Rapid prototyping and performance evaluation of ETSI MEC-based applications

Alessandro Noferi[1], Giovanni Nardini[2,*], Giovanni Stea[2], Antonio Virdis[2]
Dipartimento di Ingegneria dell'Informazione, University of Pisa
Largo L. Lazzarino, 1, 56122, Pisa, Italy
[*] corresponding author
[1]alessandro.noferi@phd.unipi.it, [2]{name.surname}@unipi.it

Abstract**—** The Multi-access Edge Computing (MEC) standard of the European Telecommunications Standards Institute (ETSI) will enable context-aware services for users of mobile 4G/5G networks. ETSI MEC application developers need tools to aid the design and the performance evaluation of their apps. During the early stages of deployment, they should be able to evaluate the performance impact of design choices - e.g., what round-trip delay can be expected due to the interplay of computation, communication and service consumption. When a prototype of the app exists, it needs to be tested live, under controllable conditions, to measure key performance indicators. In this paper, we present an open-source framework that allows developers to do all the above. Our framework is based on Simu5G, the OMNeT++ based simulator of 5G (NewRadio) and 4G (LTE) mobile networks. It includes models of ETSI MEC entities (i.e., MEC orchestrator, MEC host, etc.) and provides a *standard-compliant* RESTful interface towards application endpoints. Moreover, it can interface with external applications, and can also run *in real time*. Therefore, one can use it as a *cradle* to run a MEC app live, having underneath both 4G/5G data packet transport and MEC services based on information generated by the underlying emulated radio access network. We describe our framework and present a use-case of an emulated MEC-enabled 5G scenario.

Keywords— Simulation, Emulation, ETSI MEC, Simu5G, Real-time, Prototyping

1. Introduction

The European Telecommunications Standards Institute (ETSI) is standardizing Multi-access Edge Computing (MEC) to deliver cloud-computing capabilities at the edge of the network. Besides providing a smaller and more predictable latency, ETSI MEC will enable context-awareness, capitalizing network information such as radio access conditions, user location, etc.. Obtaining this information would otherwise be difficult, if possible at all, in a cloud-based application. This goal is achieved by having MEC communicate with the access network, via the so-called *MEC services*. These are services exported by a MEC platform connected to the access technology, accessible via a RESTful interface, that a MEC application (MEC app, henceforth) can query to acquire the user context. They provide information on the user (e.g., its own radio conditions or location) as well as on the access itself (e.g., current network load or number of users), thus allowing one to create advanced user services. Examples of these user services – already discussed in the ETSI context – are user QoS prediction based on current radio conditions, location and movement pattern; vehicular alerts, e.g. a car approaching a slippery patch of tarmac; robot swarm coordination in a factory.

While, as its very name suggests, ETSI MEC is access-technology agnostic, it is quite clear that its interplay with mobile networks – namely, 4G and 5G today, and 6G tomorrow, will be prominent. This is because ubiquitous, regulated, reliable and secure wireless access is a pre-requisite for marketable user services. In this scenario, MEC services are provided by the underlying 4G/5G radio access, leveraging the information base available at the base stations (eNB in LTE, gNB in New Radio) and in the various network entities involved in the control and management plane. In this paper we refer to this scenario, without explicitly repeating it henceforth. Similarly, we will omit repeating that we focus on the ETSI MEC standard, rather than edge computing at large (e.g., as discussed in [44]-[47]).

MEC app developers will be able to write applications that run through the cellular network *and* interact with the latter by consuming MEC services provided by it. It is therefore important for them to be able to evaluate them for functionality and performance in credible and controllable settings, i.e. taking into account: i) radio resource contention on the data plane of the underlying cellular network; ii) resource contention at the MEC host, given that MEC apps run as virtual machines or containers on a shared computing infrastructure, and iii) contention for access to MEC services (e.g., queueing delays), due to concurrent service requests. Functional and performance testing of MEC apps is required at different stages in the development cycle: early on, one may want to understand the performance implications of alternative designs, e.g., the ensuing communication patterns, before committing to a given solution. Later, one may want to test a prototype of its MEC app in real time, measuring delays and possibly factoring in user interaction. Eventually, a developer may want to showcase its own MEC app, completed and polished, in a live demo, e.g. for sales pitches to prospective financers. All this is made difficult by the lack of suitable tools, and the fact that it is hardly possible to use a live 5G network. There are indeed tools that simulate end-to-end communications on cellular networks – see, e.g., [16],[17]; tools that simulate the ETSI MEC environment [23], or edge computing in general (see, e.g., survey [24]); open-source environments to host MEC apps – see, e.g. [28]. There are also tools that allow one to test ETSI MEC service interfaces – see, e.g., [41],[42]. However, none of these can work in synergy, hence – while certainly helping a developer – they do not allow the kind of support to prototyping discussed above.

In this paper we describe an open-source complete framework to integrate all the above functionalities. It is based on Simu5G [6],[7], an open-source simulation library for 5G (and 4G) cellular networks based on OMNeT++ [8]. Our framework models the whole ETSI MEC infrastructure – including the MEC orchestrator, the MEC host running MEC apps and the MEC platform providing MEC services – in an environment devised for end-to-end, application-oriented discrete-event simulation. All the interfaces between the MEC environment and the application world are designed to be ETSI-compliant. Moreover, our framework can interface with external application endpoints and carry their traffic, and can do so *in real time* [2],[3], acting as a network emulator [4],[5]. A MEC app developer can thus exploit it as a *cradle* for distributed MEC-based applications, which can be both tested for functional compliance and evaluated for performance, maintaining full control over the experimental setup. We describe the modeling of the MEC subsystem within Simu5G and show how its models are designed for scalability. For instance,, they allow one to simulate congestion at the MEC platform efficiently, which in turn enables real-time emulation of a large-scale computation and communication scenario (hundreds of simultaneous MEC apps, complex 5G network with tens of nodes and hundreds of users) on a desktop PC. Moreover, we describe examples of usage, including one with a MEC-based application running live through our framework, showing that setting up a testbed is quite simple and requires inexpensive hardware resources. As far as we know, there are no tools comparable to the one described in this paper. In order to perform a live run of a MEC app in a 5G network, it seems that the only current possibility is to setup an ETSI MEC infrastructure, connected to a 5G network with MEC services enabled, and to run the MEC app through this. This is expensive, time consuming and might still not allow full control over experimental conditions (e.g., to create congestion or radio impairments). The ETSI MEC Industry Specification Group (ISG) has included our framework in the ETSI MEC Ecosystem wiki [43], among a selected few, where it is the only one exhibiting the functionalities described in this paper.

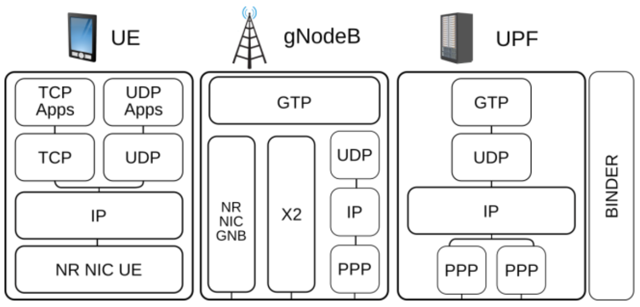


Figure 1 - Main modules of the Simu5G model library

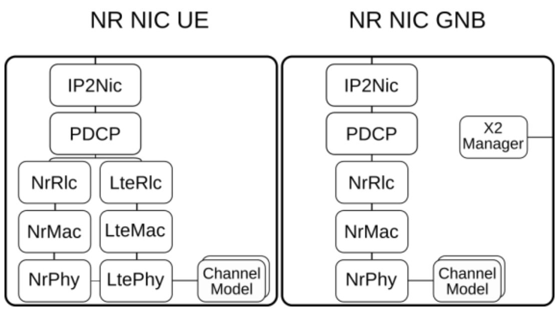


Figure 2 - Internal model of the NR NIC

The rest of this paper is organized as follows: Section 2 provides background information on the technologies we use. Section 3 reviews the related work. Section 4 and Section 5 present the architecture of our software framework, focusing on the model of the MEC infrastructure and the MEC applications and services, respectively. Section 6 discusses how users can benefit from our framework and presents relevant use cases, and Section 7 draws conclusions and highlights directions for future work.

2. Background

In this section, we provide background knowledge required to understand the design choices for our ETSI MEC prototyping framework. More specifically, we present the capabilities of the Simu5G OMNeT++-based library, and we introduce the reference architecture of ETSI MEC.

2.1. Overview of Simu5G

Simu5G [6],[7] is the evolution of the well-known SimuLTE 4G network simulator [25] towards 5G NewRadio (NR). It simulates the data plane of both the core and the radio access networks. Since it incorporates all SimuLTE's functionalities, it allows users to create legacy or mixed 4G/5G scenarios as well. Hereafter, we describe the entities and functionalities of Simu5G that are more closely related to the scope of this paper. We refer the interested reader to [6][7] for more details.

Simu5G is a *model library* for the OMNeT++ discrete-event simulation framework [8]. OMNeT++ allows one to model any system consisting of entities communicating with each other. In OMNeT++, these entities are represented by *simple modules*, and they communicate by exchanging messages through connections between module gates. A module's behavior is coded in C++. Multiple modules can be composed to form a *compound* module. Henceforth, we will generically use the word "module" to denote either simple or compound OMNeT++ modules. Simu5G is interoperable with all the libraries based on OMNeT++, such as INET [9] for TCP/IP-based network technologies, or Veins [11] for vehicular mobility. This allows its users to construct very complex scenarios straightforwardly, by importing and connecting existing libraries, with hardly any extra line of code required. Simu5G, in fact, leverages several models and functionalities taken from INET itself. Figure 1 shows the main modules of Simu5G.

As far as the core network (CN) is concerned, Simu5G defines a User Plane Function (UPF) or Packet GateWay (PGW) and allows users to construct arbitrary CN topologies, where packet forwarding is based on the GPRS tunneling protocol (GTP). For what regards radio access, Simu5G defines gNBs and UEs, which communicate using the NR protocol stack at layer 2. gNBs can be connected to the CN directly, in the so-called StandAlone deployment, or operate in an E-UTRA/NR Dual Connectivity (ENDC) deployment, where LTE and NR coexist. In this last configuration, the LTE eNB acts as a master node and is connected to the CN, whereas the gNB works as a secondary node [12].

Both UEs and gNBs include a NR Network Interface Card (NIC), which models the NR protocol stack. With reference to Figure 2, the NR NIC includes all the sublayers of NR (from the Packet Data Convergence Protocol to the Physical Layer), with 3GPP-compliant behavior. As for the physical layer, Simu5G follows the same approach as SimuLTE, i.e., to model the *effects* of propagation on the wireless channel at the receiver, *without* modelling symbol transmission and constellations. This allows us to compute the Signal-to-Interference-and-Noise Ratio (SINR) at receivers correctly, as well as to compute MAC-layer decoding probabilities at a manageable complexity, and it makes the simulator more evolvable.

Simu5G simulates radio access on multiple carriers, in both Frequency- and Time-division duplexing (FDD, TDD). Different carrier components can be configured with different FDD numerologies or different TDD slot formats. Moreover, different carrier components can have different channel models and MAC-level schedulers. Simu5G incorporates UE handover and network-controlled device-to-device communications, both one-to-one and one-to-many.

Simu5G can run in real time, thanks to OMNeT++ and INET functionalities. In fact, OMNeT++ allows events to be scheduled by a *real-time scheduler*, which matches the simulated time to the wall-clock time at the beginning of a simulation, and slows down event scheduling until the corresponding wall-clock time has expired. This is only possible, of course, if simulated time would otherwise run *faster* than wall-clock time, something which depends on both the hardware/software system that runs OMNeT++, *and* the scenario being simulated. More specifically, the larger the *scale* of the simulation (e.g., the higher the number of UEs and gNBs), the less likely it is that this is possible. Parallel to this, INET allows one to endow modules with *external interfaces* that exchange packets with the outside world. This means that an external application can exchange packets with a module in a simulated network. Since simulation can also run in real time, this allows one to perform *real-time network emulation*, transporting packets between external application endpoints.

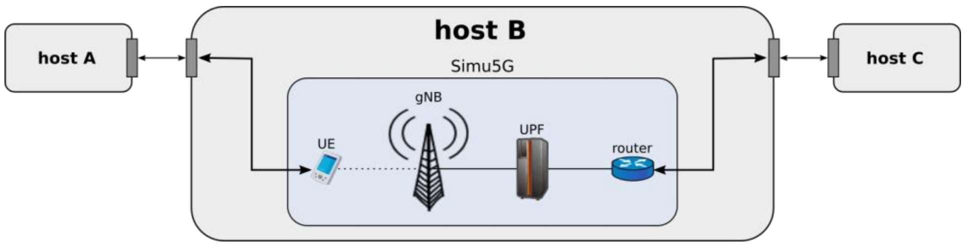


Figure 3 - Simu5G as emulator

Figure 3 shows a simple scenario, wherein host A and C are physically connected to host B, which runs an emulated scenario in Simu5G. Packets sent by host A appear in the emulation at the UE, and packets leaving the rightmost interface of the router are sent to host C. Packets flows similarly in the other direction as well. Both the UE and the router must be endowed with external interfaces for this to happen. Provided that routes are added to host B's operating system, an application on host A can thus reach any entity having an IP address in the emulated scenario (e.g., the UPF). Moreover, it can reach host C, with its packets traversing the emulated scenario. More complex configurations can also be created, such as having one remote host reachable using a public IP address.

Simu5G inherits the above capabilities – being based on OMNeT++ and INET – but is also explicitly designed for scalability, i.e., to allow real-time emulation of relatively large scenarios on a desktop machine. As shown in [3], it offers multiple models of entities (i.e., UEs and gNBs), which can be used for different purposes: on one hand, *foreground* entities run the entire protocol stack and can be connected to external interfaces. On the other hand, *background* entities run only those functions that are necessary to create communication impairments: more specifically, background UEs and gNBs create lifelike interference at the physical layer, and background UEs contend for access to resources at the MAC layer.

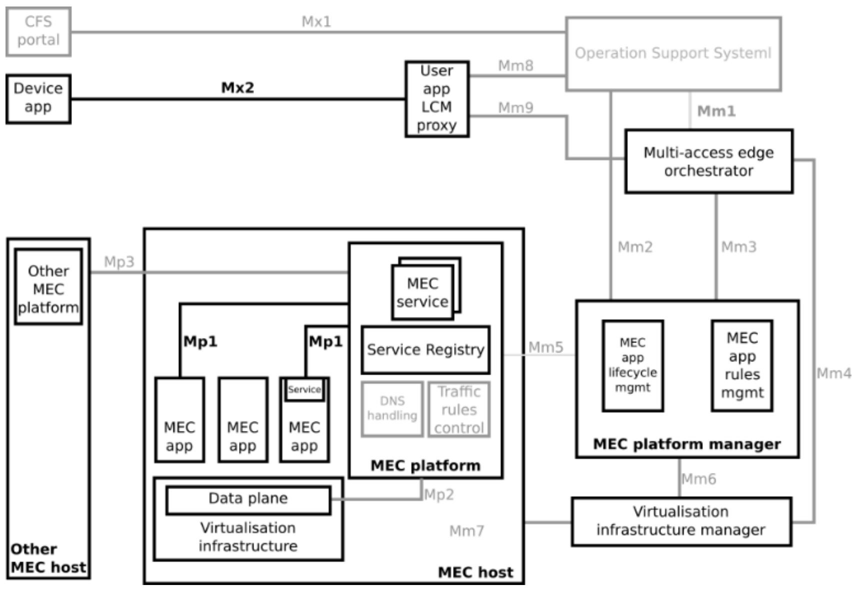


Figure 4 - Main entities in an ETSI-MEC infrastructure (taken from [34]). Black/grey modules are/are not modelled as ETSI-compliant in our framework

Background entities, however, have a reduced protocol stack and do not transmit packets over the air, hence cannot host external interfaces. Background entities create the same impairment that complete entities would, at a much lower overhead, which allows one to extend the scale of scenarios that can be emulated in real time. Work [3] shows that one can emulate scenarios with tens of gNBs and several hundreds of UEs, where two endpoints exchange several Mbps' worth of traffic, all on a desktop PC.

2.2. ETSI MEC Architecture

MEC is a computing infrastructure that can run applications (MEC apps) as virtual machines or containers on behalf of network users, interfacing with the access network (e.g., a cellular network). A MEC architecture has been standardized by the MEC ISG of ETSI [34], which includes some of the major ICT industry players [48]. The ETSI standard covers the functional entities of the MEC architecture, as well as the procedures for a user to instantiate a MEC app. The data-plane exchange between a user and a MEC app is not part of the standard. This section describes the entities that compose the ETSI MEC architecture, with a focus on the ones implemented in our framework, which are highlighted in black in Figure 4. The ETSI MEC architecture includes a *MEC host* level and a *MEC system* level. The MEC host level contains the virtualization infrastructure used to run the virtual machines containing MEC apps. Within it, the MEC platform maintains a Service Registry, i.e. a catalogue that specifies which *MEC services* can be used by MEC apps, and where they are located (i.e., at the MEC host itself, or at a different one). Examples of standardized MEC services are the Radio Network Information Service (RNIS) and the Location Service (LS). Every MEC service advertises its presence to all the Service Registries in a MEC system via the MEC platform manager. When a MEC app queries a Service Registry, the latter returns the (local or remote) endpoints at which the available MEC service should be contacted. This allows MEC apps to consume MEC services located in different MEC hosts.

MEC apps can consume MEC services via standardized MEC APIs, built on a RESTful approach. REST follows a request-response paradigm. In a MEC environment, a MEC app often acts as a client, making requests to MEC services. However, it is sometimes useful to allow the MEC service to notify MEC apps of events at specific times, in a subscribe/notify approach, which is still implemented based on the RESTful pattern [33]. In this approach, a MEC app

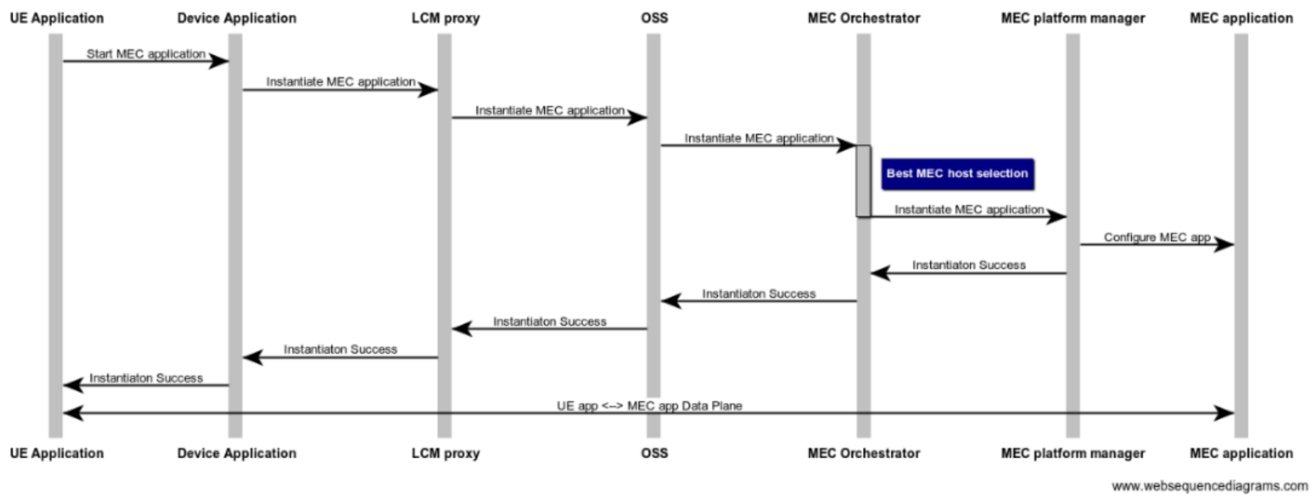


Figure 5 - Sequence diagram of the instantiation of a MEC app on behalf of the user.

registers its interest in a particular resource (subscription phase), and then, when an event occurs that concerns that resource, the MEC service sends a notification to all the subscribers (notification phase).

The MEC system level maintains a global view of the MEC hosts in a MEC system, arbitrates MEC-host resources, and manages the lifecycle of the MEC apps, i.e. instantiation, relocation and termination. Its core element is the MEC orchestrator. The latter receives the requests, after a granting operation made by the Operation Support System (OSS), for instantiation or termination of MEC apps issued by the user's *device application*. The OSS is usually managed by the network operator and deals with authentications and authorizations. The orchestrator then selects an appropriate MEC host for that MEC app, based on the required constraints (e.g., latency), available resources (e.g., memory, disk, CPU), and available MEC services. Then, it instructs the appropriate MEC platform manager to deploy the virtual machine that will run the MEC app. The User application lifecycle management proxy (UALCMP) acts as a bridge between the device application and a MEC system: it forwards requests coming from the device application to the MEC orchestrator, and forwards responses from the MEC orchestrator back to the device application. A MEC deployment can include several *MEC systems*, i.e., orchestrators managing disjoint sets of MEC hosts.

A MEC app is onboarded through an application package. The latter is composed of a bundle of files provided by the application provider, onboarded into the MEC system and used by the latter to instantiate an application. One of such files is the Application Descriptor, which describes the application requirements and rules required to run the MEC app [35].

A UE interacts with the MEC system via two logically distinct entities: the device app and the UE app. The device app interfaces with the MEC system to request specific functions related to lifecycle management of a MEC app (notably, instantiation or termination). The UE app is instead the endpoint of data-plane communication between the UE and the MEC app. Data-plane communication starts after the device app receives confirmation of the instantiation of the MEC app. The sequence diagram of a MEC app instantiation is shown in Figure 5. We remark that neither the interface between the UE app and the device app, nor the data-plane communication between the UE app and the MEC app, are part of the ETSI MEC standard.

The ETSI standard also defines a set of *reference points* [34], i.e. standardized interfaces between the entities of a MEC system. For instance, the Mx2 reference point is used by the device application to request the MEC system to perform operations on a MEC app. The UALCMP is in fact the point of termination of the Mx2 interface towards the device application. The Mp1, between the MEC platform and the MEC app, allows MEC services management (registration and discovery).

3. Related Works

In this section we review the available works on 5G simulation, MEC simulation and MEC services testing. A quick summary of our review is reported in Table 1.

Table 1 – Summary of related works.

		Features				
Ref.	Summary	4G/5G data plane	ETSI-comp. external interfaces	Runs in real time	ETSI MEC services	Integration with other libraries
[14]-[15]	Physical-layer 5G simulators	L1-L2				
[16]	5G LENA for ns3	X				X
[17]	5G-air-simulator	X				
[22]	OpenAirInterface 5G-RAN project	X				
[23]	ETSI MEC simulator				LS only	
[24]	Critical review of Fog/Edge simulators and emulators					possibly
[41]	ETSI MEC Sandbox		X	X	partly	
This work	Simu5G + ETSI MEC extension	X	X	X	X	X

As far as 5G simulators are concerned, there are both those that simulate the physical layer, such as [14]-[15], and those that allow end-to-end, application-level simulations, like Simu5G. In this last category we find 5G-Lena [16] and 5G-air-simulator [17]. As far as we know, neither supports ETSI MEC or allow real-time emulation of a 5G network.

A critical review of the Fog/Edge simulators and emulators available to date is reported in [24]. The tools described in the paper have been developed to evaluate computing infrastructures in terms of deployment scenarios, energy consumption and operational costs. They address Internet of Things application and resource management in an Edge/Fog/Cloud continuum. The most widely used seems to be iFogSim, which does not take network aspects into account too much. The underlying network condition, with increasing QoS requirements from users, has become of paramount importance and therefore needs to be carefully considered as well. More importantly, these tools do not address ETSI MEC and the related functionalities, such as MEC services, that can retrieve information about the underlying network for the edge applications, i.e. MEC apps.

The work most closely related to ours seems to be the ETSI MEC simulator described in [23]. Devised for the ns3 framework, it allows one to simulate simplified models of distributed MEC-based applications over a MEC system, using non ETSI-compliant interfaces. The simulator includes models of MEC orchestrator and app mobility. There are, however, several crucial differences with our work: first, one cannot use real MEC-based applications and run them on this framework, as it does on Simu5G. Second, there seems to be no models for MEC services and MEC platform: the simulator, in fact, includes *dummy* base stations, whose only role is to have UEs associated to them on a proximity basis, without modelling radio communication (in fact, RAN latency in the experiments described in [23] is modelled as a probability distribution). Therefore, on one hand, these base stations cannot provide any lifelike radio information to a MEC platform (e.g., the level of resource occupation, or the channel quality of a mobile); on the other hand, such an architecture also prevents straightforward interoperability with the 4G/5G libraries already available for ns3 (such as 5G-Lena), where the above information could be generated. Last, but not least, to the best of our knowledge, [23] cannot run real-time emulations.

All the above works focus on MEC modelling, and either neglect or abstract away the role of the underlying RAN. We remark that the interplay between the RAN and the MEC does not limit to the former transporting packets generated or consumed by the latter. Rather, and no less importantly, the RAN generates the information that MEC apps use via MEC services (notably, those returned by the RNIS). While it could make at least some sense to abstract away the modelling of RAN communication impairments (e.g., via a delay distribution and a bandwidth), it is just pointless to have the RNIS without an underlying credible and *detailed* model of the RAN. We also remark that an architecture including both the 5G access and the MEC system allows one to test the impact of MEC traffic on the network.

The ETSI group also provides a portal for exploring the functionalities of MEC services. It is called "ETSI MEC Sandbox" [41] and offers an environment where users can choose among different real-time access network scenarios (e.g. 4G, 5G, Wi-Fi) and experiment with ETSI MEC service APIs. RESTful resources can be queried from both browser and existing applications testing MEC application use cases. However, the available settings are limited and the network parameters, like fading, inter-cell interference and multi-carrier components, cannot be modified in order to produce different behaviors. This limits the use of this tool to the sole purpose of experimenting with the MEC services API, without allowing one to customize the network scenarios.

The OpenAirInterface 5G-RAN project [22] is under way as we write, with a schedule foreseen to complete in the second half of 2022. The project promises to deliver an open-source software implementation of a programmable 5G RAN. If and when this project is completed, it will probably be a good tool to experiment with the interplay between ETSI MEC and the RAN: for instance, one may envisage building a system where OAI 5G-RAN interacts with an ETSI MEC infrastructure (e.g., Intel OpenNESS [28]), and UEs can enjoy MEC services in an emulated environment. However, this requires OpenAirInterface 5G-RAN to interface with the MEC system to export the information needed for MEC services, and we have no information that this is included in the current plan of the OpenAirInterface consortium. Even if it was, we are somewhat skeptical that such an environment would provide a MEC app developer with the possibility to test its own MEC apps quickly and under controlled conditions, possibly including large-scale communication or congestion at the MEC host.

Finally, three previous works of ours [13],[2],[27] are related to this one. Work [13] describes the MEC model developed for SimuLTE. Such work is meant to be used for research purposes, i.e. to allow a SimuLTE user to instantiate simplified models of MEC-based applications – similarly to [23], and evaluate their impact on the data plane. While the work described in this paper reuses code from [13], suitably adapted to the 5G environment, its purpose is to serve MEC app developers, rather than networking researchers. Accordingly, it has been re-engineered and enhanced with models of ETSI-MEC components, such as the UALCMP or the device app, that are necessary for seamless emulation of MEC functionalities for a developer. For the same reason, standard-compliant external interfaces to the application world have been implemented. For instance, the use case that we describe in Section 6.1 could not have been emulated in that environment – not even substituting 4G access for 5G. Our paper [2] describes the real-time emulation capabilities of an early release of Simu5G. As a case study, we show therein that Simu5G can provide 5G transport to a MEC app running on Intel OpenNESS [28]. On one hand, the current version of Simu5G is quite different from the one described in [2], especially for what concerns real-time emulation (also thanks to the newer versions of OMNeT++ and INET). On the other hand, the MEC app described therein is a simple client/server video application running on a virtual machine instantiated on a MEC host. All the management-plane interactions are missing, and the service is statically configured offline. In [27], we used Simu5G together with and Intel® CoFluent™ studio [26] to evaluate the performance of MEC apps running over different 4G/5G deployments. CoFluent is a modelling and simulation tool for optimizing, analyzing and predicting the performance of complex systems, that models and simulates HW/SW systems with microinstruction-level accuracy. In [27], the MEC host was modelled within CoFluent, and a video-streaming MEC app was modelled within it. However, the co-simulation framework described in [27] is hardly comparable to what we describe in this paper. In fact, it is based on file exchanges between Simu5G and CoFluent, which run separately. This allows one to compute the round-trip delay, including both communication and computation. However, no feedback between the two simulators can exist (e.g., network conditions influencing the MEC app behavior, e.g. via RNIS, or user behavior depending on MEC app results). This could only be enabled by scheduling *both* CoFluent and Simu5G events in the same unified framework, which would require a considerable amount of work. Moreover, while CoFluent can model a MEC host with high accuracy, its very accuracy makes it unsuitable to model a *large-scale* MEC system, which can instead be modelled with Simu5G.

We conclude this section mentioning some of the MEC-related research works, which are perfect examples of research activities that could benefit from our tool. They indeed have a different focus, as they do not provide a *tool* for experimenting *with* MEC, rather technical solution *for* MEC. On the one hand, there exist several data-driven solutions (e.g.,[44],[45]) that aim at improving MEC performance, either or both in terms of energy consumption and QoS/QoE. As such, data-driven approaches rely on realistic data sets in the design, training and testing phases, which are typically hard to obtain the context of mobile networks. In this respect, our simulator can be used as an effective tool to create realistic scenarios and thus generate credible and statistically meaningful datasets. On the other hand, works defining MEC-based architectural solutions and protocols to extend MEC capabilities (e.g., [46][47]), need to be tested and verified with realistic end-to-end traffic, to demonstrate their benefit for the end-user. Small-scale proof-of-concept deployments are useful to demonstrate the feasibility of the above methods. However, they lack the ability to assess performance at scale, which instead can be done through our tool.

4. An Architecture for Rapid Prototyping of MEC-based Applications

This section presents our framework for rapid prototyping of MEC apps, which can be downloaded from [7]. It is based on Simu5G, which is enhanced with models of the ETSI MEC components described in Section 2.2. A design choice for our framework is to implement in a standard-compliant way (large subsets of) the MEC reference points towards applications, i.e. the Mx2 and Mp1 interfaces. This design choice allows a developer to interact with our framework in the same way it would with a real MEC system: this way, one can interface production-level code with it. Since Simu5G can run as both a simulator and an emulator, and can interface with external code, a developer has the option of writing parts of her MEC app either as Simu5G modules (e.g., a stub UE app for testing the MEC app) or as standalone external applications. In this section, we describe how we modelled MEC entities. In particular, we focus on our model of the MEC system-level entities, the MEC host, MEC platform and services, and application endpoints. Table 2 reports the notation we used throughout this section to describe the modelling of the MEC entities.

Table 2 – Symbols used in Section 4.

Symbol	Definition
R	Processing rate of a MEC host
m	Number of active MEC apps within a MEC host
r_i	Processing rate of MEC app i, under the segregation model
r_i'	Processing rate of MEC app i, under the fair sharing model
μ	Service rate of a MEC service, modelled as a M/M/1 queueing system
λ_f	Arrival rate of requests from foreground MEC apps at a MEC service
λ_b	Arrival rate of requests from background MEC apps at a MEC service
n	Number of queued requests at a MEC service
p_n	Probability for an incoming request to find n queued requests at a MEC service

ρ	Utilization of the M/M/1 system
N	Number of instructions to be executed

4.1. Model of MEC system-level components

Our framework models both the UALCMP and the MEC orchestrator. The UALCMP is an OMNeT++ module, which includes the TCP/IP stack and the application that implements the Mx2 reference point [36]. With this RESTful interface, device applications can request the instantiation and the termination of MEC apps. Such requests are then forwarded to the MEC orchestrator which, after handling them, returns a response that includes the outcome. If the UALCMP is connected to an INET external interface, it can also be contacted by a device application running outside the simulator. Note that – coherently with the ETSI standard – we also allow a device application to join an already instantiated MEC app, in a many-to-one communication model. In this case, the UALCMP simply retrieves and returns the endpoint of such MEC app. For ease of exposition, we do not discuss this possibility further, assuming one-to-one communication henceforth.

The MEC orchestrator is a simple module, connected to the UALCMP. It is configured with a list of the attached MEC hosts. Upon receiving a request for the instantiation of a MEC app, the orchestrator chooses the most suitable MEC host, among those associated to it, according to a user-definable policy which may consider application requirements (e.g., CPU, memory, disk, required MEC services). Next, it contacts the MEC platform manager of the chosen MEC host to request the instantiation of the MEC app, following admission control. A user interested in defining or testing MEC host selection policies can do so by overriding the `findBestMECHost()` method of the MEC orchestrator module. The time duration of the instantiation and termination operations can also be configured.

Interactions between the MEC orchestrator and the UALCMP or the MEC host-level entities are not implemented to be standard-compliant. Instead, they occur using OMNeT++ message passing between modules. This allows us to implement standard-compliant behavior with a simpler, more manageable implementation, without any loss of functionality towards the application endpoints. Moreover, we remark that not all reference points have been standardized at the time of writing, e.g., the interface between the UALCMP and the Operation Support System. For this reason, we could not implement the latter, and instead chose to model the above interactions via a configurable delay in the MEC orchestrator.

Finally, note that our framework allows one to instantiate several MEC systems, each one with its own MEC orchestrator and set of hosts.

4.2. Model of the MEC host-level components

The MEC host is the main building block of the MEC host level architecture. As shown in Figure 6, it includes both management entities, such as the MEC platform Manager and the Virtualisation Infrastructure Manager, and the modules required to run the MEC apps, i.e., the virtualisation infrastructure and the MEC platform. Hereafter, we first describe how the MEC host runs MEC apps. We postpone the description of the MEC platform and MEC services to the next subsection.

A MEC host comes with a set of hardware resources, namely a processing rate (measured in instructions per second), a given amount of memory and disk space. The main duty of a MEC host is to run MEC apps. As outlined before, a MEC app can be either *external* to our framework, i.e. an external application exchanging information with our framework in real time, or *internal* to our framework, i.e., an OMNeT++ module compiled within it. In the *external* case, the MEC host has an external interface towards the MEC app, which runs somewhere else (e.g., on a virtual machine in a desktop computer connected to our framework via a local network). Accordingly, the speed at which that MEC app runs depends on the hosting hardware (e.g., its CPU speed). In the *internal* case, i.e., when MEC apps run within our framework, instead, we need to manage the pace at which a MEC app is executed. In a discrete-event simulator, such as Simu5G, simulated time passes

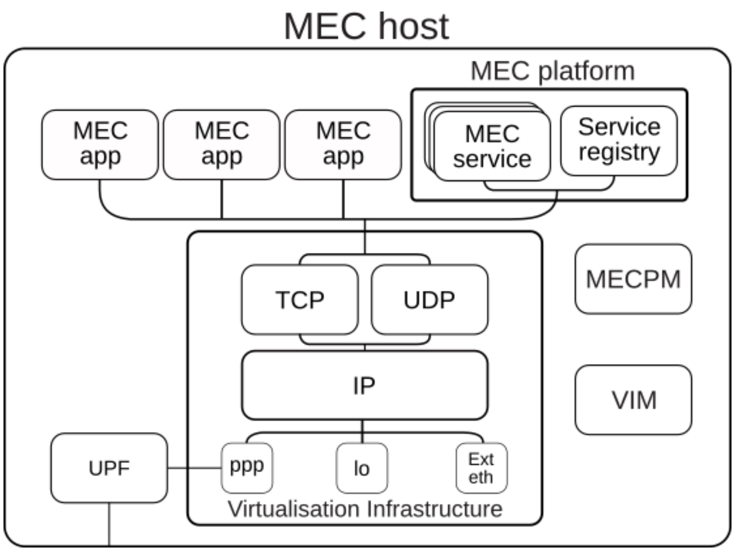


Figure 6 - MEC host modeling within Simu5G

between scheduled events. Messages and packets are events, hence a packet exchange (e.g., a request/response pattern) occurs over a span of simulated time. On the other hand, the MEC application logic is embedded within event handlers, and event handling per se does not consume simulated time: any processing done by such applications is thus instantaneous in simulated time. Therefore, to obtain a more realistic model of the processing operations in a MEC app, we modelled the execution time of a block of code by adding an event representing the required CPU computation, which in turn depends on the level of CPU contention at the MEC host. A MEC host can schedule *internal* MEC apps according to two paradigms:

- *Segregation*, whereby each MEC app obtains exactly the amount of computing resources it has stipulated at the time of admission control, even when no other MEC apps are running concurrently on the same MEC host;
- *Fair sharing*, whereby active MEC apps share all the available computing resources proportionally to their requested rate, possibly obtaining more capacity than stipulated when contention is low.

At the time of instantiation, a device application requests a *computation rate* r_i for the MEC app. The admission control done by the MEC orchestrator checks (among other things) that $\sum_{j=1}^{m} r_j \leq R$, where R is the MEC host's processing rate and m is the number of MEC apps. With segregation, a MEC app computation will run at a rate r_i, regardless of the processing contention on the MEC host. This models non-work-conserving scheduling of the MEC app virtual machines on the MEC host. With fair sharing, instead, a MEC app computation will run at a rate

$$r_i' = r_i \cdot \frac{R}{\sum_{j=1}^{m} r_j}, \tag{1}$$

where the summation at the denominator includes all the MEC apps that are active when the computation is requested. It is obviously $r_i' \geq r_i$, equality holding only when the admission control limit is reached. Fair sharing approximates Generalized Processor Sharing (GPS) scheduling on the MEC host [30]. The approximation is due to the fact that the actual processing rate r_i' is computed *once* when a computation is requested by the MEC app, rather than updated continuously over time. In fact, r_i' changes over time (because other MEC apps start and terminate, modifying r_i'), hence the time at which a computation terminates cannot be calculated once and for all *ex ante*, but must instead be updated every time the summation at the denominator changes. This is a well-known problem with GPS emulation: a quadratic complexity is required for exact tracking of finishing times. However, exact GPS emulation would only bring a significant accuracy improvement with very long continuous computations done by a MEC app. Note that one can instantiate a *dummy* MEC app on a MEC host, whose only purpose is to eat up resources (notably, processing speed) at the MEC host, so as to simulate a given processing load in an efficient way.

To model the time spent by computation requests in an *internal* MEC app, we allow a user to use a special call `calculateProcessingTime(N)`, where N is the number of instructions to be executed. This adds an event in the future, whose firing time is computed based on N, the MEC host processing speed, the scheduling paradigm and – possibly –the number of active MEC apps at the time of the call. After such delay, the block of C++ code modelling the computation phase of an *internal* MEC app is executed.

4.3. Model of the MEC platform

The MEC platform module has its own TCP/IP stack and is connected to the Virtualisation Infrastructure via an Ethernet connection. It contains MEC services and the Service Registry, which are implemented as TCP applications.

MEC Services interface with MEC apps through a RESTful approach. In our framework, they are implemented as HTTP servers and support both the request-response and subscribe-notify communication models. Incoming requests are queued up in FIFO order and served sequentially. A *notification* FIFO queue is also added (different events may generate notifications simultaneously, hence queueing may actually occur). When both are non-empty, the notification queue takes precedence. A MEC service is itself a point of contention, since several MEC apps may request the same service near-simultaneously. Therefore, we need to model delays at a MEC service in a coherent and scalable way. The service time of an HTTP request or a subscription notification event is simulated with a delay computed by a specific method, `calculateRequestServiceTime()`. A user can modify this method and generate service times according to, e.g., the request type, the number of parameters, or a random distribution. By default, the service time is extracted from a Poisson distribution with a configurable mean. Thus, the response time of an HTTP request depends on the number of requests already queued at the server and on their service times.

A user may want to test scenarios with heavy contention at a MEC service (e.g., due to a very large number of concurrent MEC apps). In a MEC system, in fact, requests can arrive not only from the MEC apps instantiated in the same MEC host where the MEC service runs, but also from MEC apps deployed in other MEC hosts [34]. Simulating many MEC apps increases the computation overhead in Simu5G, and this may constrain the size of the scenario that can be simulated or emulated in real time. To solve this, we provide an implementation which allows one to simulate arbitrary loads at a MEC service at a constant computation cost. We distinguish *foreground* MEC apps, i.e., those that are instantiated on a MEC system, and *background* MEC apps, i.e. those whose load we want to simulate, without paying the overhead of modelling the application logic itself. We model a MEC service as an M/M/1 queueing system, where the service rate is μ and the arrival rate of foreground/background requests are λ_f and λ_b, respectively, with $\lambda_f \ll \lambda_b$. Moreover, we assume that foreground and background requests are independent. If $\lambda_f + \lambda_b < \mu$, then the system is stable, and the state probabilities seen by an arriving foreground request are the same as the steady-state probabilities seen by a random observer (PASTA property), which are:

$$p_n = \rho^n \cdot (1 - \rho), \tag{2}$$

where $\rho = (\lambda_f + \lambda_b)/\mu$, and $n \geq 0$ is the number of requests in the queue. When a foreground request arrives at the queue, two cases are given: a) no other foreground requests are in the system, or b) there is already one foreground request in the system. In the first case, then one can extract a value for the number of requests already in the system from (2), let it

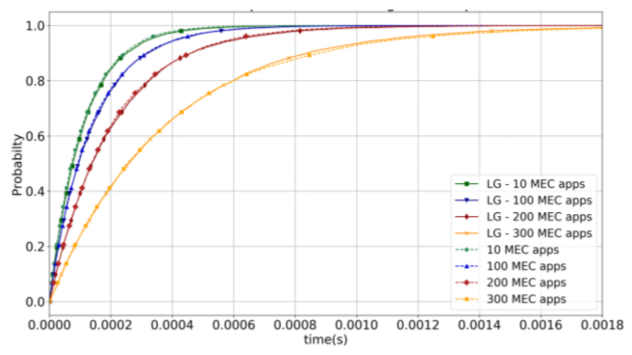


Figure 7 - CDF of the delay of the foreground MEC apps when an increasing number of real/modelled MEC apps interfere at the MEC service.

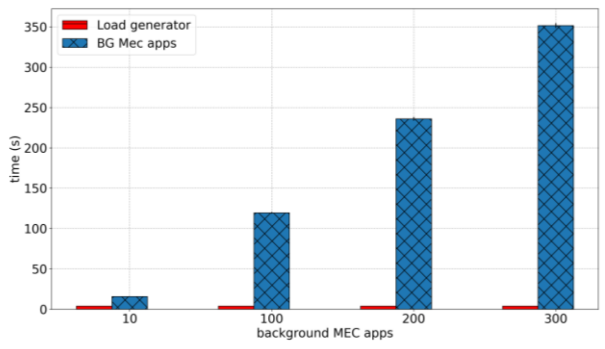


Figure 8 - Wall-clock time required to simulate 180s of a MEC system, using an increasing number of MEC applications vs. the equivalent load generator

be n^*, and schedule the departure of the arriving foreground request $n^* + 1$ service times in the future. This entails extracting a value from an $(n^* + 1)$-stage Erlang distribution with a rate μ.

Assume instead that a foreground request arrives when there is already another in the system. Let t_0, t_1 be the arriving times of the former and current foreground requests, respectively. The number of background requests arrived in $[t_0, t_1[$ is a Poisson random variable with a mean $\lambda_b \cdot (t_1 - t_0)$. Once a value is extracted from that distribution, call it n^*, then one can follow the same reasoning as above, and extract the inter-departure time between the two foreground requests from an $(n^* + 1)$-stage Erlang distribution with a rate μ.

By using the above model, one can store *only* foreground requests in the queue, hence the computation overhead of simulating contention at the MEC service is independent of the background load. If the arrivals and/or services are not exponential, a similar modelling can be used, by substituting the appropriate distributions for the number of requests in the system, the cumulative service time of $n^* + 1$ requests back-to-back, and the number of arrivals within a known interval of time.

To validate the above model, we simulated a simple scenario with a single gNB, a MEC host and three mobile UEs having MEC apps that make periodic requests to the LS running in the MEC host. The LS is loaded by other MEC apps, simulated either as real MEC apps modules or using the above load generator. These send requests exponentially with a rate $\lambda = 0.024$ each, while the three foreground apps send requests every 500ms. The number of background MEC apps varies from 10 to 300 and the simulation time is 180 seconds.

Figure 7 shows the CDF of the response time of *foreground* requests issued by the foreground MEC apps when the number of *background* MEC apps varies. As we can see, results are overlapping – even though requests from foreground MEC apps are periodic, instead of Poisson. However, the computation overhead is quite different. We show in Figure 8 the mean execution time out of 15 independent repetitions, with 95% confidence interval, in the same conditions, on a laptop with an Intel I5 processor. With the load generator, the running time is independent of the number of background MEC apps, and remains fixed at 3.8 seconds. When simulating real MEC apps modules, the running time increases with the number of background MEC apps. Note that, already with 200 real MEC apps, the mean wall-clock time required to run a simulation of 180 seconds exceeds 200 seconds, i.e., simulation time runs *slower than* wall-clock time. This means that emulation would not be workable in that case. By using the above load generator, instead, one could emulate in real time scenarios where MEC services are congested.

5. MEC Applications and Services

In this section, we describe how we model the MEC app, device app and UE app application endpoints, showing what configurations are required depending on the possible deployment options (i.e., internal to Simu5G or real applications running outside Simu5G). Moreover, we discuss how MEC services running within the MEC platform are modeled, and we present the implementation of two ETSI MEC services.

5.1. Model of application endpoints

As already mentioned, Simu5G can exchange real network packets with external applications. Since our MEC framework implements the Mx2 and Mp1 reference points, a developer can develop and test all MEC-related applications as either Simu5G modules, or external applications. In this last case, she can leverage the fact that Simu5G can run in real time to setup live experiments in an emulated environment.

5.1.1. MEC app

Each MEC app must be accompanied by a JSON file describing the Application Descriptor mentioned in Section 2. This file includes the necessary fields to allow onboarding of an application package into the MEC system, such as:

- *appDId*: unique identifier of the app descriptor;
- *appName*: name of the MEC app;
- *appProvider*: provider of the MEC app and of the Application Descriptor;
- *appServiceRequired*: the MEC services needed to run the MEC app;
- *virtualComputeDescriptor*: the MEC host computation resources required to run the MEC app (i.e., memory, disk and CPU).

In the framework, the *appProvider* field is used to specify the module name necessary for creating the OMNeT++ module that implements the MEC app, if it is internal to the simulator (it can be left empty if the MEC app is external).

If the MEC app is external to Simu5G, a field named *emulatedMecApplication* must be specified. This contains the IP address and port sub-fields identifying the external MEC app endpoint. This way, the MEC orchestrator is made aware that the MEC app to be instantiated runs outside Simu5G, hence it returns the MEC app's endpoint to the device app, instead of instantiating it inside the simulator.

Our framework allows a developer to quickly create a prototype of an internal MEC app, by deriving the *MecAppBase* module. The latter is a base class that manages sockets and OMNeT++ events, leaving to the developer only the implementation of the methods called upon reception of messages (e.g. from the Service Registry, a MEC service or a UE). For an external MEC app, on the other hand, the only required configuration is the Service Registry's IP address and port, through which the location (i.e., IP address and port) of the required MEC services can be discovered.

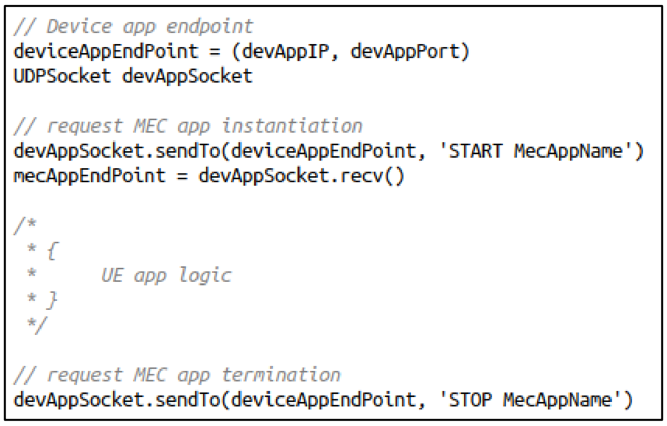
5.1.2. Device app

To facilitate the task of an application developer, our framework provides a simple device app that can request the installation and termination of a MEC app to the UALCMP via the RESTful API implementing the Mx2 reference point. That device app can be contacted by a UE app via UDP, by means of a simple interface that includes messages for the creation, termination and acknowledgment of a MEC app (e.g., START `mecAppName`, ACK `endpoint`). This interface can be used by internal and external UE apps alike. Therefore, a developer only needs to write the UE app logic and simply ask the above device app when a MEC app instantiation (or termination) is needed.

Note that our UALCMP also accepts requests from external device apps, since the interface between the two occurs via the ETSI-compliant Mx2 API. In this case, the external device app only needs to be configured with the address of the UALCMP.

5.1.3. UE app

Similar to the other apps, a UE app can be either internal or external to Simu5G. The first option can be particularly useful when the UE app is meant to run as a stub, e.g. to just issue requests at a predefined rate and record statistics.



```
// Device app endpoint
deviceAppEndPoint = (devAppIP, devAppPort)
UDPSocket devAppSocket

// request MEC app instantiation
devAppSocket.sendTo(deviceAppEndPoint, 'START MecAppName')
mecAppEndPoint = devAppSocket.recv()

/*
 * {
 *      UE app logic
 * }
 */

// request MEC app termination
devAppSocket.sendTo(deviceAppEndPoint, 'STOP MecAppName')
```

Figure 9 - Pseudo-code to execute a UE app in a MEC system

Figure 9 describes the UE app pseudocode necessary to request the instantiation (and termination) of the MEC app, via the built-in device app available in Simu5G. The above design makes it very easy for a developer to port to a MEC environment a preexisting client-server application: the existing server application can be deployed via the MEC framework as a MEC app, whereas the entire client logic is inserted within the curly brackets in Figure 9 with minimal to null modifications.

Table 3 summarizes the steps required to configure the two endpoints of a MEC application in our framework, depending on whether each endpoint is internal (i.e., a Simu5G module), external *and* written for ETSI MEC, or just one side of a non-MEC-native client-server application, which a user wants to run in an ETSI MEC environment. The configurations of the two endpoints are independent, and they can be combined in different ways. In Section 6 we present two use cases, with both endpoints being either external or internal. More use cases (e.g., internal UE app and external MEC app, and vice versa) are packaged with our software [7].

Table 3 – Summary of configurations required for the endpoints of a MEC application.

UE app	
Type of endpoint	*Required steps*
Internal (Simu5G module)	• Write the UE app logic; • Use the provided interface to the internal Device app.
External (complete with own Device app)	• Configure the external Device app with the IP address and port of the UALCMP within our framework.
External (client-side of a non-MEC-native application, without own Device app logic)	• Wrap the UE app logic around the pseudocode of Figure 9, using the provided interface to the internal Device app.
MEC app	
Type of endpoint	*Required steps*
Internal (Simu5G module)	• Write the MEC app logic, extending the MECAppBase class (i.e., implement a sequence of code blocks including logic, simulated processing load, message exchanges with UE app); • Write and onboard the MEC app descriptor on the MEC orchestrator in our framework.
External MEC app (either MEC-native or not)	• Write and onboard the MEC app descriptor on the MEC orchestrator in our framework; • Configure the MEC app with the IP address and port of the Service Registry of the MEC host that runs it within our framework, if MEC services are needed; • Start the external MEC app before Simu5G.

5.2. Model of MEC services

As far as MEC services are concerned, we provide both a general-purpose module for rapidly prototyping ETSI-compliant MEC services, and two useful standard services. Our framework comes with a basic module, called `MecServiceBase`, which implements all the non-functional requirements needed for running an HTTP server. This way, a developer only needs to implement the methods to handle HTTP requests and subscriptions of the RESTful API. The base module also maintains the set of eNB/gNB modules associated to the MEC host.

Two standardized MEC services are currently implemented, namely the RNIS and the LS. The RNIS is used to gather up-to-date information regarding the radio network conditions and the UEs connected to the base stations associated to the MEC hosts [31]. Such information allows MEC apps to leverage real-time information on the network performance, which can be used, e.g., to provide an improved Quality of Experience (QoE) to end-users: for instance, to decrease the video quality of a streaming application while the channel is poor or the cell is overloaded, thus avoiding video rebuffering. A relevant subset of the RNIS API is implemented, which includes the *Layer-2 measures* resource, reporting gNB Layer 2 measurements, such as packet delay, throughput, number of active UEs with downlink/uplink traffic, data volumes, cell utilization or packet data rate. By varying the network configuration, one can carry out evaluations and validations of MEC apps using the RNIS in different network conditions. The RNIS gets its information from gNB modules within Simu5G. Hereafter we show how this is achieved without modifying the existing modules within Simu5G. A dedicated module, named `gNodeBStatsCollector` (collector, from now on), can be added to a gNB to retrieve measures from its NIC modules. This should be instantiated only when necessary (i.e., when a MEC host exposes the RNIS), since it is costly from a processing overhead standpoint. The large overhead is due to the fact that a collector manages several timers used to trigger information-retrieving procedures, and the events that these timers generate slow down the execution time of a simulation. Each logged measure is stored in a `L2Measure` object by the collector. Different aggregators can be used to compute the value to be returned when the RNIS requires it, e.g., average, moving average, or last sample. A user can also implement its own aggregator or configure window timers according to documents [37]-[38]. Some L2 measures of the RNIS involve different sublayers of the NR stack. For instance, some delays are measured from the arrival at the PDCP layer to the reception of the HARQ ACK at the MAC layer. This is handled without modifications to the layers, by adding a new module in the NIC, called `packetFlowManager`, which receives information by the relevant layers when the above events are triggered and maintains the data structures necessary to identify the same payload at different NR sublayers.

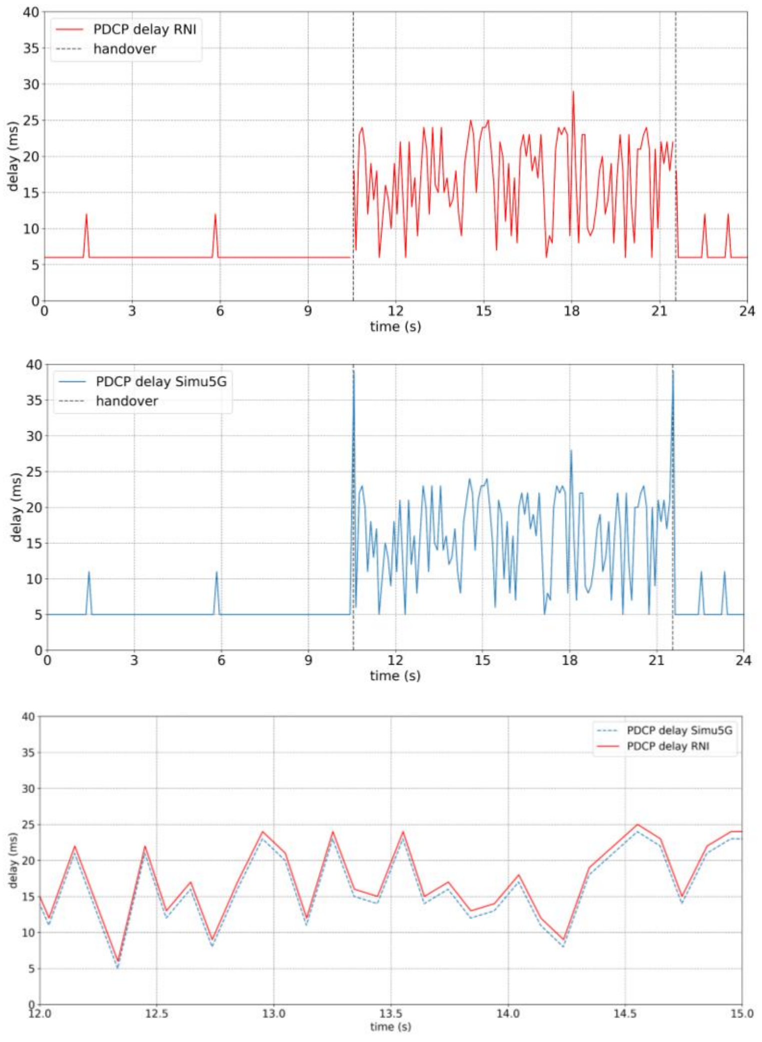


Figure 10 - Downlink packet delay obtained by querying the RNIS (top) or using Simu5G metrics (center), and zoom-in of the difference between the two (bottom) between 12 and 15 seconds.

We validate the results returned by our implementation of the RNIS by comparing them against similar metrics already provided by Simu5G. We simulate a UE that connects to three gNBs over time. The middle one has other 100 UEs periodically receiving packets from a server. In order to show readable and interpretable results, the number of available 5G resource blocks has been downsized to 10. Figure 10, top, shows the PDCP packet delay in downlink returned by the `packetFlowManager`. When the UE handovers to the middle gNB (handovers are marked by dashed vertical lines), the delay increases. Figure 10, center, depicts instead the PDCP delay calculated by Simu5G in the same simulation. It can be observed that the patterns are correlated, suggesting a correct implementation of the RNIS metric. A closer look, shown in Figure 10, bottom, and reporting a zoom of the two above graphs in a unified reference, shows that there is a constant difference of 1ms, which is due to the fact that the RNIS uses HARQ ACK reception to mark the end of the measurement interval, and that occurs 1ms after the UE has received the packet over the air, which is instead the event when the Simu5G statistic is computed. Moreover, delay at handover times are different (as per the spikes in Figure 10, center, around $t = 10$, $t = 21$), since Simu5G starts measuring the delay at the entry *within the NR stack*, whereas the RNIS starts at the entry *at the PDCP layer*. The two are usually the same, except during handover, when Simu5G adds the handover delay to the computation.

The LS provides accurate information about UE and/or gNB position, enabling active device location tracking and location-based service recommendation. The reference API is described in [32] and it is based on the RESTful API originally defined by the Open Mobile Alliance [39]. UE positions are stored and periodically updated on the gNBs to which they are currently connected and are expressed as three-dimensional Euclidian coordinates provided by the INET Mobility model library. A MEC app can query the position of the UEs with different granularities: a single UE, a group of UEs, only the UEs connected to a specific gNB or a group of them. The "UE Area subscription" is also present [32]. It follows the "Area (circle) location notification" subscription defined in [39], whereby a MEC app subscribes to receive notifications when the UE *enters* or *leaves* a circular zone described by its radius and center.

6. Examples of use-cases

In this section we provide an overview about the possible ways in which the proposed framework may be exploited by different categories of users. We then specify the workflows that such users need to follow in order to carry out two different analyses, one requiring emulation, and the other based on simulation.

Figure 11 shows the use-case diagram for our MEC simulation framework. We identify four types of users that may use the latter, namely: a) application developers, b) network operators, c) MEC system owners, and d) researchers. The first three categories of users are tightly related to the industrial context and interact with a MEC system from different perspectives, hence they likely need to carry out disjoint types of performance evaluations.

Application developers are interested in assessing the performance that can be expected from their application when running in a realistic 5G environment, e.g., its round-trip time, including the uplink/downlink latency of 5G communications, the processing time at the MEC host, and the response time of MEC services, under controllable conditions. For instance, MEC-based platooning is envisaged in [49]. This requires closed-loop control, which can be expected to be affected by round-trip delays. A MEC app developer wishing to prototype a MEC-based platoon control application would certainly want to test if, and under what conditions, the performance of the underlying communication/computation system may negatively affect its application, and it can do so using our framework.

Network operators can use our framework to evaluate how their RAN is affected by the presence of a MEC system (e.g., amount of radio capacity allocated to MEC-related traffic); MEC system owners may need to test the utilization of the resources provided by their MEC infrastructure (e.g., number of MEC applications that can be supported given the current CPU resources), as well as to test the API or the performance of (possibly new) MEC services (e.g., what the response time is of a MEC service as a function of the request rate). When MEC system owners are also network operators (which is expected to be the norm, at least initially), they will also be interested into assessing *where* the MEC infrastructure should be placed, with respect to the (mainly hierarchical) cellular network infrastructure. A more capillary infrastructure (i.e., MEC hosts close to gNBs) will likely require higher costs, but may warrant better performance (e.g., shorter round-trip times). On the other hand, deploying MEC servers farther from the network edge is likely to yield opposite benefits and problems. Finally, researchers (e.g., from academia) are potentially interested in all the above use cases.

As shown in Figure 11, all the above use cases eventually require one to run a simulation campaign, which in turn requires one to setup the desired scenario, i.e., network topology and its parametrization. Emulation can be considered as an extended use case with respect to simulations, as it consists in running a real-time simulation where at least one application endpoint runs outside the simulation environment. In this case, one also need to configure the network support of the host operating system. Depending on the evaluation one wants to perform, additional operations must be accomplished, such as implementing both the UE app and the MEC app (for an application developer) or including a MEC service into the framework (for a MEC system owner that wants to evaluate the performance of a new MEC service and/or its impact on the architecture).

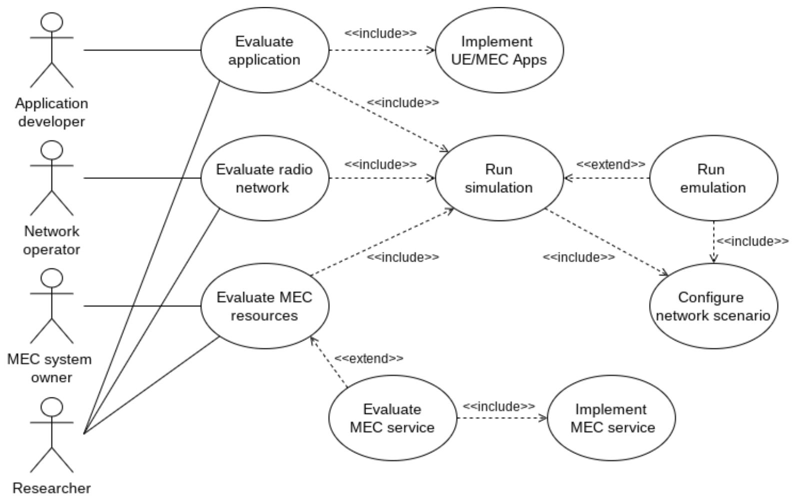


Figure 11 – Use-case diagram of the proposed MEC simulation framework

In the following, we provide insights about the workflow that is required to use the proposed framework. To accomplish that, we consider two different use cases as examples – one using emulation and one using simulation – and describe the necessary steps to complete them. In particular, we first consider the point of view of an application developer that wants to verify the correct mode of operations of a MEC-based automotive application, then we describe a simple use case on which a MEC system owner wants to evaluate two different MEC App allocation policies.

6.1. Emulation use case

In this section we describe the usage of our proposed framework to test a MEC-based application, where both endpoints (UE app and MEC app) are external. We assume that a user of the framework (e.g., a researcher or an application developer in Figure 11) wants to test an automotive application, where a vehicular UE moves in a simulated floorplan and receives alerts when it enters a *danger zone*, e.g., a black-ice area. To do so, the UE application communicate with a MEC app that, in turn, exploits the MEC LS to get the UE position and check whether it is inside the danger zone.

Figure 12 reports the workflow that the user needs to follow in order to realize such use case. If the user is also the application developer, she implements both the client- and the server-side application, i.e., the UE app and the MEC app, respectively. Such applications can be coded in any programming language, and their implementation can be completely unaware of the fact that they will be used in combination with Simu5G, the only requirement being to use IP as network-layer protocol to send and receive packets. More specifically, the UE app is implemented so as it requests the creation of a MEC app to the MEC system, via the Device App. Once the MEC app is instantiated, the applications follow the sequence diagram in Figure 13: the UE app requests the MEC app to monitor a specific zone, then the latter subscribes to the "Area (circle) notification subscription” resource [32] provided by the LS. This means that the LS will notify the MEC app when the given UE *enters* the danger zone, i.e., a circular area with given center and radius. In turn, the MEC app notifies the UE app about the event, and modifies its subscription to the LS, requesting it to be notified again when the UE *leaves* the danger zone.

Moreover, the user needs to provide the application descriptor file, as specified in 5.1.1. The MEC orchestrator implemented by Simu5G uses the *emulatedMecApplication* field of the application descriptor file to obtain the IP address and port used by the MEC app to receive messages from the UE app, i.e., the physical host where the MEC app will run.

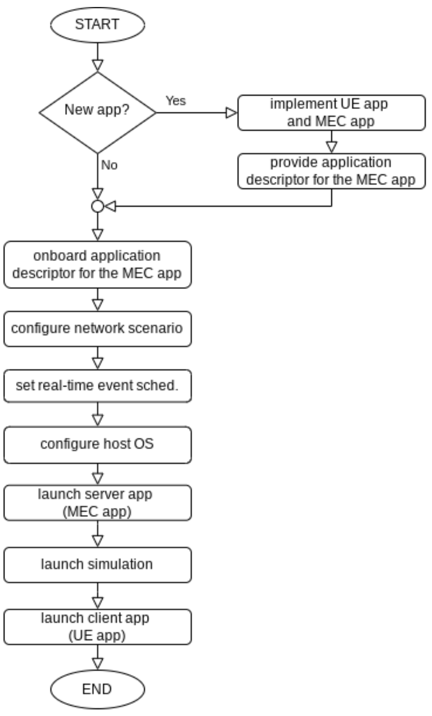

START
New app?
Yes
implement UE app and MEC app
provide application descriptor for the MEC app
No
onboard application descriptor for the MEC app
configure network scenario
set real-time event sched.
configure host OS
launch server app (MEC app)
launch simulation
launch client app (UE app)
END

Figure 12 – Workflow for running an emulated scenario

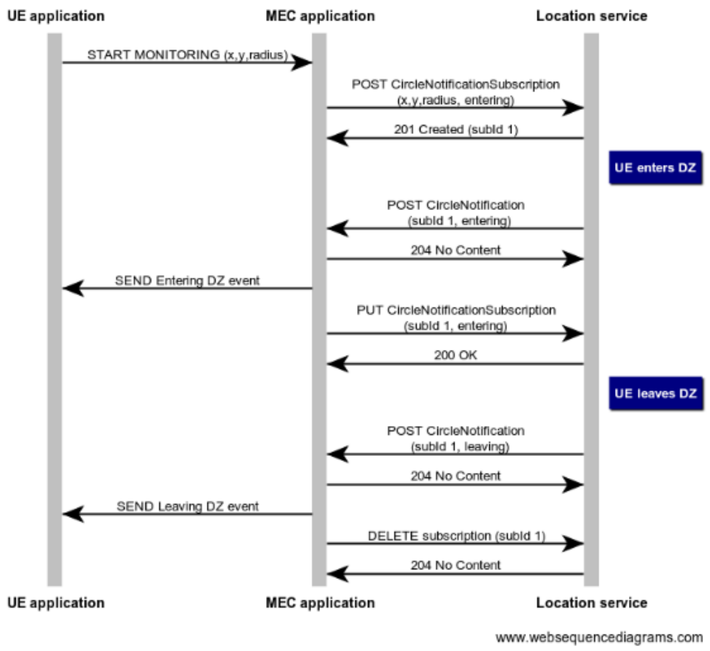

UE application
MEC application
Location service
START MONITORING (x,y,radius)
POST CircleNotificationSubscription (x,y,radius, entering)
201 Created (subId 1)
UE enters DZ
POST CircleNotification (subId 1, entering)
204 No Content
SEND Entering DZ event
PUT CircleNotificationSubscription (subId 1, entering)
200 OK
UE leaves DZ
POST CircleNotification (subId 1, leaving)
204 No Content
SEND Leaving DZ event
DELETE subscription (subId 1)
204 No Content
UE application
MEC application
Location service
www.websequencediagrams.com

Figure 13 - Sequence diagram of the interactions between UE app, MEC app and LS

Once the applications are ready, the network scenario must be configured. In Simu5G, this is done by preparing a Network Description (NED) file and an Initialization (INI) file. The former specifies the modules included in the simulation and their connection, whereas the latter specifies their parametrization. In our example, we setup the scenario in Figure 14, which is composed of a gNB and one UE, i.e., a car equipped with a NR interface, that move towards the danger zone. The UE is equipped with the Device app and it acts as the bridge between Simu5G and the external UE app (i.e., packets sent by the UE app appears as sent by the UE in the emulation). A MEC host is attached to the 5G core network, runs the LS and acts as the bridge between Simu5G and the external MEC app. In order to let Simu5G run as an emulator, the user also need to set the real-time version of the OMNeT++'s event scheduler, as discussed in Section 2.1. This is done by setting the parameter `scheduler-class = "inet::RealTimeScheduler"` in the INI file.

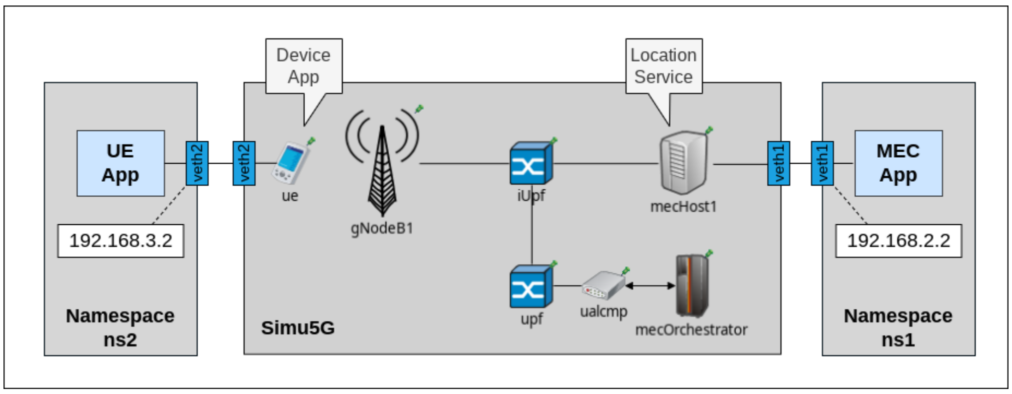
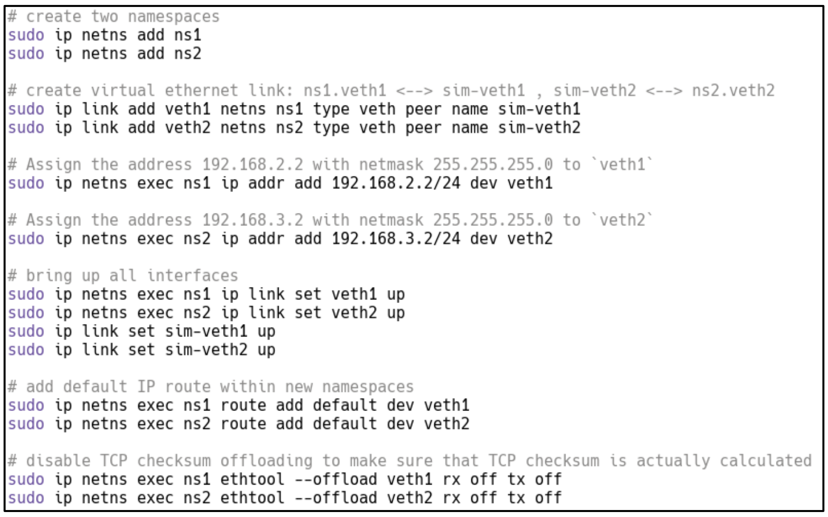


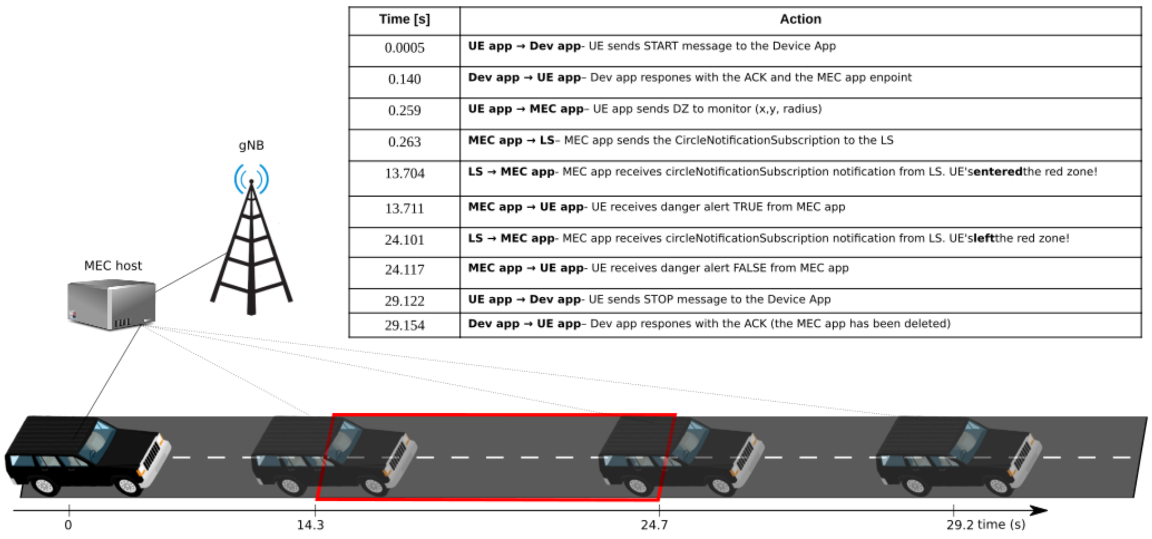
Figure 14 – Testbed and network configuration

The next step is to setup the network configuration of the physical computers implementing the emulation testbed. In a general setting, the UE app and the MEC app reside on different physical computers than the one where Simu5G runs, hence the user must modify the operating system's routing table so that packets coming from the UE app and destined to the MEC app (and vice versa) are correctly routed towards the emulation. In our testbed, we implemented both the UE app and the MEC app on the same computer hosting Simu5G using the *namespace* mechanism provided by the Linux kernel, which allows to run processes on isolated resources of the operating system (e.g., each namespace has its own routing table). As shown in Figure 14, the MEC app is run in a namespace called *ns1*, whereas the UE app is run in namespace called *ns2*. The two namespaces are equipped with a *virtual ethernet* interface (*veth1* and *veth2*, respectively), which allows the apps to exchange data with the instance of Simu5G. Figure 15 shows the list of commands required to obtain the above setting.



```
# create two namespaces
sudo ip netns add ns1
sudo ip netns add ns2

# create virtual ethernet link: ns1.veth1 <--> sim-veth1 , sim-veth2 <--> ns2.veth2
sudo ip link add veth1 netns ns1 type veth peer name sim-veth1
sudo ip link add veth2 netns ns2 type veth peer name sim-veth2

# Assign the address 192.168.2.2 with netmask 255.255.255.0 to `veth1`
sudo ip netns exec ns1 ip addr add 192.168.2.2/24 dev veth1

# Assign the address 192.168.3.2 with netmask 255.255.255.0 to `veth2`
sudo ip netns exec ns2 ip addr add 192.168.3.2/24 dev veth2

# bring up all interfaces
sudo ip netns exec ns1 ip link set veth1 up
sudo ip netns exec ns2 ip link set veth2 up
sudo ip link set sim-veth1 up
sudo ip link set sim-veth2 up

# add default IP route within new namespaces
sudo ip netns exec ns1 route add default dev veth1
sudo ip netns exec ns2 route add default dev veth2

# disable TCP checksum offloading to make sure that TCP checksum is actually calculated
sudo ip netns exec ns1 ethtool --offload veth1 rx off tx off
sudo ip netns exec ns2 ethtool --offload veth2 rx off tx off
```

Figure 15 – Configuration of namespaces and virtual ethernet interfaces



Time [s]	Action
0.0005	**UE app → Dev app**- UE sends START message to the Device App
0.140	**Dev app → UE app**– Dev app respones with the ACK and the MEC app enpoint
0.259	**UE app → MEC app**– UE app sends DZ to monitor (x,y, radius)
0.263	**MEC app → LS**– MEC app sends the CircleNotificationSubscription to the LS
13.704	**LS → MEC app**- MEC app receives circleNotificationSubscription notification from LS. UE's**entered**the red zone!
13.711	**MEC app → UE app**- UE receives danger alert TRUE from MEC app
24.101	**LS → MEC app**- MEC app receives circleNotificationSubscription notification from LS. UE's**left**the red zone!
24.117	**MEC app → UE app**- UE receives danger alert FALSE from MEC app
29.122	**UE app → Dev app**- UE sends STOP message to the Device App
29.154	**Dev app → UE app**– Dev app respones with the ACK (the MEC app has been deleted)

Figure 16 - Timeline of the message events during the execution of the use case

On the Simu5G side, *UE* and *MecHost* modules exchange packets with outside applications using *ExtLowerEthernetInterface* INET modules. Moreover, the user must configure the routing information of each simulated module so that Simu5G knows how to route packets to destinations outside the emulated network. For more details, we refer the interested reader to papers describing Simu5G configuration (e.g., [6] and [1] – describing configuration of MEC scenarios) as well as the website documentation [7].

Finally, the user is ready to run the (external) applications and Simu5G. In the testbed shown above, the user needs to execute the MEC app within namespace ns1 and the UE app in namespace ns2. This is done by launching the commands `ip netns exec ns1 <mec_app_name>` and `ip netns exec ns2 <ue_app_name>`, respectively. Figure 16 shows the timeline of the car moving towards the danger zone, with the relative events captured by the Wireshark tool monitoring the *veth1* and *veth2* interfaces.

As can be seen, the configuration effort is modest, and is mostly devoted to networking details. This is necessary for any scenario where OMNeT++ is run in emulated mode and does not depend on the MEC framework. The latter only requires the creation of the descriptor file for the MEC app and the creation of the network scenario to be executed.

6.2. Simulation use case

We now show how our framework may be used by a MEC system owner to test and evaluate MEC host selection policies at the MEC orchestrator. A MEC host selection policy can be defined by specializing the `findBestMECHost()` virtual method. After implementing the policies to be tested, a user must create the 5G+MEC scenario where she wants to test them, by writing the corresponding NED file. The workflow of all the operations needed to run a scenario is shown in Figure 17.

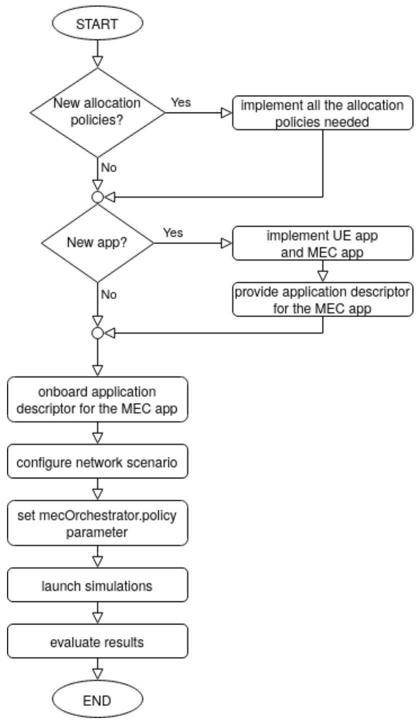

START
New allocation policies?
Yes
implement all the allocation policies needed
No
New app?
Yes
implement UE app and MEC app
provide application descriptor for the MEC app
No
onboard application descriptor for the MEC app
configure network scenario
set mecOrchestrator.policy parameter
launch simulations
evaluate results
END

Figure 17 - Workflow for running a simulated scenario

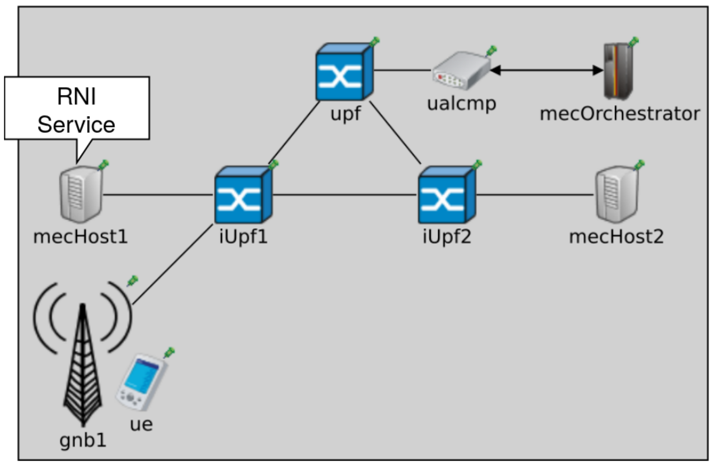


Figure 18 – Simulated network configuration

We compare two MEC host selection policies: *MEC-Service based*, which selects the MEC host whose MEC platform has the MEC services requested by a MEC app, and *Available-Resources based*, which selects the MEC host with the highest available CPU power (we assume that memory and disk are never a bottleneck in this example). In our analysis, we focus on one UE app that periodically requests some computation tasks to its MEC app. Upon the reception of the request, the MEC app first issues a blocking service request to the RNIS, then it performs a computation. Finally, the MEC app sends back a response to the UE app, which can calculate the task response time. We assume that the UE app requests a new computation after 200ms it received the response to the previous request. We evaluate the task response time under the two above-mentioned MEC host selection policies, in the network scenario depicted in Figure 18. In the latter, the MEC system manages two MEC hosts: *mecHost1* runs the RNIS and has less CPU power, while *mecHost2* has more CPU power (three times mecHost1's), but does not have any MEC service locally. MEC apps running on mecHost2 must therefore use the RNIS provided by mecHost1's MEC platform, which can be reached via a link connecting the two iUPFs and having a constant one-way delay of 2ms. The UE app under investigation is run by a UE attached to gNB 1. We study the task response times of experienced by the UE app as the number of *background* UEs increases. The latter are uniformly deployed in the network and each of them communicates with one *background* MEC app that behave exactly as the MEC app described above. This way, load is progressively added to the RAN, the MEC hosts and the RNIS. The main parameters of the simulation are summarized in Table 4.

Table 4 – Simulation parameters.

Parameter Name	**Value**
MEC system configuration	
MEC Host selection policies	[" MEC-Service Based" , "Available-Resources Based"]
# of background UEs/MEC apps	[50…650], step 50
Average service time for the RNIS	~*Exponential* (0.5) ms
Maximum CPU speed	MEC Host 1: 330000 MIPS ; MEC Host 2: 990000 MIPS
MEC app configuration	
CPU speed requirement	500 MIPS
Number of CPU instructions in the MEC app task	~*Uniform* (900000, 1100000)
UE app task period	200 ms
RAN configuration	
Numerology	0 (i.e., TTI duration is 1ms)
Number of Resource Blocks	50
Background UEs offered traffic	2 KB/s (downlink) ; 1.6 KB/s (uplink)
Simulation configuration	
Simulation duration	160s
Warm-up time	10s
# of independent replicas	31

According to the parameters in Table 4, mecHost2 always has more available CPU than mecHost1. This leads the *Available-Resources Based* policy to always select mecHost2. Figure 19 shows the mean (left) and 95th percentile (right) of the task response time (95% confidence intervals are negligible, hence omitted) against the number of background UEs/MEC apps. We can see that, when the number of MEC apps in the system exceeds 400, it is more efficient to host MEC apps on mecHost2. With this choice, in fact, the higher computation power of the MEC host overcompensates the increased round-trip times for accessing the RNIS.

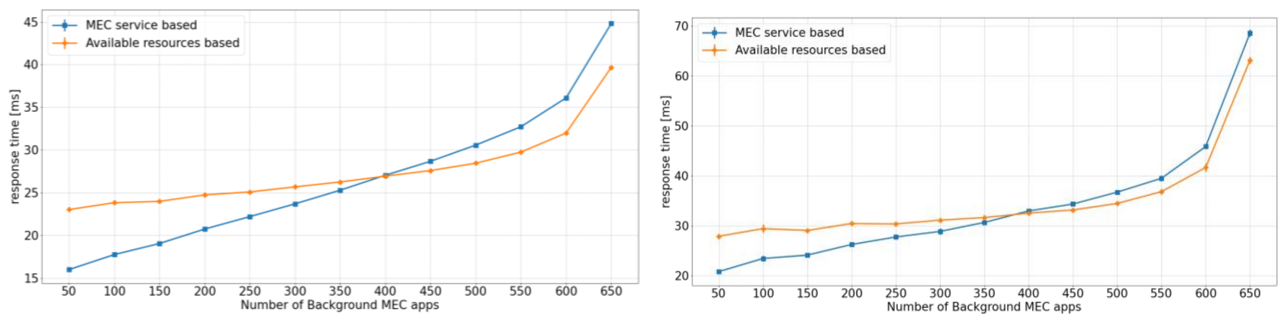
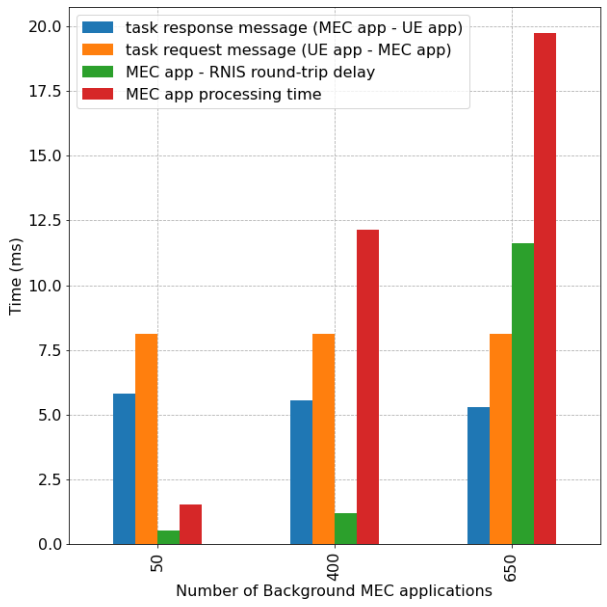


Figure 19 – Average response time for task requests (left) and 95th percentile thereof (right).

The task response time is the sum of several components:

- *network delays* of task request/response messages between the UE and the MEC app: these include uplink and downlink delays of the RAN and – possibly – the constant delay between the two iUPFs, depending on where MEC apps are hosted;
- the *processing time* to execute CPU instructions on a MEC host, which depends on the load and the MEC host itself;
- the *round-trip delay* between MEC apps and the RNIS, which includes both the RNIS response time and – possibly – a constant network delay to get to the RNIS, again depending on where MEC apps are hosted.

Our framework allows one to examine all these delays separately. Figure 20 shows the mean of each of the above delay components with both MEC host selection policies and increasing loads. The increase in the overall response times observed in Figure 19 is due to the loads on the MEC host and the RNIS. The delay on the RAN is fairly constant, since the NR frame is always lightly loaded in this scenario (about 20 Resource Blocks are used out of 50, with 650 UEs); the difference in the task request and response message delays between the two figures is in fact given by the constant delay between the two iUPFs.



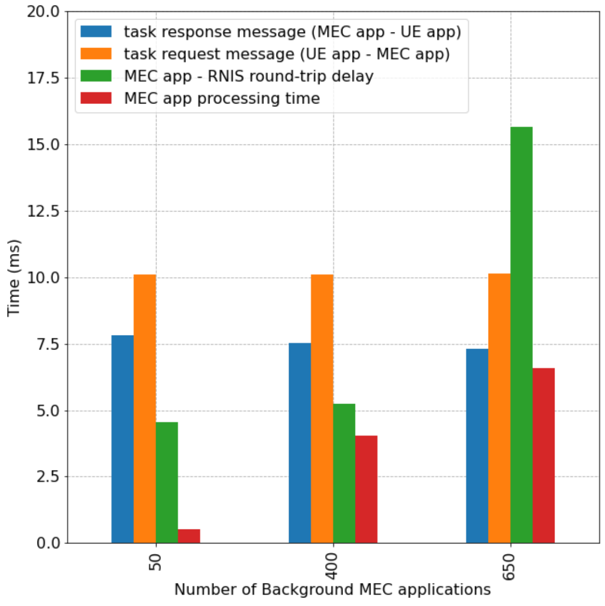


Figure 20 – Breakdown of task response delays for the Service-Based policy (left) and the Available-Resources based policy (right).

This use case is described from the perspective of a MEC system owner that wants to evaluate MEC host selection policies. However, a very similar workflow has to be followed by a MEC developer who wants to test an early-stage prototype of her MEC applications before implementing it for real. She can write an initial version of her application, arbitrarily composing real C++ code blocks, abstract code blocks in the form of processing delays, calls to MEC services, and message exchange between endpoints, leveraging all the already existing code for MEC app lifecycle management and onboarding. Then, she can run simulations with different parameters and network conditions to see how her application behaves in a MEC environment running in a 5G network.

7. Conclusions and Future Work

This work described a framework for rapid prototyping of ETSI MEC applications. Our framework, based on the Simu5G discrete-event simulator, gives a developer two options: the first one is to write application logic (UE app and MEC app) as Simu5G modules. This is quite simple, and allows the developer to test the application logic in a pre-production stage, obtaining reliable performance metrics in a customizable 5G scenario. The other option is to use existing MEC-based application endpoints, and run them through our framework, which provides not only 5G packet transport, but also MEC signaling functionalities and MEC services. This can also be done in real time, e.g. for demonstration purposes or to test interactions with a human end user or other external software. We have described the modelling of the ETSI MEC components in our framework, validated our implementation of MEC services, and showed two use cases – one emulated, the other one simulated – from the perspective of a MEC app developer and a MEC system owner, respectively. We believe that the work described in this paper will be useful to MEC app developers. To the best of our knowledge, there are no tools with similar functionalities available to the developer community.

At the time of writing, the above framework is being used in the framework of the Hexa-X EU project [29]. In that framework, it will support the development, validation, evaluation and demonstration of federated learning of explainable AI models. More specifically, our framework is used to evaluate network protocols for federated learning, where the learning logic can run both on UEs and in MEC systems.

Acknowledgments

This work was partially supported by the Italian Ministry of Education and Research (MIUR) in the framework of the Cross-Lab project (Departments of Excellence), by the Center for Logistic Systems of the University of Pisa, and by the European Commission through the H2020 projects Hexa-X (Grant Agreement no. 101015956).

References

[1] A. Noferi, G. Nardini, G. Stea, and A. Virdis, “Deployment and configuration of MEC apps with Simu5G”, 8th OMNeT++ Community Summit 2021, virtual, September 8-10, 2021, ArXiv abs/2109.12048

[2] G. Nardini, G. Stea, A. Virdis, D. Sabella, P. Thakkar, "Using Simu5G as a Realtime Network Emulator to Test MEC Apps in an End-To-End 5G Testbed", PiMRC 2020, London, UK, 1-3 September 2020

[3] G. Nardini, G. Stea, A. Virdis, “Scalable Real-time Emulation of 5G Networks with Simu5G”, IEEE Access, 2021, DOI: 10.1109/ACCESS.2021.3123873

[4] M. Carson, D. Santay, "NIST Net: a Linux-based network emulation tool", SIGCOMM Computer Communication Review 33, 3 (July 2003), 111–126. DOI: https://doi.org/10.1145/956993.957007

[5] P. Imputato, S. Avallone, "Enhancing the fidelity of network emulation through direct access to device buffers", Journal of Network and Computer Applications, vol. 130, 2019, pp. 63-75, ISSN 1084-8045, DOI: https://doi.org/10.1016/j.jnca.2019.01.014.

[6] G. Nardini, D. Sabella, G. Stea, P. Thakkar, A. Virdis "Simu5G – An OMNeT++ library for end-to-end performance evaluation of 5G networks", IEEE Access, vol. 8, pp. 181176-181191, 2020, DOI: 10.1109/ACCESS.2020.3028550

[7] Simu5G Website, http://simu5g.org, accessed July 2022.

[8] OMNeT++ Website: https://omnetpp.org, accessed July 2022.

[9] INET Library Website. https://inet.omnetpp.org, accessed July 2022.

[10] M. Hu, X. Luo, J. Chen, Y.C. Lee, Y. Zhou, D. Wu, "Virtual reality: A survey of enabling technologies and its applications in IoT", Journal of Network and Computer Applications, vol. 178, 2021, ISSN 1084-8045, DOI: https://doi.org/10.1016/j.jnca.2020.102970.

[11] C. Sommer, R. German, F. Dressler, "Bidirectionally Coupled Network and Road Traffic Simulation for Improved IVC Analysis," IEEE Transactions on Mobile Computing (TMC), vol. 10 (1), pp. 3-15, January 2011

[12] 3GPP TR 38.801 v14.0.0, “Study on new radio access technology: Radio access architecture and interfaces (Release 14)”. March 2017

[13] G. Nardini, A. Virdis, G. Stea, A. Buono, "SimuLTE-MEC: extending SimuLTE for Multi-access Edge Computing", 5th OMNeT++ Community Summit 2018, Pisa, Italy, 5-7 September 2018

[14] Y. Kim et al., "5G K-Simulator: 5G System Simulator for Performance Evaluation," 2018 IEEE International Symposium on Dynamic Spectrum Access Networks (DySPAN), Seoul, 2018, pp. 1-2, doi: 10.1109/DySPAN.2018.8610404.

[15] M. Müller, F. Ademaj, T. Dittrich, et al. "Flexible multi-node simulation of cellular mobile communications: the Vienna 5G System Level Simulator". J Wireless Comm. Network 2018, 227 (2018). doi.org/10.1186/s13638-018-1238-7

[16] N. Patriciello, S. Lagen, B. Bojovic, L. Giupponi, "An E2E simulator for 5G NR networks", Simulation Modelling Practice and Theory, Volume 96, 2019, doi.org/10.1016/j.simpat.2019.101933.

[17] S. Martiradonna, A. Grassi, G. Piro, and G. Boggia,"5G-air-simulator: an open-source tool modeling the 5G air interface", Computer Networks (Elsevier), 2020.

[18] Simnovus Callbox website: https://simnovus.com/products/callbox/, accessed October 2021.

[19] Viavi solutions website, available at: https://www.viavisolutions.com/, accessed October 2021.

[20] Polaris Networks 5G emulators website: https://www.polarisnetworks.net/5g-network-emulators.html, accessed October 2021.

[21] Keysight technologies website, available at https://www.keysight.com/, accessed October 2021.

[22] OpenAirInterface 5G Ran project website https://openairinterface.org/oai-5g-ran-project/, accessed October 2021.

[23] S. Massari, N. Mirizzi, G. Piro, and G. Boggia,"An Open-Source Tool Modeling the ETSI-MEC Architecture in the Industry 4.0 Context", Proc. of IEEE Mediterranean Conference on Control and Automation (MED), 2021

[24] A. Aral, V. De Maio, "Simulators and Emulators for Edge Computing", in: Edge Computing: Models, technologies and applications, Editors: J. Taheri, S. Deng, IET. 2020

[25] A. Virdis, G. Stea, G. Nardini, "Simulating LTE/LTE-Advanced Networks with SimuLTE", in Obaidat M.S., Kacprzyk J., Oren T., Filipe J. (eds) "Simulation and Modeling Methodologies, Technologies and Applications", Springer, 2015

[26] Intel CoFluent Studio, available at: https://www.intel.com/content/www/us/en/cofluent/overview.html, accessed October 2021.

[27] A. Virdis, G. Nardini, G. Stea, D. Sabella, "End-to-end performance evaluation of MEC deployments in 5G scenarios", MDPI Journal of Sensor and Actuator Networks, 9(4), 57, 2020, DOI: 10.3390/jsan9040057

[28] Intel OpenNESS, available at: https://www.openness.org, accessed October 2021.

[29] Hexa-X European Project: https://hexa-x.eu, accessed October 2021.

[30] A. K. Parekh and R. G. Gallager, "A generalized processor sharing approach to flow control in integrated services networks: The single node case," IEEE/ACM Trans. Networking, vol. 1, pp. 344–357, June 1993.

[31] ETSI GS MEC 012 v2.1.1, "Mobile Edge Computing (MEC); Radio Network Information API", 2019

[32] ETSI GS MEC 013 v2.1.1, "Mobile Edge Computing (MEC); Location API", 2019

[33] ETSI GS MEC 009 v3.1.1, "Multi-access Edge Computing (MEC); General principles, patterns and common aspects of MEC Service APIs", 2021

[34] ETSI GS MEC 003 v2.2.1, "Multi-access Edge Computing (MEC); Framework and Reference Architecture", 2020

[35] ETSI GS MEC 010-2 v2.1.1, "Multi-access Edge Computing (MEC); MEC Management; Part 2:Application lifecycle, rules and requirements management", 2019

[36] ETSI GS MEC 016 v2.2.1, "Multi-access Edge Computing (MEC); Device application interface", 2020

[37] ETSI TS 136 314 v15.1.0, "LTE; Evolved Universal Terrestrial Radio Access (E-UTRA); Layer 2 – Measurements (3GPP TS 36.314 version 15.1.0 Release 15) ", 2018

[38] ETSI TS 136 314 v16.0.0, "5G; NR; Layer 2 measurements (3GPP TS 38.314 version 16.0.0 Release 16)", 2020

[39] OMA-TS-REST-NetAPI-TerminalLocation-V1-0-1-20151029-A: "RESTful Network API for Terminal Location"

[40] OMA-TS-REST-NetAPI-ZonalPresence-V1-0-20160308-C: "RESTful Network API for Zonal Presence"

[41] ETSI Sandbox Website: https://try-mec.etsi.org, accessed October 2021.

[42] Location API sWebsite: https://networkbuilders.intel.com/commercial-applications/links-foundation, accessed October 2021.

[43] ETSI MEC ecosystem wiki, https://mecwiki.etsi.org/index.php?title=MEC_Ecosystem, accessed June 2022

[44] V. Balasubramanian, F. Zaman, M. Aloqaily, S. Alrabaee, M. Gorlatova and M. Reisslein, "Reinforcing the Edge: Autonomous Energy Management for Mobile Device Clouds," IEEE INFOCOM 2019 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), 2019, pp. 44-49, doi: 10.1109/INFCOMW.2019.8845263.

[45] V. Hayyolalam, S. Otoum, Ö. Özkasap, "Dynamic QoS/QoE-aware reliable service composition framework for edge intelligence.", Cluster Computing 25, 3 (Jun 2022), 1695–1713. https://doi.org/10.1007/s10586-022-03572-9

[46] I. Al Ridhawi, M. Aloqaily, Y. Kotb, Y. Al Ridhawi, Y. Jararweh, "A collaborative mobile edge computing and user solution for service composition in 5G systems", Transactions on Emerging Telecommunications Technologies, 29(11), e3446.

[47] V. Balasubramanian, M. Aloqaily, F. Zaman and Y. Jararweh, "Exploring Computing at the Edge: A Multi-Interface System Architecture Enabled Mobile Device Cloud," 2018 IEEE 7th International Conference on Cloud Networking (CloudNet), 2018, pp. 1-4, doi: 10.1109/CloudNet.2018.8549296.

[48] Members of the MEC ISG at ETSI, https://portal.etsi.org/TB-SiteMap/MEC/List-of-Members, accessed July 2022

[49] C. Quadri, V. Mancuso, M. Ajmone Marsan, G.P. Rossi, "Edge-based platoon control". Computer Communications, 181: 17-31 (2022)