specified in the so-called *AppDescriptor* JSON file and can include a list of required MEC services and/or computing resources (RAM, CPU and disk), which are checked against the available resources of the MEC hosts. The location in the file system of the AppDescriptor file can be specified by either the Device app triggering the MEC app instantiation or the MEC orchestrator directly. As per ETSI specifications in [2], the *AppDescriptor* file for our implementation must specify at least the following fields: *appDid*, *appName*, *appProvider*, *appServiceRequired*, as shown in Figure 7.

Once the MEC host has been identified, the MEC orchestrator triggers the MEC app instantiation, which is accomplished by dynamically creating the OMNeT++ module running the application logic (see 4.2) and marking the required MEC host resources as allocated. Similarly, upon a MEC app termination request, the MEC orchestrator informs the MEC platform manager to delete the module and to release the resources previously allocated. We model the processing delay at the MEC orchestrator by scheduling a timer with configurable duration: when the latter expires, the MEC orchestrator replies to the LCM proxy by acknowledging the instantiation/termination of the MEC app.

The Device app is usually embedded in the UE client. However, its functions are quite standard, i.e., managing instantiation/termination of MEC apps by interfacing with the LCM proxy. For this reason, our MEC architecture includes a basic Device app able to query the LCM proxy via the RESTful API and communicating with the UE app over an UDP socket by means of a simple interface including messages for creation, termination and related acknowledgments regarding a MEC app. This way, a user only needs to code the data-plane colloquium between the UE app and the MEC app, as well as their logic.

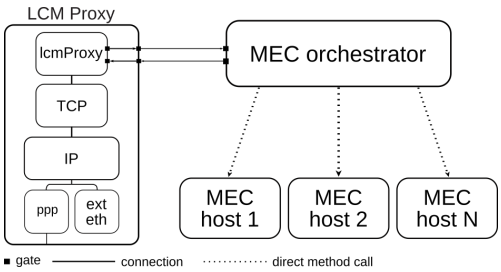

Figure 4: MEC system-level modeling

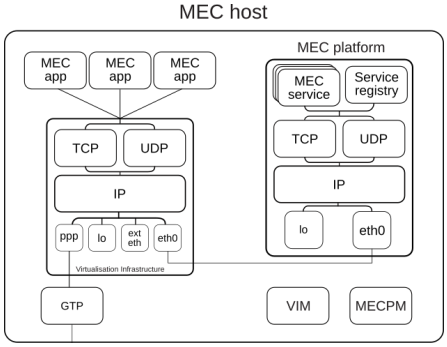

Figure 5: MEC host-level modeling

4.2 MEC host-level modelling

The MEC host module models the MEC host-level of the MEC architecture, as shown in Figure 5. It is configurable (via NED/INI files) with a maximum amount of resources, i.e. RAM, storage and CPU, that can be allocated to MEC apps. Its main components are the Virtualisation Infrastructure Manager (VIM) and the MEC platform modules. As previously mentioned, communications with the MEC platform can also involve real MEC apps when Simu5G runs as an emulator. For this reason, all the entities, i.e. MEC service and Service Registry, implement a RESTful HTTP server as a TCP application running on top of the TCP/IP protocol stack of the MEC platform. In particular, the framework provides a module called *MecServiceBase* and it implements all the non-functional requirements needed for running an HTTP server (e.g. queueing, TCP connection management, gNB connections etc.). This

allows the user to rapidly deploy an ETSI-compliant MEC service by only implementing the HTTP methods (e.g. GET, POST) according to the service behavior.

Two ETSI MEC services are currently implemented: *Radio Network Information service (RNIS)* [4], that allows users to gather up-to-date information regarding radio network condition and the UEs connected to the base station associated to the MEC host, and the *Location service* [5], that provides accurate information about UEs and/or base station locations, enabling active device location tracking and location-based service recommendations.

The *ServiceRegistry* module implements the REST resources – and HTTP methods – needed to allow MEC apps to discover MEC services via the Mp1 reference point [3].

MEC apps are deployed as applications over TCP and UDP transport-layer protocols of the Virtualization Infrastructure compound module. They implement the *IMECApp* interface and are dynamically created by the VIM upon instantiation requests triggered by the MEC orchestrator through the MEC platform manager. The VIM manages MEC host resources and keeps track of the MEC apps currently running on the virtualization infrastructure through a data structure called *mecApps*. For each MEC app, the latter contains a *mecAppEntry* structure with the pointer and the associated gate indexes of the module, used for the deallocation phase, the endpoint of the deployed MEC app returned to the UE in order to communicate with it, and the allocated resources. Such resources are used by the VIM to compute the delay used to model processing time. More in detail, the CPU constraint is expressed in terms of instructions per second, so that we can model the processing time of a MEC app, i.e. the time required by the MEC app to execute a set of instructions. This is done by the VIM according to two different paradigms: *Segregation*, where the MEC app obtains exactly the amount of computing resources it has configured in the *AppDescriptor* file, even when no other MEC apps are running concurrently and *Fair sharing*, in which MEC apps share all the available computing resources proportionally to their requested rate, possibly obtaining more capacity than the amount stipulated in the *AppDescriptor* file.

Finally, a MEC host also has a GTP module that allows it to be placed anywhere in the CN of the 5G network. This is useful, for example, to test different MEC hosts deployments.

5 Deployment scenarios

As already mentioned, Simu5G can run in both simulated and emulated mode. So, the user can develop and test MEC-related applications in either mode, according to her needs. For instance, it could be the case that only a real MEC app should be tested, leaving the UE application as a stub method inside the simulator, possibly because its behavior is not needed or it is not yet available. This section illustrates the configurations required to run some MEC scenarios in both Simu5G execution modes.

5.1 Running a fully simulated MEC system

The network used for the following simulation is depicted in Figure 6. The MEC system contains a LCM proxy, a MEC orchestrator and two MEC hosts, namely *mecHost1* and *mecHost2*, that are associated to a gNB. *Car* is a vector of UEs, each of them running an application called *UEWarningAlertApp*. The latter instantiates a MEC app named *MECWarningAlertApp*, which is supposed to be onboarded in the MEC system during network initialization and to send alert messages to cars when they enter a danger zone. Cars also run the Device app responsible to request the MEC app instantiation to the LCM proxy on behalf the UEs. Besides the module implementing the logic of the *MECWarningAlertApp*, the *AppDescriptor* is also required. In

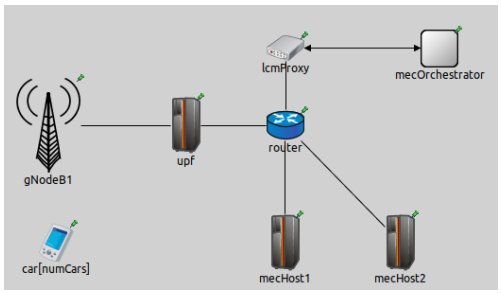

Figure 6: Simu5G network

```
{
  "appDId" : "WAMECAPP",
  "appName" : "MECWarningAlertApp",
  "appProvider" : "lte.apps.mec.warningAlert.MECWarningAlertApp",
  "virtualComputeDescriptor" :{
      "virtualDisk" : 10,
      "virtualCpu" : 1500,
      "virtualMemory" :10
  },
  "appServiceRequired" : [{
      "ServiceDependency" :{
            "serName" : "LocationService",
            "version" : "v2",
            "serCategory" : "Location"
      }
  }]
}
```

Figure 7: AppDescriptor for WarningAlertApp

accordance with 4.1, the *AppDescriptor* file that describes the *MECWarningAlertApp* is shown in Figure 7.

The MEC app requires 10 MB of RAM, 10 MB of storage, 1500 instructions per second of CPU and it consumes the Location Service. Next, it is necessary to configure both the car[*] modules and the MEC entities. For the former, the snippet of the INI file in Figure 8 shows the configuration relevant to the MEC system. Both the MEC hosts have the same computational capacity, but only mecHost2 has a Location Service running on its MEC platform, hence it will be the one chosen by the MEC orchestrator to deploy the MEC app.

```
#-----------UEWarningAlertApp--------------
*.car[*].numApps = 2

# app[0] is the DeviceApp
*.car[*].app[0].typename = "DeviceApp"
*.car[*].app[0].localPort = 4500
*.car[*].app[0].lcmProxyAddress = "lcmProxy"
*.car[*].app[0].lcmProxyPort = 1000
# app[1] is the UE app
*.car[*].app[1].typename = "UEWarningAlertApp"
# Device App address
*.car[*].app[1].destAddress = "car["+string(ancestorIndex(1))+"]"
# Device App port
*.car[*].app[1].destPort = 4500
```

Figure 8: car configuration

```
#####################_MEC Hosts Side_####################
# available resources
*.mecHost*.maxMEApps = 100   # max ME Apps to instantiate
*.mecHost*.maxRam = 32GB          # max Ram
*.mecHost*.maxDisk = 100TB   # max Disk Space
*.mecHost*.maxCpuSpeed = 400000    # max CPU
#----------------------------------------
# gNBs associated to the MEC Hosts
*.mecHost*.eNBList = "gNodeB1"

#-------REST Services:--------------
# MEC host 2 services configurations
*.mecHost2.mecPlatform.numMecServices = 1
*.mecHost2.mecPlatform.mecService[0].typename = "LocationService"
*.mecHost2.mecPlatform.mecService[0].localAddress = "mecHost2.mecPlatform"
*.mecHost2.mecPlatform.mecService[0].localPort = 10020

*.mecHost2.mecPlatform.serviceRegistry.localAddress = "mecHost2.mecPlatform"
*.mecHost2.mecPlatform.serviceRegistry.localPort = 10021

# MEC hosts associated to the MEC system
**.mecOrchestrator.mecHostList = "mecHost1, mecHost2"
# the MECPM needs to know its MEC orchestrator
**.mecHost*.mecPlatformManager.mecOrchestrator = "mecOrchestrator"
# List of MEC app descriptors to be onboarded during initialization
**.mecOrchestrator.mecApplicationPackageList = "WarningAlertApp"
```

Figure 9: MEC system configuration

If the user wanted to request the onboarding of the *appDescriptor* file from the Device app, instead of it being loaded at initialization time, it has to insert the path to the file in the *appPackageSource* parameter of the Device app and remove the *appDescriptor* file name from the list of the MEC orchestrator *mecApplicationPackageList* parameter.

5.2 Running a MEC system in emulation mode

We now describe the configuration required to run a MEC system in emulation mode within Simu5G. More in detail, the following operations will be highlighted: routing rules, in both the simulator and the host running the real applications, and the configuration to instruct the MEC orchestrator to manage the instantiation of a real MEC app. In our test, both the UE app and the MEC app are real applications running on the same host where Simu5G runs. The real applications replicate the behavior described in section 5.1 in a network scenario including one car and one MEC host, as shown in Figure 10. The Device app runs inside the UE and communicates with the real UE app through the serialization and deserialization of the

messages of the interface mentioned in Section 4.1. Although we focus on a single-host testbed, it is also possible to run the applications on different hosts by adjusting routing configurations accordingly.

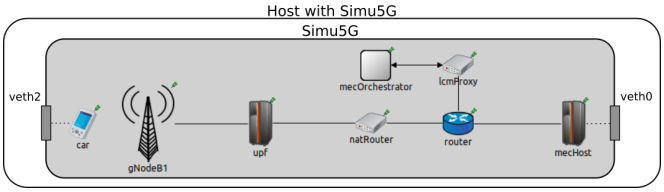

Figure 10: Simu5G network for emulation

The interaction with the real world occurs via two INET's *ExtLowerEthernetInterface* modules included into the car and mecHost modules. Such interfaces can receive real packets by network interface cards attached to them. In our case, such network interfaces are created as Virtual Ethernets (veth). Data packets directed to the simulator are routed to the relative veth attached to the *ExtLowerEthernetInterface* modules. Since all the applications run in the same host, a *natRouter* module must be used to bypass the host operating system and steer the traffic towards the simulator. This way, both real applications send packet to the IP addresses of the *natRouter*, which in turn performs Network Address Translation by changing the destination addresses to the proper real application's addresses. The INI configuration for the *natRouter* router is shown in the topmost part of Figure 11. In the latter, with reference to Figure 10, the IP addresses `10.0.2.1` and `10.0.3.2` are the addresses of the left and right *natRouter* interfaces, respectively.

To conclude the Simu5G network setup, the configuration of the routing tables of all the network devices in the simulated network is needed to enable the forwarding for packets destined to the real applications. Finally, the bottom part of Figure 11 depicts the INI configuration of the *ExtLowerEthernetInterface* modules to allow the communication with the real world.

```
############## natRouter configuration ##############
*.natRouter.ipv4.natTable.config =
      xml("<config> \
            <entry type='prerouting' \
            packetDataFilter='*Ipv4Header and destAddress=~10.0.2.1' \
            srcAddress='10.0.3.2' destAddress='192.168.2.2'/> \
            <entry type='prerouting' \
            packetDataFilter='*Ipv4Header and destAddress=~10.0.3.2' \
            srcAddress='10.0.2.1' destAddress='192.168.3.2'/> \
            </config>")

############## Ext Interfaces configuration ##############
*.mecHost1.virtualisationInfrastructure.numExtEthInterfaces = 1
*.mecHost1.virtualisationInfrastructure.extEth[0].typename = "ExtLowerEthernetInterface"
*.mecHost1.virtualisationInfrastructure.extEth[0].device = "veth2"
*.mecHost1.virtualisationInfrastructure.ipv4.forwarding = true

*.car.numEthInterfaces = 1
*.car.eth[0].typename = "ExtLowerEthernetInterface"
*.car.eth[0].device = "veth0"
*.car.extHostAddress = "192.168.3.2"
*.car.ipv4.forwarding = true
```

Figure 11: natRouter and *ExtLowerEthernetInterface* configuration

As far as the MEC system is concerned, to allow the instantation of a real MEC app in the *appDescriptor* file related to a real MEC app, a field called *emulatedMecApplication* must be inserted. The latter contains *ipAddress* and *port* sub-fields identifying the real MEC app endpoint. This way, the MEC orchestrator is made aware that the MEC app to instantiate is running outside Simu5G and it does not need to request the creation of the MEC app module inside the simulator. Then, the MEC orchestrator communicates to the Device app the IP-port

pair the UE app will need to use to communicate with the MEC app, i.e. the address of the natRouter interface in this case, although real hosts' addresses are also allowed.

Once the Simu5G environment is configured, the OS of the host running all the applications must be configured too. The following commands refers to a host equipped with Linux Ubuntu 18.04 OS. *veth* interfaces are created through the command: `ip link add veth0 type veth peer name veth1` (the same for the couple *veth2-veth3*). After the interfaces have been created, we assign an IP address to them and enable them by `ip addr add 192.168.3.2 dev veth1` (and 192.168.2.2 for *veth3*) and `ip link set veth0` up (for all the 4 interfaces), respectively. Finally, routes to forward the packets within the simulator and its modules have to be added. In particular, packets must reach the Device app, the natRouter and the MEC platform modules, as shown in Figure 12.

```
# UE's eth interface is 192.168.4.1, route for Device app
sudo route add -net 192.168.4.0 netmask 255.255.255.0 dev veth1
# natRouter left interface address is 10.0.2.1
sudo route add -net 10.0.2.0 netmask 255.255.255.0 dev veth1
# natRouter right interface address is 10.0.3.2
sudo route add -net 10.0.3.0 netmask 255.255.255.0 dev veth3
# mecPlatform interface address is 10.0.5.2 (for MEC services)
sudo route add -net 10.0.5.0 netmask 255.255.255.0 dev veth3
```

Figure 12: *veth* interfaces configuration

6 Conclusions

In this paper, we described the modeling of a MEC framework to be integrated with the Simu5G simulator, providing a complete tool for rapid prototyping of MEC applications in MEC systems running on a 5G network. The implementation of standard ETSI APIs, combined with the ability to run Simu5G in emulation mode, allows MEC developers to test and evaluate real MEC apps running on real hosts – possibly interfacing with external frameworks like Intel OpenNESS. Such real applications exploit Simu5G to emulate the 5G transport network between them and to consume MEC services for obtaining information related to the 5G network. We also presented the configurations needed to run an MEC system — and MEC-related applications — with Simu5G, in both simulation and emulation modes.

References

[1] ETSI. MEC 009-V2.1.1 - Multi-access Edge Computing (MEC); General principles for MEC Service APIs, 2019.

[2] ETSI. MEC 010-2-V2.1.1 - Part 2:Application lifecycle, rules and requirements management, 2019.

[3] ETSI. MEC 011-V2.1.1 - Edge Platform Application Enablement, 2019.

[4] ETSI. MEC 012-V2.1.1 - Radio Network Information API, 2019.

[5] ETSI. MEC 013-V2.1.1 - Mobile Edge Computing (MEC); Location API, 2019.

[6] ETSI. MEC 003-V2.2.1 - Multi-access Edge Computing (MEC); Framework and Reference Architecture, 2020.

[7] ETSI. MEC 016-V2.2.1 - Multi-access Edge Computing (MEC); Device application interface, 2020.

[8] G. Nardini, D. Sabella, G. Stea, P. Thakkar, and A. Virdis. Simu5G – An OMNeT++ Library for End-to-End Performance Evaluation of 5G Networks. *IEEE Access*, 8:181176–181191, 2020.

[9] ETSI White Paper. MEC in 5G networks, 2018.

[10] A. Virdis, G. Nardini, and G. Stea. *Cellular-Networks Simulation Using SimuLTE*, pages 183–214. Springer International Publishing, Cham, 2019.