ICT-318784 STP TROPIC

Distributed computing, storage and radio resource allocation over cooperative femtocells

D61

Performance assessment of the femto-clouding

Contractual Date of Delivery to the CEC:	**30th February 2015**
Actual Date of Delivery to the CEC:	**18th May 2015**
Author(s):	**Fabrizio Gambetti, Francesca Fogliuzzi, Enrico de Marinis (DUNE), Antonio Pascual, Olga Muñoz, Sandra Lagén, Adrián Agustín (UPC), Miguel Angel Puente (ATOS), Sergio Barbarossa, Stefania Sardellitti (SAP)**
Participant(s):	**DUNE, UPC, ATOS, SAP**
Workpackage:	**6**
Est. person months:	**49,5**
Security:	**Public**
Dissemination Level:	**PU**
Version:	**e**
Total number of pages:	**101**

Abstract:

This document describes the simulator developed in WP6, activity 6A1, as part of the demonstrator of the project. The purpose of the simulator is the evaluation of the system envisaged by TROPIC by modelling both the LTE-EPC infrastructure and the cloud processing implemented on a cluster of Small Cells empowered with computing capabilities. This enhanced Small Cells are the core of the TROPIC innovations and, therefore, the simulation approach is only possible for evaluating the Cloud platform inside the standard LTE. The simulator has been built upon the *ns-3 framework* with the *LENA* module for the LTE-EPC model. It makes available accurate models for both the radio LTE and the network communications. On the contrary, the whole Cloud platform, with the virtualized computing environment, has been implemented totally from the scratch according to the models and procedures proposed in WP5.

Keyword list: LTE, ns-3, cloud processing, evaluation platform

Document Revision History

DATE	ISSUE	AUTHOR	SUMMARY OF MAIN CHANGES
9 May 2015	a	DUNE, UPC	Final document
12 May 2015	b	DUNE, UPC	Final revision
18 May 2015	e	UPC	Editorial workout

Executive Summary

This deliverable describes the simulator developed in WP6, activity 6A1, as part of the demonstrator of the project. The purpose of the simulator is the evaluation of the system envisaged by TROPIC by modelling both the LTE-EPC infrastructure and the Cloud processing implemented on a cluster of Small Cells empowered with computing capabilities. This enhanced Small Cells are the core of the TROPIC innovations and, therefore, the simulation approach is only possible for evaluating the Cloud platform inside the standard LTE.

In section 2 a quick overview of the existing simulators (mainly for LTE) is shown, as result of the initial activity of 6A1. Among them, *LENA-ns3* is the one choosen for starting the development. The architecture and the main peculiarities of the simulator are described in section 3, with an overview of the simulation approach, the basic structure of nodes and the interfaces with their data flow. In sections from 4 to 6, the most important three simulation models are described. In particular, for the *LTE model* and the *EPC model* their original implementation is first briefly illustrated, and then the new features are shown. Section 7 explains how the external libraries are used inside the simulator for the evaluation of algorithms that cannot be directly implemented. Sections 8 to 10 illustrate the simulation scenario and the setup of the main models implemented. The results of the simulations are shown in section 11, followed by conclusions.

Table of Contents

References

[Baldo-Miozzo-01]	N. Baldo, M. Miozzo, M. Requena-Esteso, J. Nin-Guerrero - *A new model for the simulation of the LTE-EPC data plane* - WNS3 2012
[Baldo-Miozzo-02]	N. Baldo, M. Miozzo, M. Requena-Esteso, J. Nin-Guerrero - *An Open Source Product-Oriented LTE Network Simulator based on ns-3*
[Lena-Manual]	Centre Tecnològic de Telecomunicacions de Catalunya (CTTC) - *LTE Simulator Documentation - Release M6*
[NS3-Model]	ns-3 project - "*ns-3 Model Library - Release ns-3.19*"
[NS3-Source-Help]	http://www.nsnam.org/docs/release/3.18/doxygen/classes.html
[TROPIC-D21]	Z. Becvar et al. - "*Scenarios and requirements*", - ICT-318784 STP TROPIC, July 2014
[TROPIC-D22]	F. Lobillo Vilela, A. Juan Ferrer, M. Á. Puente, Z. Becvar, M. Rohlik, T. Vanek, P. Mach, O. Muñoz, J. Vidal, H. Hariyanto, F.X. Ari Wibowo, M.Goldhamer, E. De Marinis, F. Gambetti, F. Lo Presti -"*Design of network architecture for femto-cloud computing*" - ICT-318784 STP TROPIC, February 2013
[TROPIC-D311]	A. Agustin, J. Vidal, O. Muñoz, A. Pascual, S. Barbarossa, S. Sardellitti, P. Di Lorenzo, A. Baiocchi, M. Sarkiss, M. Kamoun, G. Vivier "*Building blocks for MP2MP energy-constrained communication systems*" - ICT-318784 STP TROPIC, July 2013
[TROPIC-D321]	S. Barbarossa, S. Sardellitti, P. Di Lorenzo, A. Baiocchi,M. Sarkiss, M. Kamoun, M. Goldhamer, S. Lagen, Adrian Agustin, Josep Vidal – "*Cooperative and distributed interference estimation-detection*" - ICT-318784 STP TROPIC, October 2013
[TROPIC-D41]	Tropic deliverable report - *HeNB virtualisation and design of femto-cloud management layer*
[TROPIC-D42]	E. Calvanese Strinati, J. Oueis, M. A. Puente, F. Lobillo Vilela, A. Juan Ferrer, F.Lo Presti, V. Di Valerio - "*Adaptation of virtual infrastructure manager and implemented interfaces*" - ICT-318784 STP TROPIC, February 2014
[TROPIC-D52]	M. A. Puente, F. Lobillo, Z. Beckvar, F. LoPresti, A. Wibowo - "*Distributed Cloud Services*" - ICT-318784 STP TROPIC, February 2014
[TROPIC-D51]	Z. Becvar, M. Vondra, J. Plachy, M. Rohlik, S. Barbarossa, S. Sardellitti, P. Di Lorenzo, A. Baiocchi, O. Muñoz, A. Pascual, M.A. Puente, F. Lobillo Vilela, E. de Marinis, F. Gambetti, F. Lo Presti, H. Hariyanto, A. Herutomo, L. Chandra, F. A. Laksono, F.X. Ari Wibowo, M. Goldhamer, M. Sarkiss, M. Kamoun, J. Oueis, E. Calvanese Strinati - "*System level aspects of TROPIC femto-clouding*" - ICT-318784 STP TROPIC, December 2014
[R4-092042]	3GPP R4-092042 "Simulation assumptions and parameters for FDD HeNB RF requirements"

List of abbreviations & symbols

BLER	Block Error Rate
CPU	Central Processing Unit
DES	Discrete-Event Simulation
DL	DownLink
EPC	Evolved Packet Core
EPS	Evolved Packet System
FDD	Frequency Division Duplex
GTP	GPRS Tunnelling Protocol
GTP-U	GTP User plane
HARQ	Hybrid Automatic Repeat reQuest
HeNB	Home enhanced Node B
LENA	Lte-Epc Network simulAtor
LTE	Long Term Evolution
MAC	Media Access Control
MI	Millions of Instructions
MIMO	Multiple-Input Multiple-Output
MIPS	Million Instructions Per Second
MSG	Message
NCL	Neighbor Cell List
NI	Number of Instructions
PDCP	Packet Data Convergence Protocol
PDN	Packet Data Network
PGW	PDN GateWay
PHY	PHYsical layer
RAN	Radio Access Network
RB	Resource Block
RBID	Radio Bearer ID
RLC	Radio Link Control
RRC	Radio Resource Control
SAP	Service Access Point
SC	Small Cell
SCC	Small Cell Cloud
SCeNB	Small Cell enhanced Node B
SCeNBce	Small Cell enhanced Node B cloud enabled
SCM	Small Cell Manager
SGW	Serving GateWay
SISO	Single-Input Single-Output
SRB	Signaling Radio Bearer
TDD	Time Division Duplex
TEID	Tunnel End-point Identifier
TFT	Traffic Flow Templates
TH	Thread
TS	TimeSlice
UE	User Equipment
UL	UpLink
VCPU	Virtual CPU
VM	Virtual Machine

1 INTRODUCTION

This document is the final report of activity 6A1, where the simulator for the overall system assessment has been developed. A very wide range of abstraction levels have to be taken into account for the proper and effective evaluation of the new features proposed by TROPIC. The fundamental concept, in fact, is the offloading of demanding applications, from the user device towards a cluster of Small Cells, able to offer a Cloud Service for the processing of the application, as the traditional Cloud, but with an improved experience, thanks to their position closer to the user. This means that the simulator has to be able to model the communications between UEs and Small Cells over the LTE-EPC infrastructure, the network communication among the nodes of the cluster and also the processing of an application over the typical virtualized computing platform used for offering a Cloud Service. Therefore the following main models have been implemented:

- an abstraction model of the radio communications that encompasses all the layers of the LTE stack to allow the optimization of the radio resources;
- the EPC model that implements all the nodes of the standard, allowing the network communications;
- a model of the processing over a Cloud with all its components, that is, an Hypervisor for managing the Virtual Machines where the application are executed, along with a Scheduler for the assignement of the shared resources among the Virtual CPUs.

Only considering these main models it is evident their different level of abstraction. In order to avoid the implementation of such a big number of characteristics and focusing only on the new features of the project, a proper simulation platform has been selected as initial development core. The final choice, among the free-access simulators taken into account, has been *ns-3* which can satisfy many of the needed constraints provided that the LENA module for the LTE model is included.

In the next section we show the structure and characteristics of the simulator implemented, derived, of course from *LENA-ns3*. We then enter progressively into the details of the specific features implemented for TROPIC. With the purpose of a full understanding of the development, a quick description of the implementation of the models provided by LENA-ns3 is included. The modifications with respect to the initial models and the new ones introduced are described even though, because of the extension of the "objects" involved, the details included cannot be complete.

The model of the Small Cell Cloud, with the full virtulized platform, represents the wider model added to the starting platform and developed, completely from the scratch, according to the abstraction model proposed in D51. The same model has also been used in the implementation of a Centralized Cloud for the evaluation of the Cloud Services offered by the traditional Cloud Providers.

The results are gathered from different sets of simulations for the evaluation of the behavior, and the performance of the system concludes the document. The assessment of the system performance is basically focused on the aspects that TROPIC expects to improve, that is, the energy spent by the UE and the latency of demanding applications, with the offloading on the Small Cell Cloud. The performance of this type of offloading is compared to both the local processing, with the application executed on the user device, and the offloading towards the Centralized Cloud.

As shown in the last section, offloading towards the Small Cell Cloud seems to be able to guarantee the improvement in terms of overall latency, at least until the number of UEs that require the service to their Small Cell is not too big, even if this limit is (no surprise) highly influenced by the total resources and the type of application. Regarding the energy, the reduction of the battery consumption can be obtained even when the latency constraints cannot be satisfied anymore by the SCC.

2 OVERVIEW OF THE EXISTING SIMULATORS

The analysis of the new features proposed by TROPIC requires the simulation of the entire LTE structure, from the radio stack to the EPC network, in order to reproduce all the operations that allow the offloading of an application from a User Device on a cluster of Small Cells.

This is why the selection of an accurate LTE simulator, as a basis of the development, has been the first essential step in WP6; the alternative would have been the development of the LTE architecture as well, but this is out of the scope of project and resources wasn't allocated for this purpose.

In the field of open source simulators, three projects stand out for the features implemented and the accuracy of the models adopted:

1. **LTE-simulators**: this refers to a family of simulators, both system-level and link-level, developed by the University of Vienna (*Institute of Telecommunication - Technische Universitat Wien*) and based on the Matlab platform.
2. **LTE-sim**: this is a system-level simulator, C++ based, for Linux environment, developed by the University of Bari (*Telematics Lab - DEE - Politecnico di Bari*).
3. **LENA-ns3**: *ns-3* is a generic network simulator for which a multitude of additional modules are developed by the open source community to model different typologies of network. The LTE module is done by the *CTTC - Centre Tecnològic de Telecomunicacions de Catalunya*.

These three LTE simulators have been studied to gather the information required to make a selection and adopt the most suitable platform for the purpose of TROPIC. Table 1 summarizes the main features of these simulators, the most relevant from the point of view of a selection, for a quick comparison among them.

In the last row the table is shown the number of files and directory enclosed in development tree of the corresponding project (for the releases available at the time of the research). It is just a simple indicator of the complexity of these simulators and, although it cannot be considered a good estimate, it can provide information about the learning curve for these the simulators.

	LTE simulator (Vienna Un.)	**LTE-sim (Bari Un.)**	**LENA-ns3**
Environment and language	Matlab (object-oriented)	Linux C++	Linux C++/Pyhton
LTE stack	Accurate lower layers	Full stack	Full stack
Network element (EPC)	No EPC network entities modeled	• eNB, • HeNB, • MME/GW	• eNB, • HeNB, • S-GW / P-GW, • MME
Propagation and channel model	Wide selection of models (including the WINNER model)	3GPP models	3GPP models
Computational resources and memory	High computing resources and execution time	Very limited resources required	Medium amount of resources required
GUI	Output only	None	None
Complexity of the project	files: 368 directories: 38	files: 318 directories: 36	files: 2.278 directories: 271

Table 1. LTE simulators characteristics

In the end of the study of these simulators, based on the official documentation, on the source code and on simple simulations with default configurations, the *LENA-ns3* has been chosen, mainly for the following reasons:

- the presence of a model for the EPC entities;
- the accuracy of the model for the LTE radio stack and for the EPC entities and protocols;
- the models made available by the *ns-3* framework, as the IP stack for packet-level simulation of network communication;
- the limited computational resources required, especially if compared with the simulator developed with Matlab;
- the excellent software structure and its modularity.

Finally, it is worth to mention that the PHY abstraction model, needed to take into account the residual error of the physical layer at system-level, in the last two simulators is obtained through link-level simulations made using the simulator of the first project (Vienna University).

Also for the Cloud computing was explored the possibility of using an available simulator on which develop the algorithms and the features studied in TROPIC. In this field, most of the simulators are focused on Grid computing, but a network of computer used for distributed computing is very different from the approach needed for a correct simulation of a Cloud, based on the concept of virtualization.

On the basis of the researches carried out, the most important simulator of Cloud computing is *CloudSim* and the valid alternatives are really few. It is a Java based simulator, developed by the University of Melbourne, and its main characteristics are listed below:

- support for modeling and simulation of virtualized server hosts, with customizable policies for provisioning host resources to virtual machines;
- support for modeling and simulation of large scale Cloud computing data centers;
- support for modeling and simulation of federated clouds;
- support for user-defined policies for allocation of hosts to VMs and policies for allocation of host resources to VMs;
- support for modeling and simulation of energy-aware computational resources.

The major weakness of *CloudSim* is the network model because it uses a really poor model to simulate the communication among the nodes. The simulator doesn’t implement any network entity, as router or switch, and the exchange of messages among the nodes is modeled by a simple matrix of delays; each element of the matrix expresses the latency for the transfer of a message between a couple of nodes in the network and is used to compute the arrival time of the a message.

This would severely restrict the ability of analysis in TROPIC, where the use of an accurate network model is required to evaluate many of the characteristics of the new proposed solutions. This is one of the reasons why the Cloud part of the simulation will be implemented from the scratch inside the simulation framework of *ns-3*.

3 SIMULATION ARCHITECTURE

3.1 Overview of the simulation platform

One of the innovative features proposed by TROPIC is the implementation of Cloud Services over a cluster of LTE Small Cells, with enhanced capabilities, for the offloading of high demanding applications, from the point of view of computing and/or energy. Unlike the conventional services offered by Cloud Provider, with their large server farms far from the users, the Small Cell Cloud can offer an improved Quality of Experience, being the computing resources always close to the UE.

According to the nature itself of the project, which envisages the introduction of the cloud technologies used in the cloud computing, that is a Virtualized Environment, in the existing LTE-EPC infrastructure, the simulator can be thought, conceptually, as composed by the following two main parts:

1. Radio part: it encompasses all the functionalities of the LTE radio communication among UEs and serving cells, both macro and small cell, and also the EPC network for the end-to-end IP communication with nodes in the Core Network.

2. Cloud part: it contains all the elements for the implementing and managing the Small Cell Cloud for remote processing of UE applications.

This distinction is solely for the sake of clarity during the description, because the Radio and the Cloud part are strictly related (physically and functionally) and, therefore, their models cannot be implemented as objects totally distinct. An example is the new Small Cell with cloud computing capabilities, the SCeNBce: the radio section for LTE communications belongs to the Radio part, while the computing section, hosting the Virtual Machines on which the applications are executed, belongs to the Cloud part, but in the real system they are part of the same "object". The same consideration can be extended to the Small Cell Cloud Manager (SCM), the new node in charge of the management of the computing resources of the Small Cells: it is the "main actor" of the new Cloud capability but it is placed inside the LTE infrastructure, as new node inside the EPC.

The platform selected as simulation core for implementing the demonstrator of the project is the *ns-3 framework*, with the *LENA* module for the LTE-EPC model (developed by CTTC - Centre Tecnològic de Telecomunicacions de Catalunya, but currently integrated inside *ns-3*). Hereinafter simply ***LENA-ns3***. The accuracy of the models provided by the LENA module has been one of the key points in the process of platform selection, because it allows testing the new models proposed by TROPIC, without having to build the whole LTE architecture in advance. Inside this framework the model of a Cloud has been implemented totally from scratch:

- the existing model of the Small Cell has been extended adding the computing capabilities through the creation of a Virtualized Environment for the execution of the applications offloaded from the UE;
- the new computing capability of the Small Cell is fully modeled thanks to implementation of a set of Virtual Machines, with their Virtual CPU, managed by an Hypervisor. The model is completed by the processing Scheduler for the management of the access to the shared computing resources;
- a new entity, the Small cell Cloud Manager (SCM), has been introduced for the supervision of this cluster of computing nodes (now called *SCeNB cloud enabled → SCeNBce*) and for the management and the optimization of the resources of the Small Cell Cloud (SCC);
- a protocol has been introduced for the communications and data exchange among the nodes of the cluster.

The main characteristics of the *ns-3 framework* and its basic structure can be found in the official documentation, but it is worth to remind here some key points:

- LENA-ns3 is a *system-level simulator*. This entails that the link-level algorithms studied in WP3 and WP5 need a proper abstraction model to be implemented in the simulator.
  Because of the requirements in terms of simulation time and computation resources, it isn't possible to simulate the PHY layer, with the sufficient details (symbol level), in a typical system-level simulation scenario, where a great number of entities are deployed (unlike the link-level simulations, normally limited to a single eNB and one or a few UEs). To take into account at system-level the error due to the propagation in the radio channel, the simulator uses a proper PHY layer abstraction model.
  The abstraction model, however accurate, is not able to provide most of the parameters used in the link-level algorithms; this is the reason for the need of an abstraction model also for implementing what has been studied in the previous WPs.

- LENA-ns3, as described in next sections, is an *event-driven simulator*. This means that the simulation doesn't evolve over time using a sequence of fixed simulation intervals to track the changes in the system (as in the time-driven simulators), but only when an event occurs, the status of the system is updated. Even if the simulation *event-queue* is common for the Radio part and the Cloud part, their progress is independent, unless one part doesn't produce an event towards the other. This is important because the events generated by the LTE model have a granularity which is finer the one produced by the Cloud model, in order to correctly simulate the radio resources allocation and perform the evaluation required by the standard.

- LENA-ns3 is a *packet-level network simulator*. This characteristic implies that the communication among the nodes is implemented with the exchange of real packets, built using the specific protocol stack used in each link (the LTE stack in the radio links and the IP stack in the network links). This is another important aspect for the system evaluation.

It is important to notice the different time scale needed to properly simulate and analyze the behavior of the Radio part and the Cloud part.

At the radio level, the granularity of the model has be at least that of the Resource Block (RB), that is 1 ms, because this is the fundamental unit being used for resources allocation. With the adoption of a larger granularity, it is not possible to model accurately packet scheduling and inter-cell interference. The reason is that, since packet scheduling is done on a per-RB basis, an eNB might transmit on a subset only of all the available RBs, hence interfering with other eNBs only on those RBs where it is transmitting.

In the Cloud side, the computing events occur with a scale of tens of milliseconds or more and the analysis of the dynamics requires longer observation windows. The simulation of a Cloud service, with the remote running of an application, doesn't need to take into account all the events for the scheduling of the processing resources as system events.

The model implements the resources assignment among simultaneously running processes, but the scheduling algorithm is used only to find the computation time of the each process, based on the time and the processing resources assigned. The end of task processing is the corresponding event pushed inside the system event queue, not all the single events in which the scheduler changes the state of the processes and the shared resources assignment to them (a numeric example in 6.3.3).

3.2 Simulation technique

The simulation technique adopted in TROPIC is the *discrete event-driven* simulation (DES) which models a system as discrete sequence of events in time. Each event occurs at a particular instant in time and produces a change of state in the system; between two consecutive events, no change in the system is assumed to occur and thus the simulation can directly jump in time from one event to the next.

This approach is opposite to that used in *time-driven* simulators in which the simulation continuously tracks the system dynamics over time; time is broken up into small time slices and the system state is updated according to the set of activities happening in the time slice. The time-driven simulation approach is adopted in the link-level simulators with a short simulation interval, chosen on the basis of the frequency of the simulated signals.

Because discrete-event simulations do not have to simulate every time slice, they can typically run much faster than the corresponding continuous simulation.

Conceptually, the simulator keeps track of the events, generated by all the entities in the simulation scenario and stored inside the *event queue*. The events are scheduled to execute at a specified simulation time and are, therefore sorted by time inside the queue. The job of the simulator is to execute these events in sequential time order. Once the completion of an event occurs, the simulator will move to the next event and the simulation will be over when the event queue is empty, or at the stop-event scheduled at specified time.

From the highest level of abstraction, the execution of a simulation needs a few elements:

1. a number of objects that generate and execute events;
2. an event queue where events are stored;
3. a scheduler responsible for inserting (ordering in time) and removing events from the queue;
4. a representation of the simulation time;

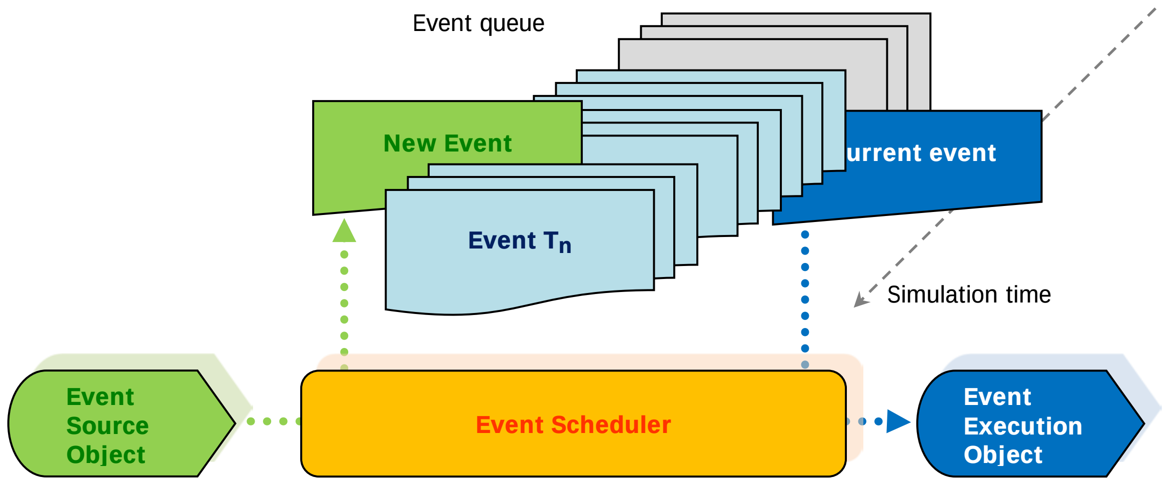


Figure 1. Discrete event simulation

The adoption of the *ns-3* platform as a starting point for the development has made available, of course, the whole infrastructure for the events management. Therefore, for every new entity introduced inside the simulator, the only job to be done for its management is the creation of its proper simulation events for the interaction with the other elements of the system.

3.3 Nodes structure and basic data flow model

The communication among the entities in the simulation scenario may be described using the simple concept of *Nodes* that access a *Channel* among them, through a *NetDevice*. The *Channel* is the physical connection and the *NetDevice* is the I/O interfaces used to transfer packets over this link.
This scheme is independent from the type of the channel used, but *NetDevices* are strongly bound to *Channels* of a matching type.

A *Node* is like a "container" to which applications, stacks, and communication interfaces are added. Inside the node, therefore, an *Application* uses the corresponding *Protocol Stack* to send and receive messages across the channel to and from the other nodes. The interface between the Application and the Protocol Stack is a socket (socket-like API).

A *Node* can have multiple *NetDevices* to communicate with nodes on different channels. This is the case of a SCeNB that is connected to the UEs by a radio channel and to the EPC nodes by a wired channel. When the SCeNB is "cloud enabled" (SCeNBce), it will have a second network device for the communication with the SCM and, if needed, also with the other Small Cells of the cluster.

The creation of a node is completed with its configuration. The set of configuration parameters is different for each type of nodes, depending on the channel; among these, applicable to the UE nodes, there is the mobility, with the possibility of the mobility model selection.

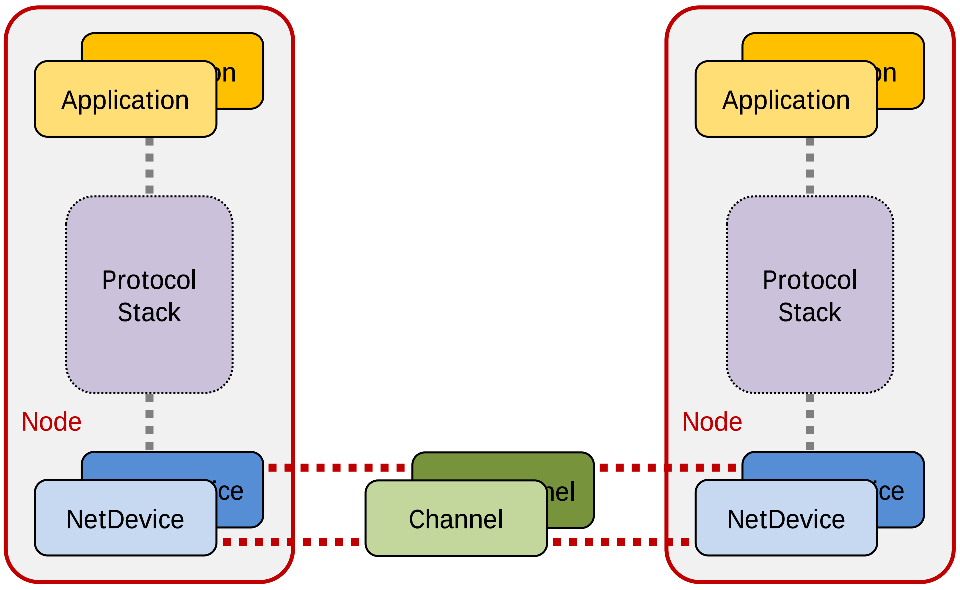


Figure 2. Simulation nodes structure

It should be noted that the *ns-3 framework*, adopted as simulation platform for WP6, implements the packet-level network model for the communication among the nodes. This guarantees the accuracy required for the analysis of the solutions studied by TROPIC.

3.4 Simulation entities

The *ns-3* framework is basically a network simulator based on the *packet-level model*. This means that every single packet is simulated and each node in the network implements the whole protocol stack used in the communication. The same is done in the *LTE model* in order to allow the full-chain communication from a UE to a generic Host in the IP network.
This characteristic is fundamental for testing the Offloading algorithms and the whole *Cloud model* with the real exchange of messages among the nodes belonging to the cluster and the cluster manager.

This simulation approach implies that each “object” inside the scenario is modeled as a *Node* with its own *NetDevice*, specialized for the channel used in the communications, and the *Protocol Stack* specific for such channel. Moreover, as shown in Figure 3, the communication between a couple of nodes requires the setup of a *link* among their *NetDevice* for the complete definition of the characteristics of “channel” and the management of the packets exchange during the simulation.

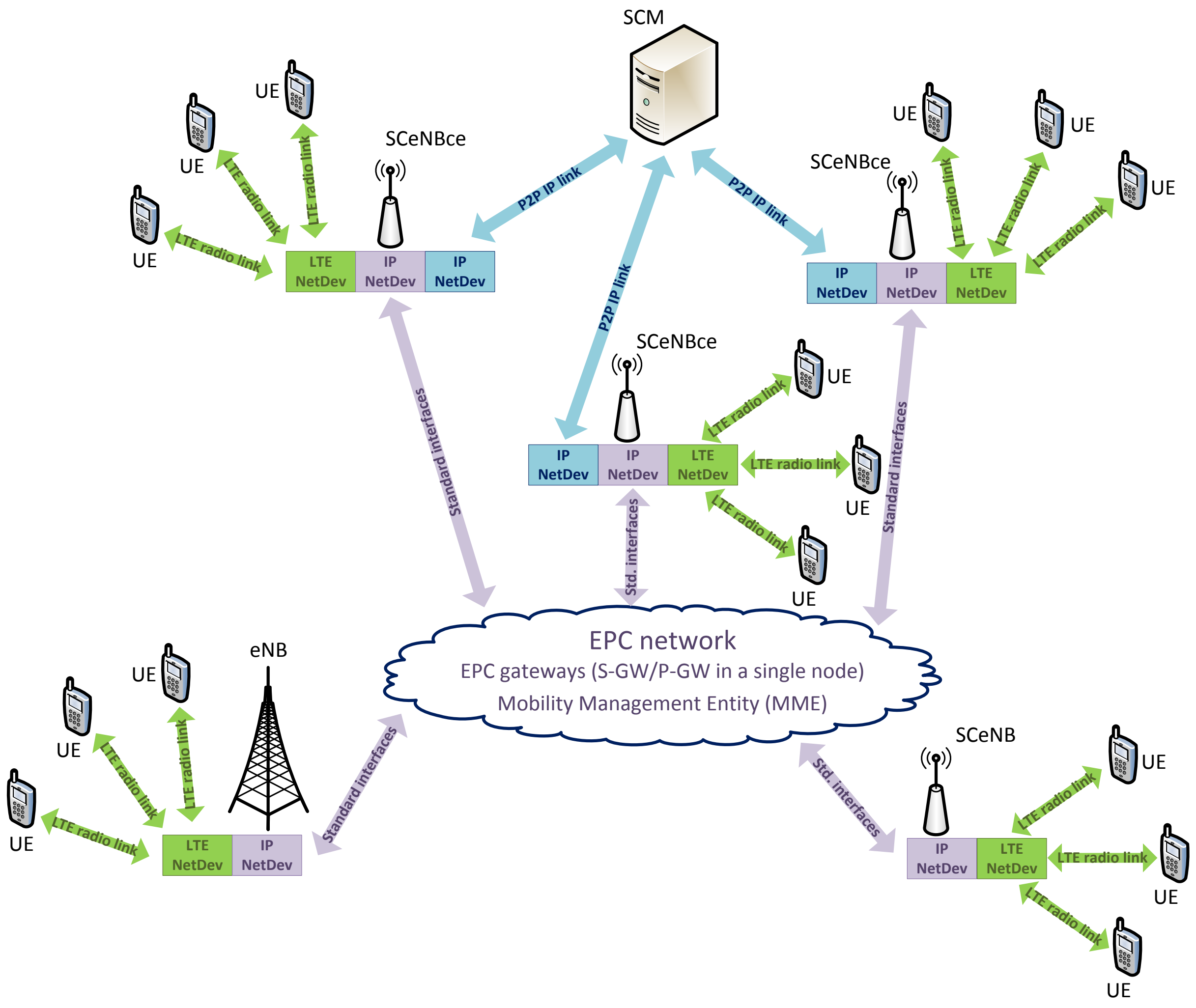


Figure 3. Node object in the simulation scenario

In detail, the “entities” modeled by the simulator and deployed in the simulation scenario are the following:

- **Macro Base Station (eNB)** - The original LENA model is used and it doesn’t require any further modification, but only a possible fine-tuning of the configuration parameters. In terms of simulation node, it has two *NetDevices*: one, with the LTE stack installed, is used for the radio link with the UEs, and the second, for the link towards the EPC nodes, with the IP/GTP stack.

- **Small Cell eNB (SCeNB)** - The Small cells have exactly the same model as the eNB, but different configuration parameters, as for instance, the transmission power. This is absolutely correct since, in effect, a Small cell is basically an eNB with reduced coverage area, but has the same functionalities of an eNB.

Furthermore, when a Small cell an indoor node (*Home eNB*), the appropriate propagation model is used to take into account the effect of the walls and the different radio environment inside the buildings.

- **User Equipment (UE) -** The model provided by LENA has been extended adding a model for the battery consumption, with both the radio communications and the application processing (local part) as sources of energy drains.
  Moreover, the UEs have been differentiated depending on the cell they are attached to: the UEs served by a SCeNB with cloud capabilities (SCeNBce) have the additional modules for the management of the cloud related communications, for the offloading and for the local processing of an application.
- **EPC gateways: Serving Gateway (S-GW)** and **Packet Data Network Gateway (P-GW) -** The LENA module models the two gateways in a single node without the implementation of the S5/S8 interface between them. The approach proposed by TROPIC for the integration of the cloud platform in the existing LTE-EPC infrastructure has minimum impact on this latter, as illustrated in [TROPIC-D52] (section 5.1). This is why there is no need to change this model and the S-GW/P-GW node is used as is.
- **Mobility Management Entity (MME) -** The available model for this node is sufficient to the purpose of TROPIC, also because the previous observation about the introduction of the Cloud infrastructure is valid in this case as well.
- **Small Cell eNB cloud enabled (SCeNBce) -** The Small Cell with enhanced cloud capabilities is a new type of node introduced for the specific simulation of the TROPIC concepts.
  The SCeNBce, also in the simulator, can be thought as an “enhancement” of the SCeNB, in the sense that the existing model has been extended with the addition of the models for the simulation of a virtualized environment. Moreover, in order to exchange messages with the SCM, a further *NetDevice* has been “installed” on the node.

  The original LENA model coexists with the new ones and continues to provide the access to the LTE resources also thanks to a specific “bridge” module, in charge of the proper management of the traffic flows (normal LTE traffic towards the ECP network and the traffic related to Cloud service, from and towards the SCM or the internal Virtual Machines).
  Below a summary of the characteristics introduced:
  - new network interface for cloud communications;
  - module for internal Radio-Cloud model interaction;
  - Hypervisor model for Virtual Machine management;
  - Scheduler model for computing resources management;
  - Virtual Machine model, with Virtual CPUs;
  - Offloading module, for the management of the requests from the UEs;
  - Application model, with parallel or serial tasks

- **Small cell Cloud Manager (SCM) -** The introduction of the computation and storage capabilities in the Small Cells and their aggregation in cluster, has required the addition of this new “entity” for the coordination of nodes, the management of the resources and the applications running in the SCeNBces. The SCM, moreover, is in charge of the administration of the whole life cycle of the Virtual Machine deployed in the SCeNBces, from their creation to their destruction and, if necessary, the migration in nodes different from the set used in the initial deployment.

  The communication among the SCM and the nodes of its cluster is based on a specific protocol, named Z-protocol. It is an application protocol that defines not only the communication between the SCM and the SCeNBces, but it also covers the communication between the SCM and the EPC and the intercommunication among the SCeNBces as well.

The nodes described above represent the set of elements necessary to create the simulation scenario, described in section 8 and used for a complete evaluation of the project novelties. Along with these latter, another “entity” has been placed in the scenario, the ***Remote (centralized) Cloud***, with the purpose of providing a term of comparison with the processing of an application on a remote node that uses the services offered by the traditional centralized cloud.

- **Remote Cloud -** The Remote is reached by the UEs through the EPC gateways, being a node on Internet. A backhaul congestion model has been introduced in this portion of the communication link in order to take into account the background traffic of this portion of the network because it would not be possible to simulate it by putting a large number of UEs in the scenario.

  The implementation of the Centralized Cloud is simplified in modelling the *server farm*, because it is out of the scope of the project, but not in modelling the processing of the UE applications. Indeed, the pool of server of the Cloud Provider is modelled as a single node, but endowed with the high computing resources that this kind of cloud has with respect to the resources available on a SCeNBce.

For the sake of completeness, we have to provide a few words about the *HeNB-GW*. This device is an optional node in the LTE-EPC architecture and has not been considered an essential element in the system evaluations. For this reason its implementation has not been planned.

3.5 Interfaces among the nodes

The *LTE-EPC model* provided by the LENA module implements most of the interfaces defined in the LTE standard, with both the Data Plane and the Control Plane, even if, it has to be said, not all of them with the same accuracy. Most of these interfaces have been modelled with high accuracy and real packets are built and exchanged among the nodes that use such interface, while for few other interfaces the information is exchanged through software mechanisms.

A brief summary of the LTE-EPC interfaces based on the information extracted from the official documentation is included below. For more details see this latter (NS3-Model).

- The **S1-U** interface is modeled in a realistic way by encapsulating data packets over GTP/UDP/IP, as in real LTE-EPC systems.
- The **S1-AP** interface provides control plane interaction between the eNB and the MME. This interface is modeled in an ideal fashion, with direct interaction between the eNB and the MME objects, without actually implementing the encoding of S1AP messages and information elements and without actually transmitting any PDU on any link.
- The **X2-C** interface is the control part of the X2 interface and it is used to send the X2-AP PDUs. In the original X2 interface the SCTP is used as the transport protocol, but such protocol is not modeled in the ns-3 simulator, therefore, the UDP protocol is used instead of the SCTP protocol.
- The **X2-U** interface is used to send the bearer data when there is DL forwarding during the execution of the X2-based handover procedure. Similarly to the S1-U interface, data packets are encapsulated over GTP/UDP/IP when being sent over this interface.

The interfaces of the *Cloud model* are those between the SCM and each SCeNBce of the cluster, the SCM and one or more nodes of the EPC, depending on the architecture option (see TROPIC-D22). Furthermore, messages for Cloud purposes are also exchanged between the UEs and their serving SCeNBces on the radio channel.

- SCM - SCeNBce interface: the new Z-protocol has been defined in the project for the communications inside the cluster and it is implemented in the simulator.

- SCM - EPC node interface: The solution proposed in [TROPIC-D52] for adding this new node in existing LTE-EPC structure doesn't require any specific interface towards the EPC nodes modeled by the LENA module.

- UE - SCeNBce interface: the communication of the UE with the "computing section" of its serving SCeNBce and with the SCM (through its Small Cell), is at the center of all the new features proposed by TROPIC. The simulator has implemented this specific interface for the exchange of the messages related to the Cloud, with a dedicated bearer activated for them.

3.6 Main simulation model

According to the terminology adopted in the documentation of *ns-3*, all the entities deployed in the simulation scenario belong to a specific *simulation model*, which provides, along with thesimulation behaviour, also the tools for their creation and configuration (the *Helper*).Following the same notation and structure, the new entities introduced in TROPIC belongto the *Cloud model* and the specific management tool, the *Cloud modelHelper*, has been created.

Figure 4 shows how each of the entities previously described (section 3.4) are distributed in the main models provided by LENA-ns3, the *LTE model* and the *EPC model*, and the new *Cloud model*.

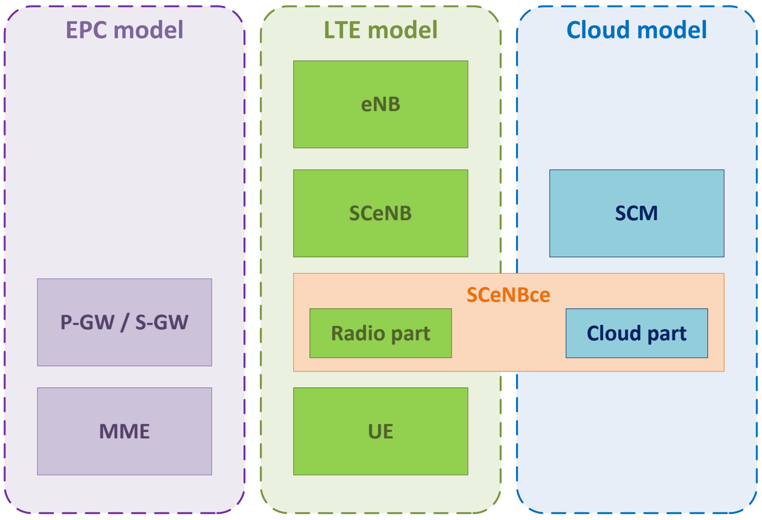


Figure 4. Simulation models

The *LTE model* includes the whole LTE Radio Protocol stack, with the simulation objects that model the stack layers (RRC, PDCP, RLC, MAC, PHY); these entities reside entirely within the UE and the eNB nodes (macro and small cells).

The *EPC model* provides means for the simulation of end-to-end IP connectivity over the LTE model. It includes the core network interfaces and the protocols that reside within the SGW-PGW and MME nodes, and only partially within the eNB nodes.

The *Cloud model* includes all the objects for building a virtualized computing environment and for managing the cluster of Small cells. The former is located exclusively inside the SCeNBce, while the functionality of cluster management resides inside the SCM node.

It should be noted that the SCeNBce is actually built through two models, the *LTE model* and *Cloud model*, because of its nature of LTE small cell enhanced with computational capabilities. Using the

Helper of the LTE model the SCeNB object is created and configured, then through the *Helper* of the *Cloud model*, the virtualized environment is installed on it to transform it in SCeNBce.

4 LTE MODEL

The simulator developed for demonstrating the features proposed by TROPIC highly takes advantage of the existing LTE model of *ns-3 framework* for the simulation of all the radio communications among UEs and serving Small Cells, for reaching the virtual computing environment and the SCM, in order to perform the offloading.

The original models have been extended with the implementation of the features and the models needed for a full demonstration of the project solutions.

The following two paragraphs describe the characteristics of the original LTE model (4.1) and the new peculiarities introduced in it (4.2). The first one is only a brief overview of the architecture of the model, needed for understanding the successive one, while for a more detailed description refer to LENA-ns3 documentation.

4.1 LENA-ns3 LTE model architecture

The radio communication involves two entities, the UE and the (SC)eNB; inside both of them the model implements the entire LTE Radio Protocol Stack as separate layers (RRC, PDCP, RLC, MAC and PHY) with the corresponding interfaces for the access to the service offered by the layer. Each layer and each inter-layer interface is modeled by a dedicated class.

The LTE model is completed by the models of the antenna and the physical channel to describe the access to the radio channel and the propagation over it.

It should be noted that, the HeNB (called SCeNBs with a wider meaning in TROPIC), having basically the same functionality of eNB, has the same structure of this latter and hence it is modeled using the same classes. This is why the architecture of the SCeNB is not explicitly described in this section.

In order to describe the architecture of the *LTE model* in a clear manner, the component objects are separated between the UE side and the eNB side, and then, for both of them, the description is split again in *Data Plane* and *Control Plane*, as it is in the LTE architecture. The following sections are dedicated to the description of their architecture, basing the explanation on the *LENA-ns3* official documentation.

4.1.1 UE side: Data Plane

The following picture shows the main classes used for the simulation of LTE stack on the Data-plane for a UE. The table provides a brief description of such classes.

Two different kinds of objects are used in the model:

- the objects that model the layers of the protocol stack;
- the objects that implement the interfaces among the stack layers.

This consideration has a general validity and can be observed in all the subsequent sections as well.

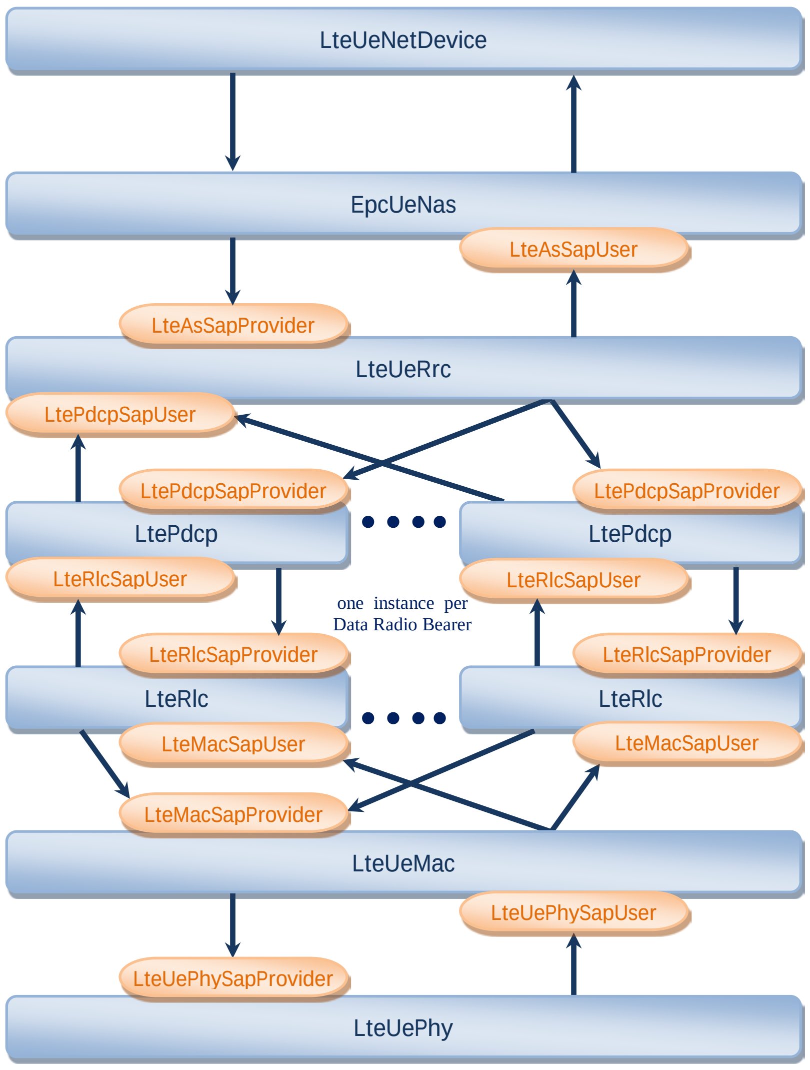


Figure 5. LTE stack architecture of UE on data-plane

Object name	**Description**
LteUeNetDevice	The LteUeNetDevice class implements the UE net device
EpcUeNas	This class manages the NAS protocol at UE side, interacting directly (for semplicity) with the MME object in the EPC.
LteUeRrc	Each of these class models the LTE layer of the same name: LteUeRrc → RRC layer LtePdcp → PDCP layer LteRlc → RLC layer LteUeMac → MAC layer LteUePhy → PHY layer
LtePdcp	
LteRlc	
LteUeMac	
LteUePhy	
LteAsSapUser	These classes implement the Access Stratum (AS) Service Access Point (SAP), i.e., the interface between the *EpcUeNas* and the *LteUeRrc*.
LteAsSapProvider	
LtePdcpSapUser	These classes represent the Service Access Point (SAP) offered by the PDCP entity to the RRC entity (as in 3GPP 36.323 Packet Data Convergence Protocol (PDCP) specifications). They are the interface between *LteUeRrc* and *LtePdcp*.
LtePdcpSapProvider	
LteRlcSapUser	These classes model the Service Access Point (SAP) offered by the UM-RLC and AM-RLC entities to the PDCP entity (see 3GPP 36.322 Radio Link Control (RLC) protocol specifications). They implement the interface between *LtePdcp* and *LteRlc*.
LteRlcSapProvider	
LteMacSapUser	This couple of classes represents the Service Access Point (SAP) offered by the MAC to the RLC, according to the Femto Forum MAC Scheduler Interface Specification v1.11, Figure 1. They are the interface between *LteRlc* and *LteUeMac*.
LteMacSapProvider	
LtePhySapUser	These classes implement the Service Access Point (SAP) offered by the PHY to the MAC, i.e. the interface between *LteUeMac* and *LteUePhy*.
LtePhySapProvider	

Table 2. Description of the UE data-plane classes

4.1.2 UE side: Control Plane

Many classes of the LTE model are developed to manage both the data-plane and the control-plane in order to avoid code duplication due to the common functionalities. This is the reason why some classes of the data-plane can be found in the following control-plane diagram as well. The description table contains only the new classes, specific of the control-plane.

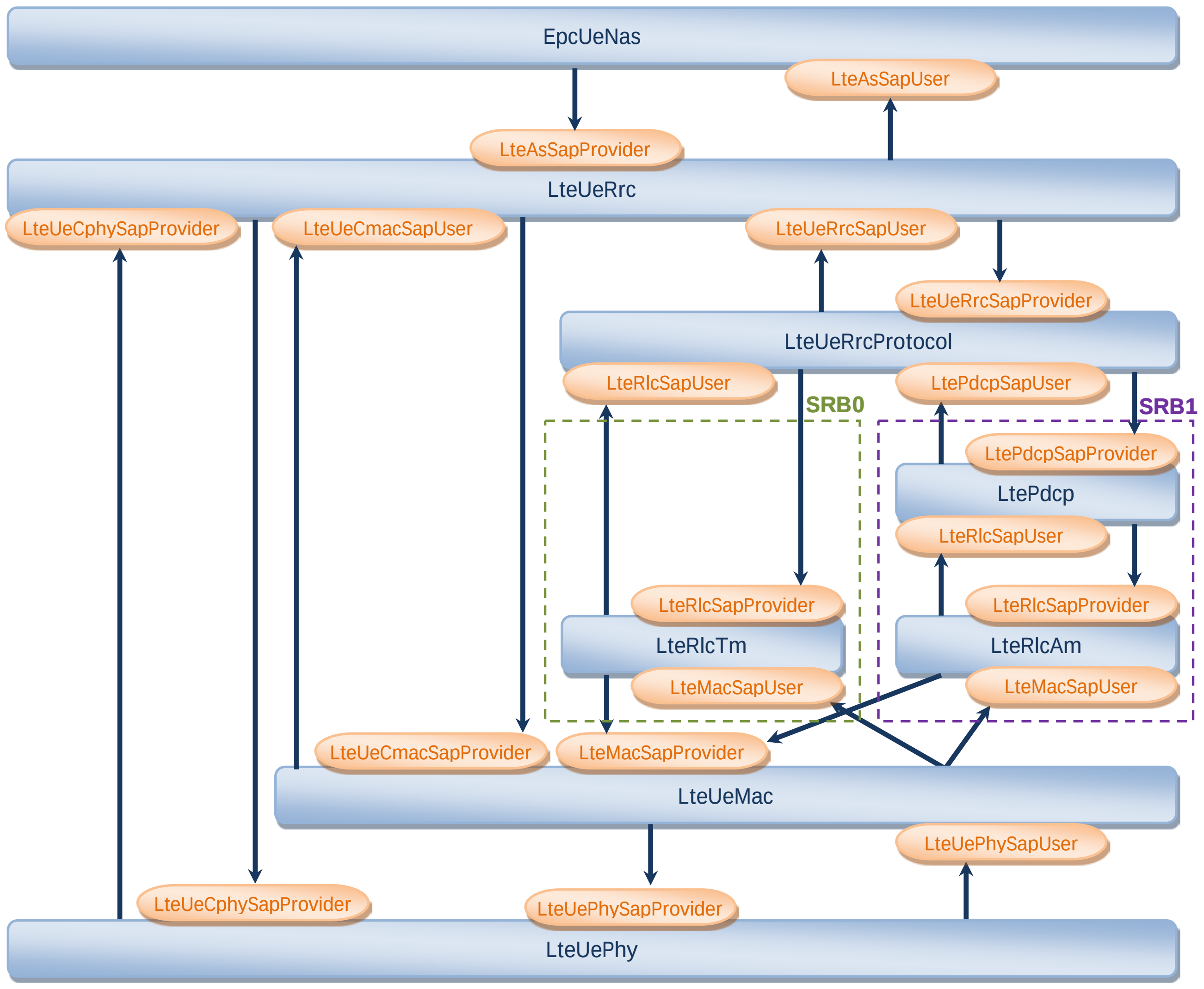


Figure 6. LTE stack architecture of UE on control-plane

Object name	Description
LteUeRrcProtocol	This class models the transmission of RRC messages from the UE to the eNB. This can be implemented in an ideal fashion, without errors and without consuming any radio resources, or in a real fashion, by creating real RRC PDUs and transmitting them using radio resources allocated by the LTE MAC scheduler.
LteUeRrcSapUser	These two classes represent the Service Access Point (SAP) used by the UE RRC to send and receive messages to/from the eNB (each method defined in this class corresponds to a message that is defined in section 6.2.2 of TS 36.331).
LteUeRrcSapProvider	
LteRlcTm	These three classes implement the LTE RLC modes specified in 3GPP TS 36.322, the Transparent Mode (TM), the Acknowledge Mode (AM) and the Unacknowledge Mode (UM).
LteRlcAm	
LteRlcUm	
LteCmacSapUser	These are the classes used to implement the Service Access Point (SAP) offered by the UE MAC to the UE RRC for control purposes.
LteCmacSapProvider	
LteCphySapUser	These classes models the Service Access Point (SAP) offered by the UE PHY to the UE RRC for control purposes.
LteCphySapProvider	

Table 3. Description of the UE control-plane classes

Regarding the RLC layer, the model implements all the three operation modes described in the 3GPP specification, the Transparent Mode (TM), the Acknowledge Mode (AM) and the Unacknowledge Mode (UM). The Figure 6, for simplicity, shows only the classes for the AM mode and the TM mode, since the operations of the UM RLC are similar to those of the AM RLC, with the difference that retransmission are not performed.

4.1.3 eNB side: Data Plane

The architecture of the eNB is very similar to that of the UE: in both cases there is a single MAC instance and a single RRC instance, that work together with pairs of RLC and PDCP instances (one RLC and one PDCP instance per radio bearer). This is why some of the layers of the protocol stack can be modeled using the same classes of the UE.

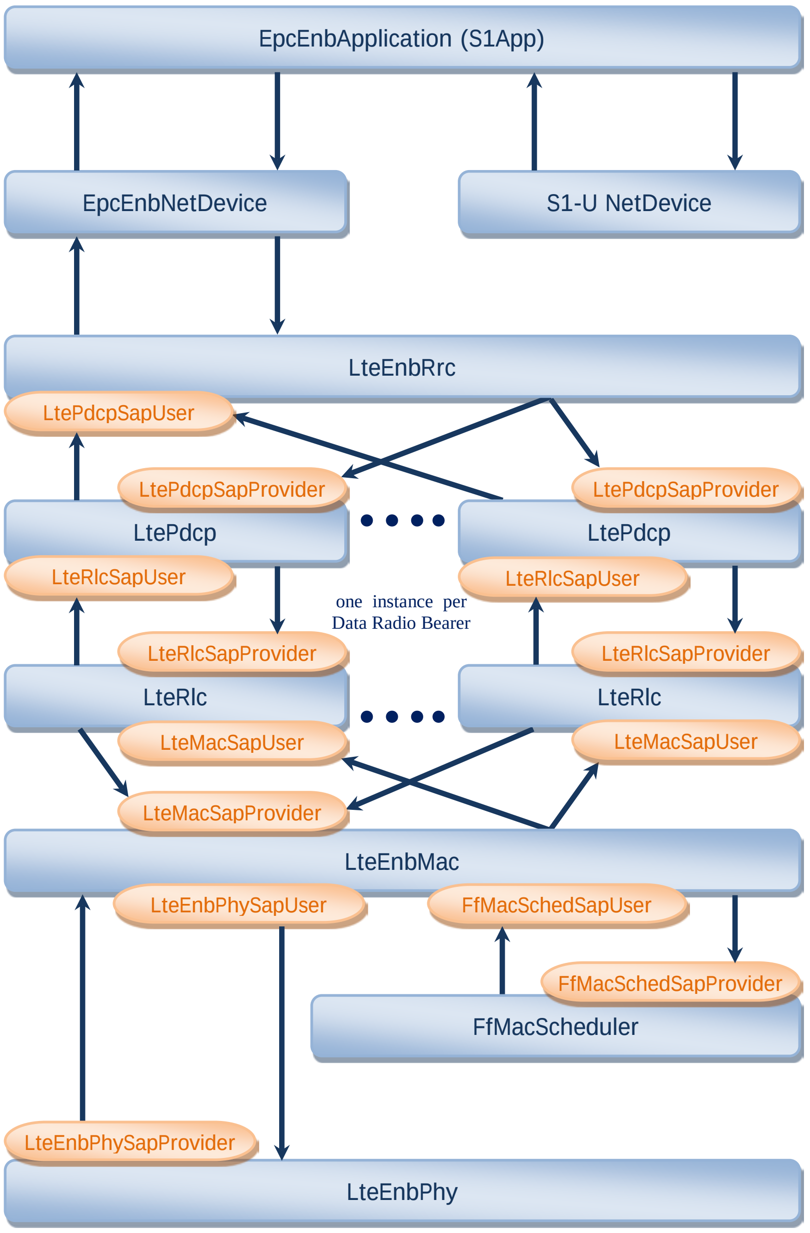


Figure 7. LTE stack architecture of eNB on data-plane

Object name	Description
LteEnbNetDevice	These two classes implement the couple of devices of the eNodeB towards the radio channel and the EPC.
S1-U NetDevice	
EpcEnbApplication	This class implements the application, installed inside eNBs, that provides the bridge functionality for user data plane packets between the radio interface and the S1-U interface.
LteEnbRrc	Each of these classes implements the corresponding layer of the stack. The layers PDCP and RLC are modeled with the same classes used at UE side, while the other three layers require specific classes. LteEnbRrc → RRC layer LtePdcp → PDCP layer (same of UE) LteRlc → RLC layer (same of UE) LteEnbMac → MAC layer LteEnbPhy → PHY layer
LtePdcp	
LteRlc	
LteEnbMac	
LteEnbPhy	
LtePdcpSapUser	The classes used as Service Access Points (SAP) of the PDCP, RLC and MAC layers are the same of the UE side model.
LtePdcpSapProvider	
LteRlcSapUser	
LteRlcSapProvider	
LteMacSapUser	
LteMacSapProvider	
LteEnbPhySapUser	These two classes model the Service Access Point (SAP) offered by the eNB-PHY to the eNB-MAC, one is from MAC to PHY, the second for the opposite direction.
LteEnbPhySapProvider	
FfMacScheduler	This class identifies the interface by means of which one the possible implementations of the scheduler is plugged on the MAC layer.
FfMacSchedSapUser	This couple of classes defines the MAC Scheduler interface specified in the Femto Forum Technical Document (LTE MAC Scheduler Interface Specification v1.11)
FfMacSchedSapProvider	

Table 4. Description of the eNB data-plane classes

4.1.4 eNB side: Control Plane

For the control-plane of the ENB, similar considerations apply to those made the previous sections about the common classes, but with more limitations due to the complexity of this architecture.

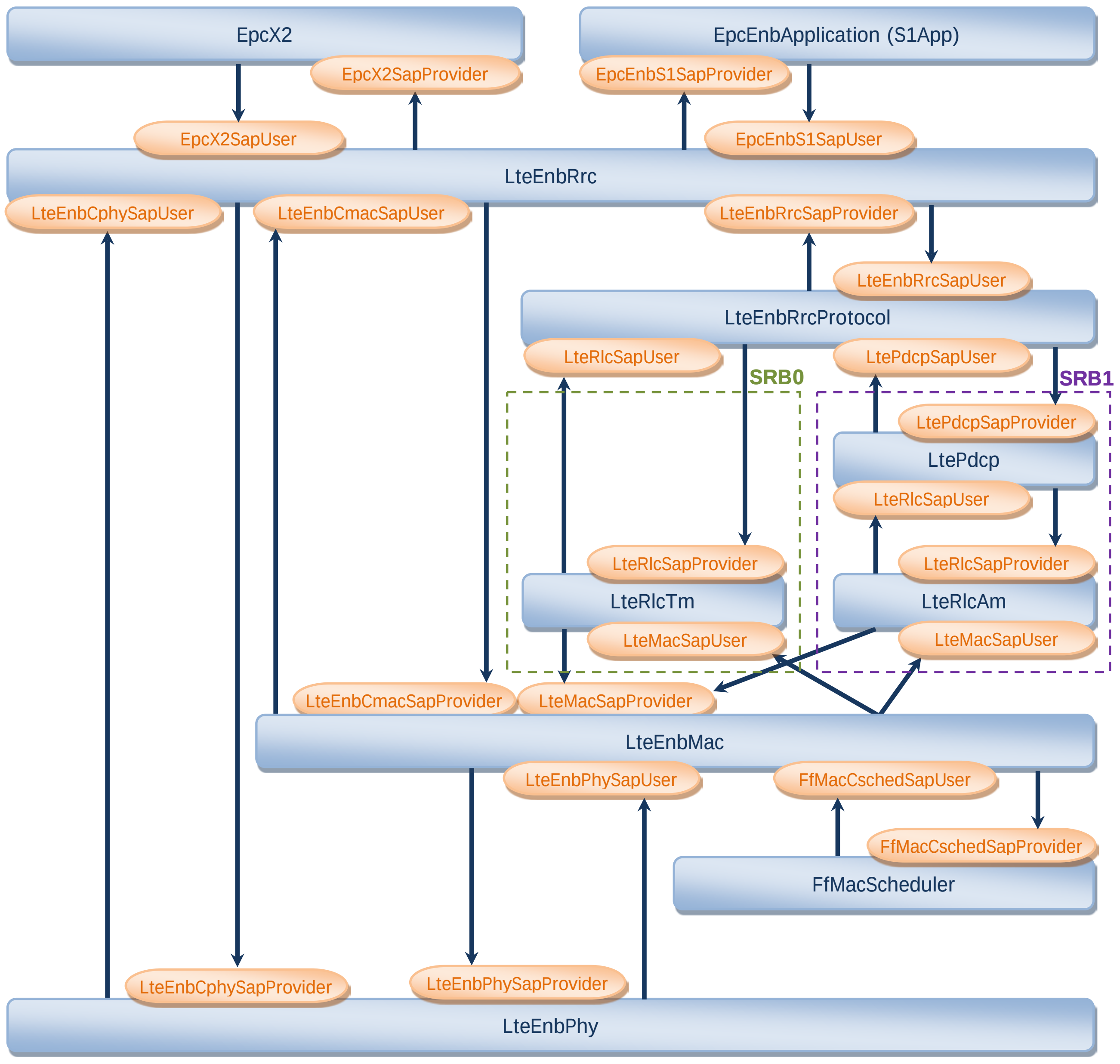


Figure 8. LTE stack architecture of eNB on control-plane

Object name	Description
EpcX2	This entity is installed inside an eNB and provides the functionality for the X2 interface
LteEnbRrcProtocol	This class models the transmission of RRC messages from the UE to the eNB. Two models are available: an ideal model, without errors and without consuming any radio resources, and a real model, by creating real RRC PDUs and transmitting them over Signaling Radio Bearers using radio resources allocated by the LTE MAC scheduler.
LteRlcTm	The classes for the RLC modes are the same of the control-plan for the UE side.
LteRlcAm	
LteRlcUm	
EpcX2SapUser	These two classes implement the Service Access Point (SAP) between the X2 entity and the RRC entity. The X2 SAP follows the specification 3GPP TS 36.423, "X2 application protocol (X2AP)". The EpcX2SapUser is the service primitive provided by RRC and EpcX2SapProvider is the interface provided by the X2 entity.
EpcX2SapProvider	
EpcEnbS1SapUser	These classes implement the Service Access Point (SAP) between the LteEnbRrc and the EpcEnbApplication.
EpcEnbS1SapProvider	
LteEnbRrcSapUser	The classes model the Service Access Point (SAP) used by the eNB RRC and UE RRC to exchange messages. The messages implementation follows the 3GPP specification TS 36.331.
LteEnbRrcSapProvider	
LteEnbCmacSapUser	These classes implement the Service Access Point (SAP) between the eNB MAC and the eNB RRC (see Femto Forum MAC Scheduler Interface Specification v 1.11)
LteEnbCmacSapProvider	
LteEnbCphySapUser	These two classes model the Service Access Point (SAP) offered by the eNB PHY to the eNB RRC and vice versa for control purposes.
LteEnbCphySapProvider	
FfMacCschedSapUser	This couple of classes implements the MAC Scheduler interface specified in the Femto Forum Technical Document (LTE MAC Scheduler Interface Specification v1.11), in particular provides the CSCHED SAP.
FfMacCschedSapProvider	

Table 5. Description of the eNB control-plane classes

4.1.5 PHY and channel model architecture

The following figure shows the architecture used for the implementation of the PHY layer and channel model. Both the UE and the eNB architecture are displayed even if the two architectures are basically the same. The unique difference resides in the fact that the eNB manages the downlink and uplink flows of multiple instances, one for each of the UEs attached to it.

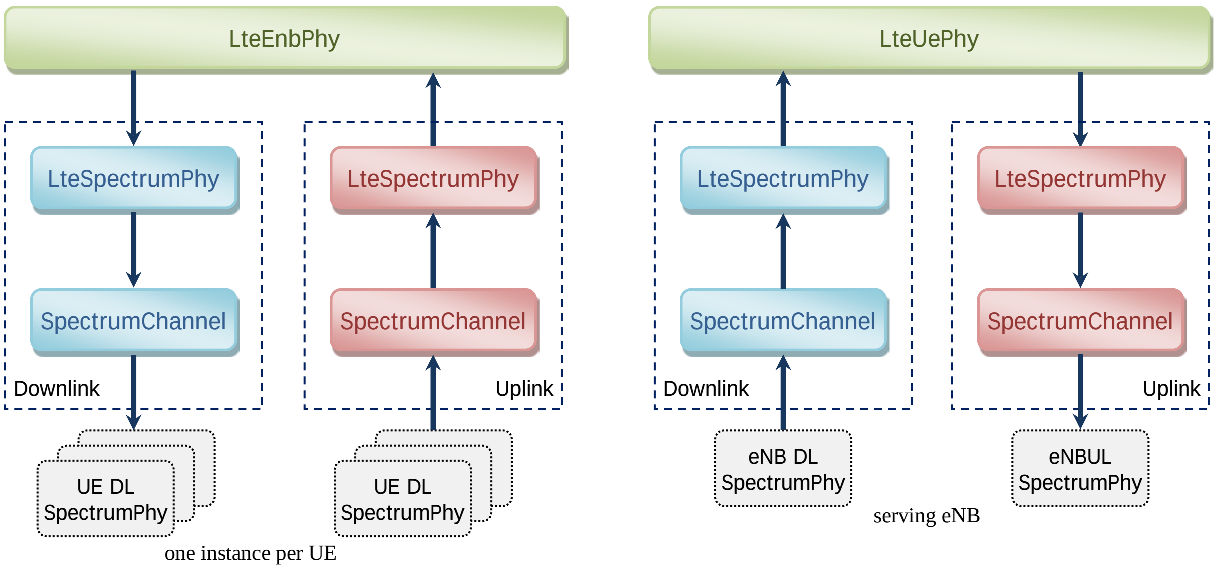


Figure 9. PHY and channel architecture

Object name	Description
LteEnbPhy LteUePhy	This two classes model the PHY layer of the stack. They are the same already described in the previous sections for the UE and eNB models.
LteSpectrumPhy	The class models the physical layer of LTE. It supports a single antenna model instance which is used for both transmission and reception. “Single antenna” is the default model, but it possible to change the configuration, selecting among the following: • mode 1: SISO; • mode 2: MIMO Tx Diversity, • mode 3: MIMO Spatial Multiplexity Open Loop, • mode 4: MIMO Spatial Multiplexity Closed Loop, • mode 5: MIMO Multi-User, • mode 6: Closer loop single layer precoding, • mode 7: Single antenna port 5.
SpectrumChannel	This class defines the interface for spectrum-aware channel implementations

Table 6. Description of the PHY and channel classes

4.2 *Extended LTE model*

This section illustrates how the new features introduced by TROPIC are implemented inside the original model of the LTE structure. The changes are spread over multiple objects, both classes of the specific LENA module and generic classes of the *ns-3 framework*, so many references to these objects will be inevitably mentioned. For details, the refer to the official documentation.

4.2.1 SCeNBce node architecture

The SCeNBce is the new entity introduced by TROPIC with the purpose of adding computational capabilities to the traditional LTE HeNB. This suggests that a SCeNBce can be modeled as an extension of the existing model of the HeNB, in the same manner of what it is its real structure, that is, in an extremely simplified view, a HeNB with an additional network interface to communicate with the SCM, along with the software layer for the clouding implementation.

In the specific case of the *ns-3* framework, following the general structure used for every node and described in section 3.3, the SCeNBce entity can be modeled as in Figure 10. It is only a very synthetic representation for showing how is changed the structure of the new node with respect to the node already available in *LENA-ns3*.

In detail, compared with a HeNB node, the model of the SCeNBce requires:

- an additional *NetDevice* used for the cloud related communication;
- the corresponding Protocol Stack for the new device;
- a model for the implementation of the cloud functionalities (in the figure simply summarized with *Cloud Lib*);
- an application (*Cloud App*), implemented in the “radio-side” with the task of managing the interface with the “cloud-side”. Moreover, this application provides to the “cloud-side” the access to the radio link towards the UEs.

The object labeled *Cloud Lib* is, actually, a complex item that encloses many different models:

- a model for the physical resources of the node,
- a model for the virtualized platform and its “entities” (Virtual Machines and Virtual CPUs),
- the management of the shared virtual resources.

The characteristics listed above are implemented by the ***Cloud model*** whose detailed description is done in section 6.

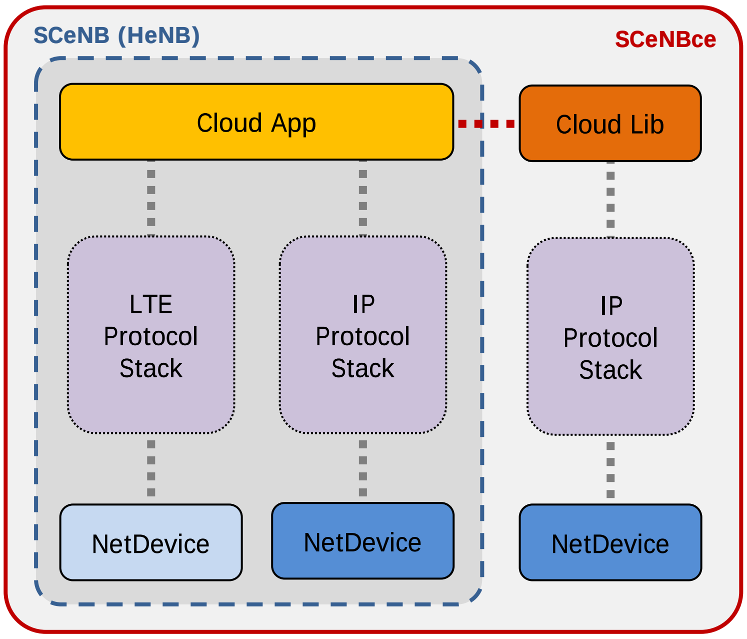


Figure 10. SCeNBce node architecture

4.2.2 UE Energy consumption model

The UE energy consumption depends on the UE state from the processing point-of-view (IDLE, RUN) and on the communication state (RX, IDLE, TX, RX_CRTL, ..).

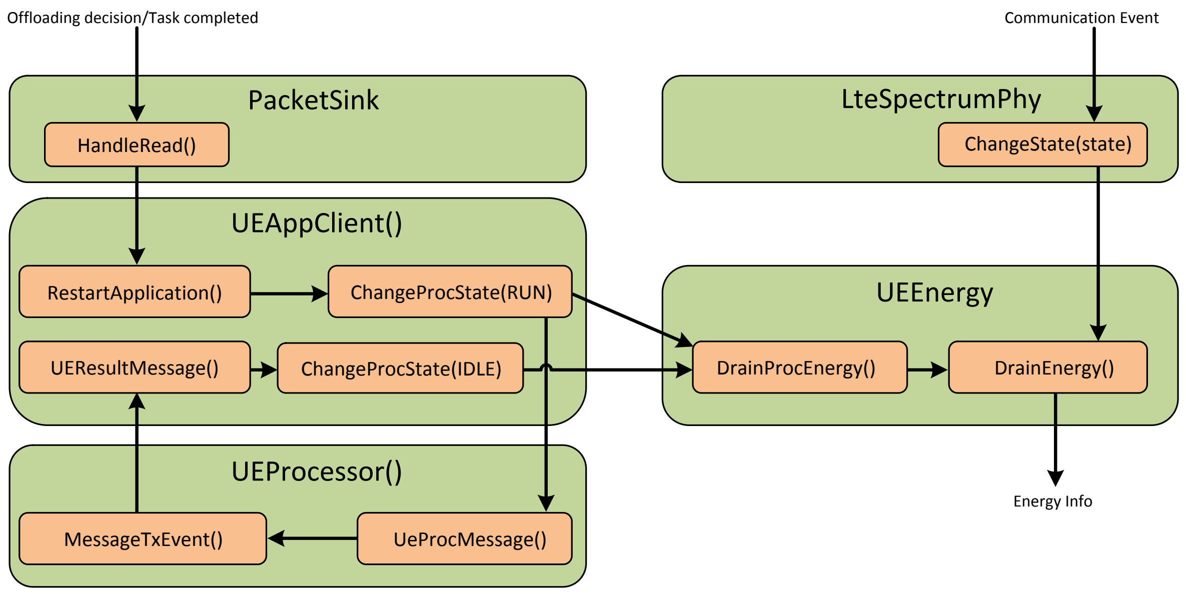


Figure 11. Energy consumption model at UE

When a task processing inside the UE is enabled, the *UEProcessor::ChangeProcState(RUN)* call enables the *UEEnergy::DrainProcEnergy* to set the energy to be drained in RUN state.

When the UE processing ends, the *UEProcessor::ChangeProcState(IDLE)* enables the *UEEnergy::DrainProcEnergy* to set the energy to be drained in IDLE state. The *UEEnergy::DrainEnergy* drains energy according to the processing state when a communication event occurs (see below).

The *LteSpectrumPhy::ChangeState(state)* is called when a communication state transition occurs. The calls enables *UEEnergy::DrainEnergy()* to drain the energy due to the communications according to the state transition and to the time duration of the previous state.

The processing inside the UE is simply modelled by the *UEProcessor* module which is activated through the *UEProcessor::UeProcMessage()* call. It de-activates itself when the foreseen processing time is elapsed and the *UEProcessor::MessageTxEvent()* call is used to warn the *UEAppClient* that the UE processor state has to be changed (RUN→IDLE).

Class::Method	**Description**
PacketSink::HandleRead(Ptr <Socket> socket)	The method allows the UE to receive messages from the attached SCeNBce. According to the packet Header and its content (offloading decision, i-th task completed) the method calls the *UEAppClient::RestartApplication()* to start the processing of the next task or to start again the application if all tasks have been processed.
UEAppClient::RestartApplication(int MsgId)	The method, according to the message Id, allows to start the processing of the next task or to start again the application if all tasks have been processed. It also changes the UE processing status from IDLE to RUN through the method *UEAppClient::ChangeProcState*.

Class::Method	Description
UEAppClient::ChangeProcState(int newState, double dE, double dTMs)	When the transition IDLE to RUN occurs, the method enables the *UEEnergy* module to drain the processing energy from the UE in RUN mode. The dE energy provided is the total energy consumption inside during the whole processing time foreseen in dTMs. When the transition RUN to IDLE occurs the amount of energy to be drained in IDLE mode every 1 ms is provided and the time interval is fixed to 1ms. In both cases the *UEEnergy::DrainEnergy()* module will calculate the consumption every 1ms (about) in order to drain the energy at their rate (about 1 ms).
UEAppClient::UEResultMessage(Ptr <Packet> p, MsgUEProc msg)	The method receives the UE processing results, enables the transition RUN to IDLE of the UE processor and enables the *UEEnergy* module to drain the energy needed in IDLE state.
UEEnergy::DrainProcEnergy(int state, double totalEnergyProc, double totalProcTimeMs)	The method calculates the energy consumption due to the processing in the current state every 1 ms (due to the processing in the current state) and provides the value to the method *UEEnergy::DrainEnergy()*.
UEEnergy::DrainEnergy(int state, int mcs, double stx)	The method is enabled by the PHY layer when a communication state transition occurs. It drains the energy according to the elapsed time between two communication state transition using the previous communication state and related parameters (MCS, Stx). It also drains the energy spent by UE processor in the same time interval according to the UE processor state.
LteSpectrumPhy::ChangeState(State newState)	The method enables the *UEEnergy* module when a communication state transition occurs also providing the new state value.
UEProcessor::UEProcMessage(Ptr <Packet> p, MsgUEProc msg)	The method receives a processing request when a task or a portion of it has to be processed. It simulates the processing performing a delay equal to the foreseen processing time which depends on the UE model and characteristics.
UEProcessor::MessageTXEvent(Ptr <Packet> p, MsgUEProc msg)	The method is enabled when the UE processing ends and enables the *UEAppClient* application to change the UE processor state from RUN to IDLE.

Table 7. UE Energy consumption model: classes and methods

4.2.3 Offloading and Processing model

The offloading and the successive processing are described together in this section in order to give a global overview of the implementation, even if entities of both the *LTE model* and of the *Cloud model* are involved.

The offloading of an application from a user device towards the Small Cell Cloud can start only after the reception of the response message from the SCM, containing the optimal configuration of the parameters used by the adopted algorithm.

The SCM can use one between the *Energy-Latency Tradeoff* algoritm and the *Joint Radio-Cloud Optimization* algorithm. They consider a different set of parameters to be optimized and therefore a slight difference there is also in the initial sequence of actions, while after this phase, the processing of the application, split in tasks, proceeds in the same manner. The main differences between the algorithms are listed below.

- With the *Energy-Latency Tradeoff* algoritm the SCM requires the estimations of the time and the energy about the local processing from the UE, based on its channel condition. The second one doesn't need specific information from the UE. Actually, as explained in section 7.1.3, the information about the channel are collected by the *External Library manager*, in a simplified manner, without message exchange.
- The *Energy-Latency Tradeoff* algoritm needs also the estimations of the time and the energy about the processing performed on the SCC. For this reason, as depicted in Figure 12, the request message from the UE is dispatched to the Hypervisor, before reaching the SCM.
- With the *Joint Radio-Cloud Optimization* algorithm, instead, the SCM needs from the Hypervisor the information about the status of the application running on its Virtual Machines for all the UEs. This kind of data is automatically updated by the Hypervisor by means of the status messages, sent to the SCM when the conditions change for a UE. At the moment of a new UE request, nothing have to be sent by the Hypervisor.
- For both the algorithms, the optimal transmission power to be used by the UE is computed and sent inside the answer message.
- The *Joint Radio-Cloud Optimization* algorithm computes also the optimal allocation of the computing resources to assigne to every UEs by the Hypervisor. This is why the SCM sends, along with the answer message to the UE, a message to the Hypervisor, as well. It isn't sent with the *Energy-Latency Tradeoff* algoritm.
- The *Energy-Latency Tradeoff* algoritm is based on data-partition approach of the application, therefore it provides also the optimal partition between local and remote processing. The information is contained in the answer message.
- For the *Joint Radio-Cloud Optimization* algorithm the information about the application is an input; it needs to know, for every UEs, the number of bit to be still transmitted and the number of instruction to be still processed. The Hypervisor, again, updates this information on the SCM by means of the status message.

The following Figure 12 and Figure 13 show the the offloading request and the decision for the *Energy-Latency Tradeoff* algoritm. Further details about the *Joint Radio-Cloud Optimization* algorithm, being implemented through an extern library, are found in section 7.1.3. Moreover, the processing schema depicted in Figure 12 is valid for this latter as well.

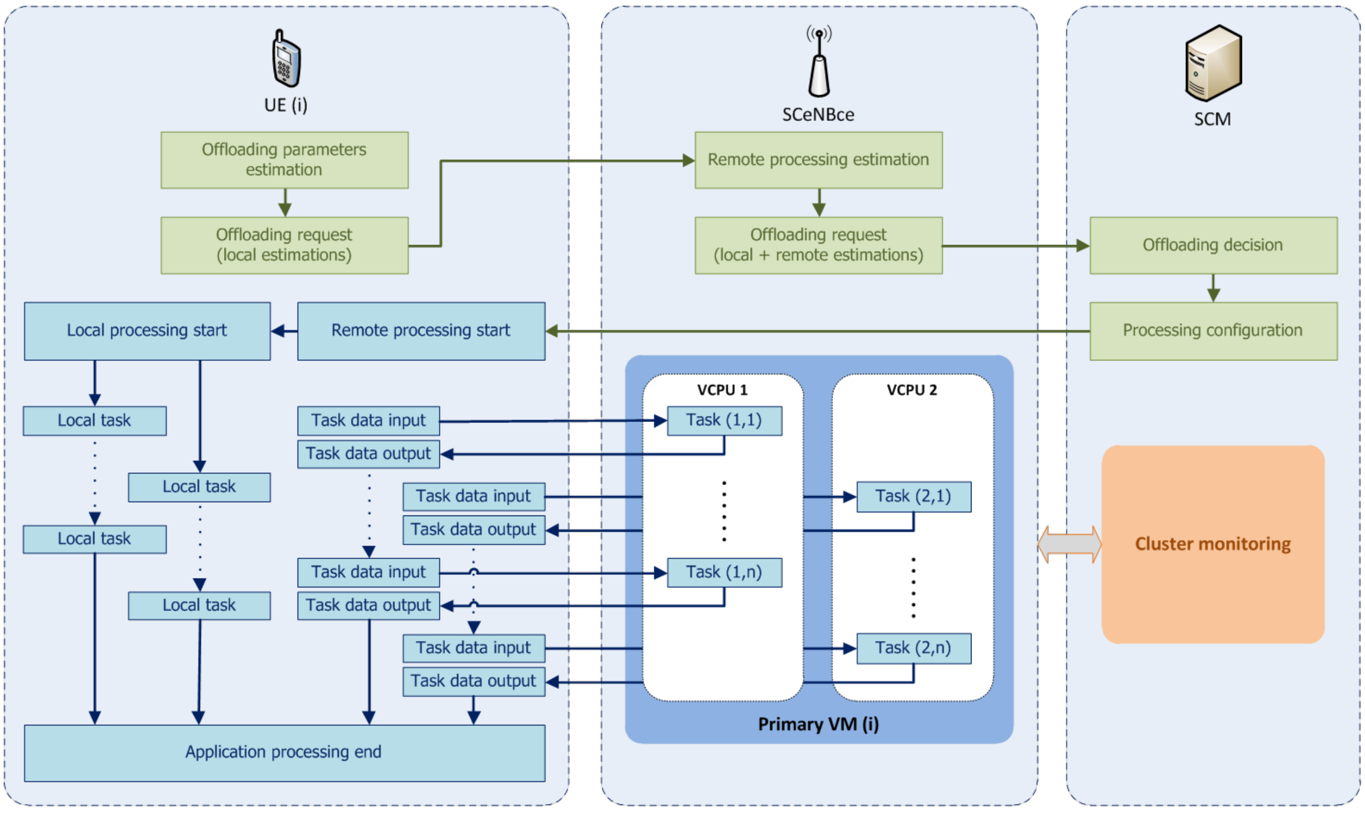


Figure 12. Offloading and processing schema with Energy-Latency tradeoff algorithm

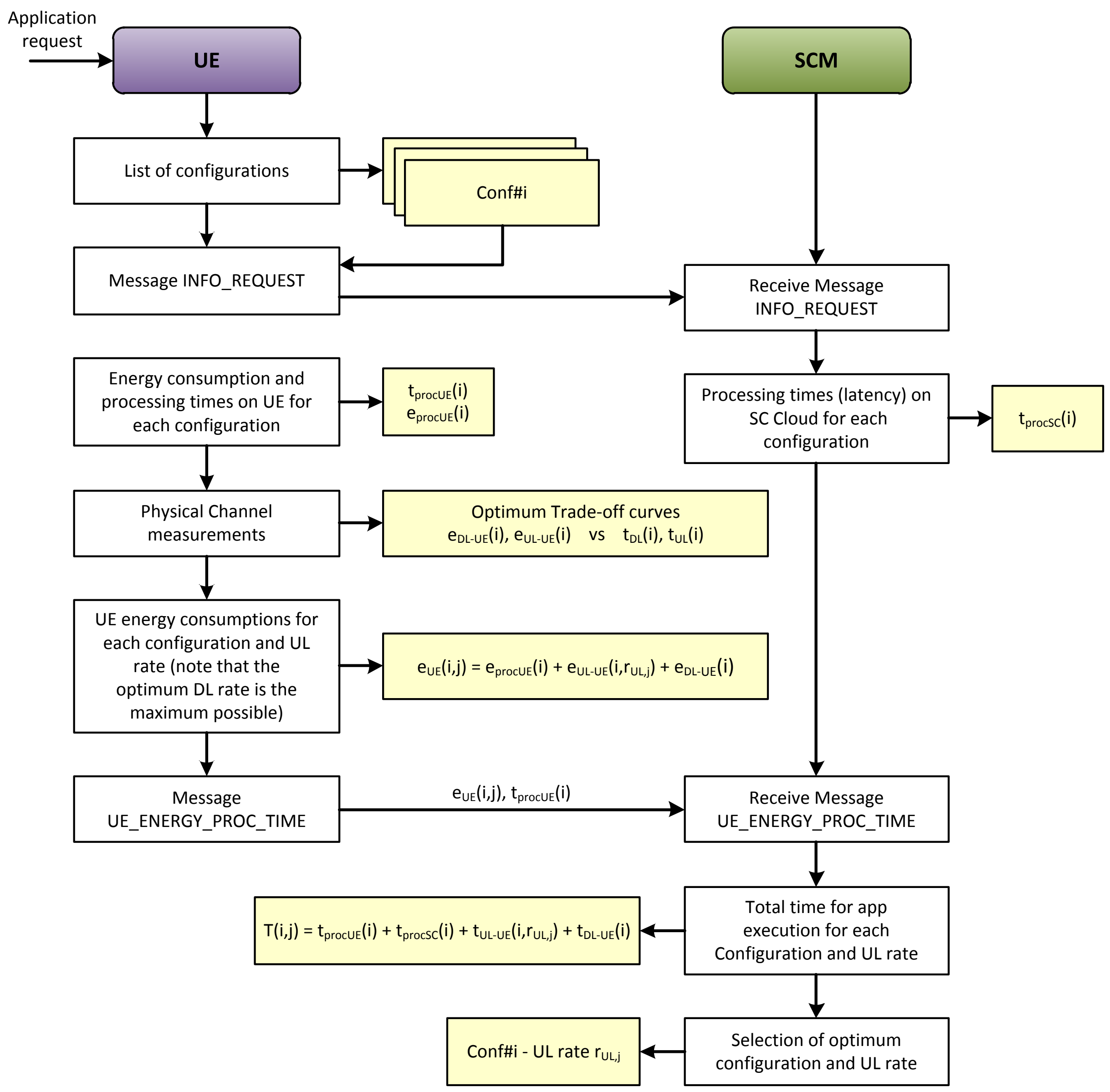

Application request
UE
SCM
List of configurations
Conf#i
Message INFO_REQUEST
Receive Message INFO_REQUEST
Energy consumption and processing times on UE for each configuration
$t_{procUE}(i)$
$e_{procUE}(i)$
Processing times (latency) on SC Cloud for each configuration
$t_{procSC}(i)$
Physical Channel measurements
Optimum Trade-off curves
$e_{DL\text{-}UE}(i), e_{UL\text{-}UE}(i)$ vs $t_{DL}(i), t_{UL}(i)$
UE energy consumptions for each configuration and UL rate (note that the optimum DL rate is the maximum possible)
$e_{UE}(i,j) = e_{procUE}(i) + e_{UL\text{-}UE}(i,r_{UL,j}) + e_{DL\text{-}UE}(i)$
Message UE_ENERGY_PROC_TIME
$e_{UE}(i,j), t_{procUE}(i)$
Receive Message UE_ENERGY_PROC_TIME
Total time for app execution for each Configuration and UL rate
$T(i,j) = t_{procUE}(i) + t_{procSC}(i) + t_{UL\text{-}UE}(i,r_{UL,j}) + t_{DL\text{-}UE}(i)$
Selection of optimum configuration and UL rate
Conf#i - UL rate $r_{UL,j}$

Figure 13. Offloading decision schema with Energy-Latency tradeoff algorithm

4.2.3.1 Offloading and Processing at UE node

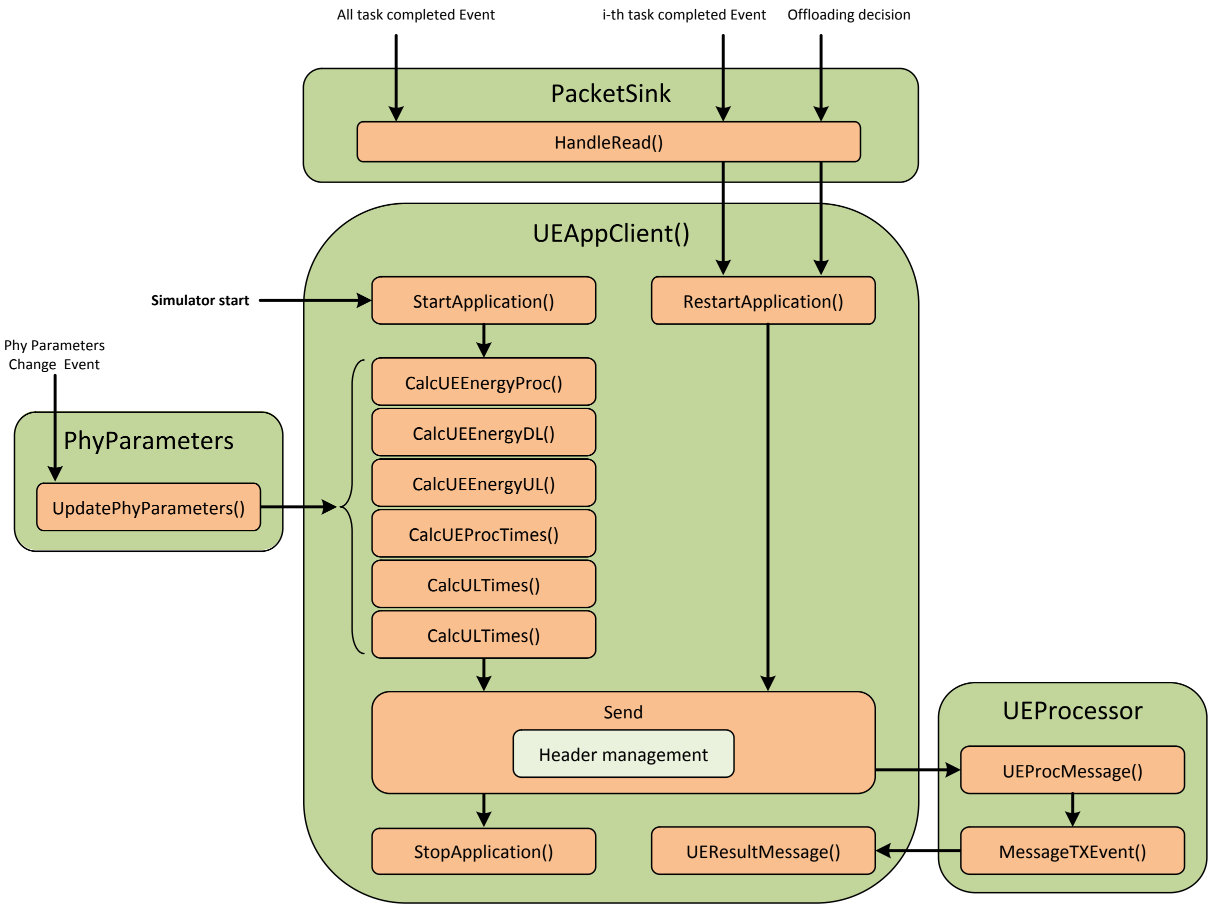


Figure 14. Offloading and processing model at UE

During the initial setup of the simulator, the UE defines the type of application to be executed and the possible configurations that define the percentage to be processed locally at the UE or remotely on the SCC with an offloading request.

The sequence of operations performed by a UE when it starts a new application is the same with both the offloading algorithms implemented, but are different the data inside the initial request message.

When the ***Energy-Latency Tradeoff*** algorithm is selected, the UE has to send to the SCM the estimations of time and energy. According to the application details (number of instructions, I/O,…), the UE characteristics (processor type and speed), the available UL and DL communications speeds and some physical layer parameters, the UE calculates the energy required for each configuration, the communication times and the UE processing time. These parameters are then sent to the SCeNBce via a dedicated bearer in order to enable the SCM to select the best configuration.

The *PhyParameters* module updates the needed physical parameters and makes them available to the UE when they changes.

When the SCM selects the optimal configuration, the UE receives the offloading decision message and calls the *UEAppClient::RestartApplication()* which enables the processing of the first task according to the selected configuration. The percentage of the task to be processed on the UE (if any) defines the time duration of the RUN state of the UE processor. The percentage of the task to be processed on the SCeNBce defines the processing details sent via a dedicated bearer to the SCeNBce.

If the current task is defined as 'not parallel', the UE waits for the task results before the next task sending, otherwise the next task is immediately sent to the SCeNBce. When the processing of the i-th task has been done, the UE receives a message from the SCeNBce. When all tasks have been completed the current application ends and a new application is started.

When the ***Joint Radio-Cloud Optimization*** algorithm is used, instead, all the information needed for the decision on the SCM are provided to it by the Hypervisor on the SCeNBce by means of the status messagge. In this case the answer message from the SCM contains only the information about the transmission power to be used. The rest of the actions until the end of the processing are the same, except for the fact that also the messages with the data from each task processed on the SCC contain the information about the transmission power. The reason is related to the continuous optimization of the parameters performed by the SCM every time a new offloading request is received, so the UEs update the transmission power even if the offloading is alredy running. The same is done by the Hypervisor with the computing resources assigned to the Virtual Machines.

Class::Method	Description
PacketSink:: HandleRead (Ptr <Socket> socket)	The method allows the UE to receive messages from the attached SCeNBce. According to the packet header and its content (offloading decision, i-th task completed) the method calls the *UEAppClient::RestartApplication()* to start the processing of the next task or to start again the application if all tasks have been processed.
PhyParameters:: UpdatePhyParameters (void)	The method provides and updates the PHY layer parameters used by the *UEAppClient* application.
UEAppClient:: StartApplication (void)	The method initializes the communication socket and enables the method *UEAppClient::Send()* in order to send the first message to the attached SCeNBce.
CalcUeEnergyProc (UeAppParameters par)	The method calculates the energy consumption due to the processing inside the UE for each foreseen configuration.
CalcUeEnergyUL (UeAppParameters par)	The method calculates the energy consumption due to the UL communications for each foreseen configuration.
CalcUeEnergyDL (UeAppParameters par)	The method calculates the energy consumption due to the DL communications for each foreseen configuration.
CalcProcTimes (UeAppParameters par)	The method calculates the time needed to perform the processing inside the UE for each foreseen configuration.
CalcULTimes (UeAppParameters par)	The method calculates the time needed for UL communications for each foreseen configuration and for each UL foreseen speed.
CalcDLTimes (	The method calculates the time needed for DL communications for each foreseen configuration and

Class::Method	Description
UeAppParameters par)	for each DL foreseen speed.
UEAppClient:: Send (void)	The method detects the application state and, according to it: - *UEAPP_IDLE*: starts application - *UEAPP_INFOTX*: sends offloading request to the SCM after energy/times calculations for each foreseen configurations - *UEAPP_INFORX*: receives offloading decision from the SCM, performs the related actions and enables the transition to *UEAPP_DATATX* - *UEAPP_DATATX*: sends the i-th task (or a portion of it) and related data to the SCeNBce in order to process it and enables the UE processor if a portion of task has to be processed inside the UE - *UEAPP_DATARX*: receives the i-th task processing results from SCeNBce, sends the next task or restarts the application if all tasks have been completed.
UEAppClient:: StopApplication (void)	The method disables the application which is reactivated by the method *UEAppClient::RestartApplication()*.
UEAppClient::RestartApplication (int MsgId)	The method, according to the message Id, allows to start the processing of the next task or to start again the application if all tasks have been processed. It also changes the UE processing status from IDLE to RUN through the method *UEAppClient::ChangeProcState*.
UEAppClient:: UEResultMessage (Ptr <Packet> p, MsgUEProc msg)	The method receives the UE processing results, enables the transition RUN to IDLE of the UE processor and enables the *UEEnergy* module to drain the energy needed in IDLE state.
UEProcessor:: UEProcMessage (Ptr <Packet> p, MsgUEProc msg)	The method receives a processing request when a task or a portion of it has to be processed. It simulates the processing performing a delay equal to the foreseen processing time which depends on the UE model and characteristics.
UEProcessor:: MessageTXEvent (Ptr <Packet> p, MsgUEProc msg)	The method is enabled when the UE processing ends and enables the *UEAppClient* application to change the UE processor state from RUN to IDLE.

Table 8. UE Offloading and processing model: classes and methods

4.2.3.2 *Offloading and Processing at SCeNBce node*

The SCeNBce responds to messages from the UEs and from the SCM. The UEs send messages for starting the application processing, with an offloading request, or messages with the request for processing the i-th task of the running application. As for the UE node, the management of the offloading request messages depends on the type of algorithm used bt the SCM, while there are no differences for the processing requests.

When the ***Energy-Latency Tradeoff*** algorithm is selected (Figure 15), the offloading request is dispatched to the Hypervisor through the *ServerCallback* module in order to obtain the estimation of the processing time for the possible configurations of the application. The processing times are evaluated by the Hypervisor on the basis of the current status of the Virtual Machines.
The estimation, along with the information sent by the UE, are then sent to the SCM for the selection of the optimal configuration.

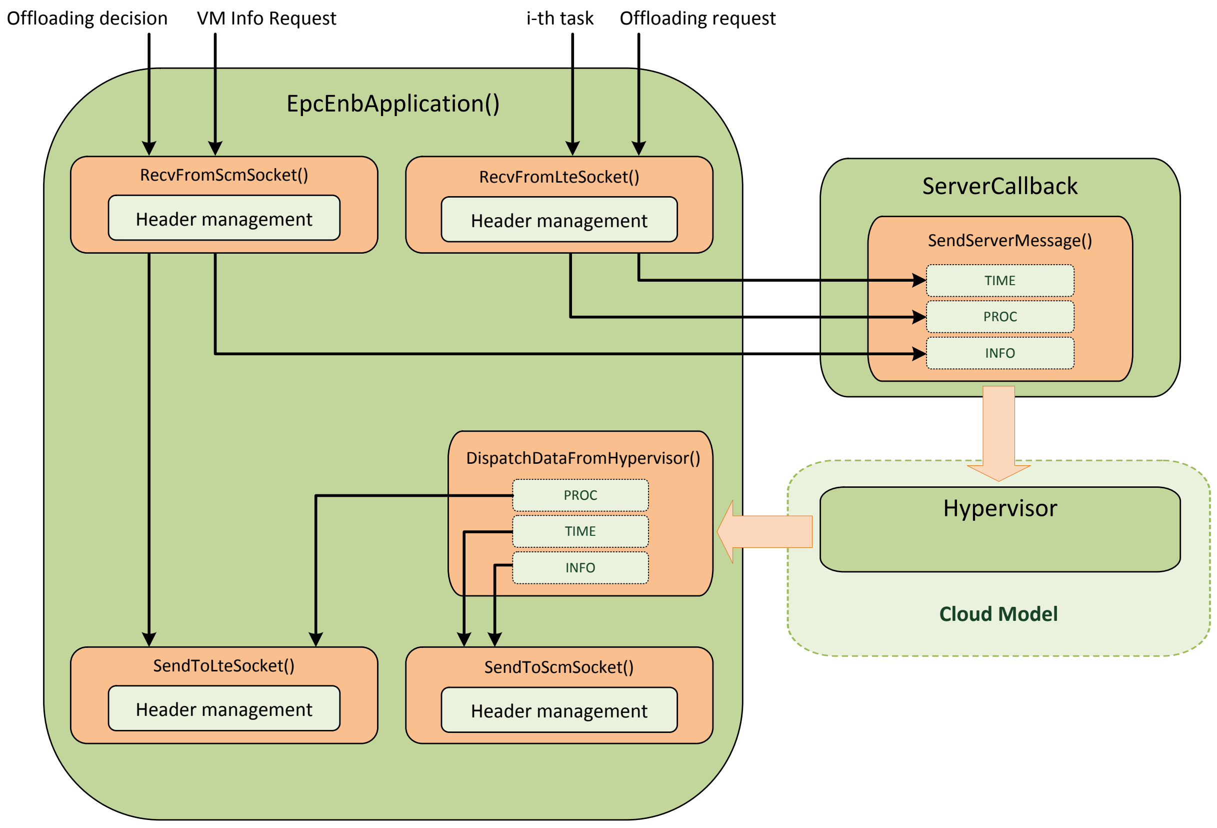


Figure 15. Offloading and processing model at SCeNBce (Energy-Latency algorithm)

When the ***Joint Radio-Cloud Optimization*** algorithm is adopted, the offloading request message is dispatched directly to the SCM, because the estimations from the Hypervisor are not needed and the information for the decision are always available and updated on the SCM thanks to the monitoring messages. At the reception of the answer message from the SCM, the SCeNBce sends two messages:

1. a message to the UE as response to its offloading request;
2. a message to the Hypervisor with the optimized parameters for the corresponding resources allocation.

After the conclusion of the offloading request phase, the UEs start to send processing request messages for the tasks of the application. These messages are dispatched to the Hypervisor which sends back a message to the SCeNBce, as soon as the processing of a task has been completed. The SCeNBce simply forwards it to the UE, after the proper message header management.

Class::Method	Description
EpcEnbApplication:: RecvFromScmSocket (Ptr <Socket> socket)	The method allows the SCeNBce to receive messages from the SCM. According to their contents they are dispatched to the UE (message "offloading decision") or to the Hypervisor (Message "offloading request") to enable the calculation of the processing time needed inside the SCeNBcefor each foreseen configuration.
EpcEnbApplication:: RecvFromLteSocket (Ptr <Socket> socket)	The method allows the SCeNBce to receive messages from the UE. According to their contents they are dispatched to the Hypervisor to enable: - Message "offloading request", calculation of the processing time needed inside the SCeNBcefor each foreseen configuration - Message "i-th task", processing of the task (or a portion of it)
EpcEnbApplication:: DispatchDataFromHypervisor (Ptr <Packet> packet, MsgSCHyp msg)	The method allows the SCeNBce to receive messages from the Hypervisor.
EpcEnbApplication:: SendToLteSocket (Ptr <Packet> packet, uint16_t rnti, uint8_t bid)	The method allows the SCeNBce to send messages to the UE.
EpcEnbApplication:: SendToScmSocket (Ptr <Packet> packet, uint32_t teid)	The method allows the SCeNBce to send messages to the SCM.
ServerCallback:: SendServerMessage (Ptr <Packet> packet, MsgSCHyp msg)	The method allows the SCeNBce to send messages to the Hypervisor.

Table 9. SCeNBce Offloading and processing model: classes and methods

5 EPC MODEL

The *EPC model* implemented by *LENA* module has required a reduced number of modifications with respect to the work done on the *LTE model*. Most of them have involved the *EPC model Helper*. This is due mainly to:

1. the approach for the introduction of the Small Cell Cloud concept choosen by TROPIC has been to have the minimum impact on the existing LTE infrastructure (see D22);
2. the entire Cloud model has been developed as a new model that can be "installed" in whatever node of the *ns-3 framework*.

5.1 LENA-ns3 EPC model architecture

The EPC model provides the simulation objects needed to connect the LTE devices to the IP network for a complete end-to-end IP connectivity of the UEs to the nodes in the Core Network.
The EPC model encompasses the main entities defined in the LTE standard for the Core Network. In particular, the nodes Serving Gateway (S-GW), Packet Data Gateway (P-GW) and the Mobility Management Entity (MME) are modeled, with their communication protocols and interfaces.

Below are listed the main general characteristics of the EPC model, while the architecture is described in the following subsections. The same approach used for the LTE model, is also applied to illustrate the EPC model, separating the description in data-plane and the control-plane.

- The Serving Gateway (S-GW) and the PDN Gateway (P-GW) are implemented in a single network node; this simplification implies the absence of a model for the corresponding interface (S5/S8) between the gateways.
- Only the IPv4 type is supported in the Packet Data Network (PDN).
- The model manages several bearers for each UE, so it allows the simulation of multiple UE applications with different QoS requirements.
- The implemented network model allows the simulation of the applications working on top of TCP or UDP stack.

5.1.1 EPC data plane

The data plane, as it is implemented in the model provided by the *LENA* module is depicted in Figure 16 and explained in [Baldo-Miozzo-01]. As already mentioned, the EPC gateways P-GW and S-GW are modelled in a simplified manner, with their functionalities enclosed within a single node, and therefore, the interface between them is neglected.
The model provides two different layers for the network communication, for a complete connectivity of the LTE users. The first one involves the UEs, the P-GW and a generic remote hosts, final node of the communication, along with any eventual routers and hosts in between; the (SC)eNB is not involved in this kind of networking (it only provide the radio access).
The second layer concerns the eNB and the S-GW/P-GW node; this network is implemented as "point-to-point" links, one for each eNB that has to be connected to the gateways.
The end-to-end IP communications, as defined in 3GPP specification, is tunneled over the local EPC IP network using the GTP-U protocol.

The following two subsections describe schematically the processes of downlink and uplink communication; most of the "objects" involved in the explanation have been already mentioned in the *LTE model* (section 4) while the new ones can be found in the section dedicated to the EPC control-plane (section 5.1.2).

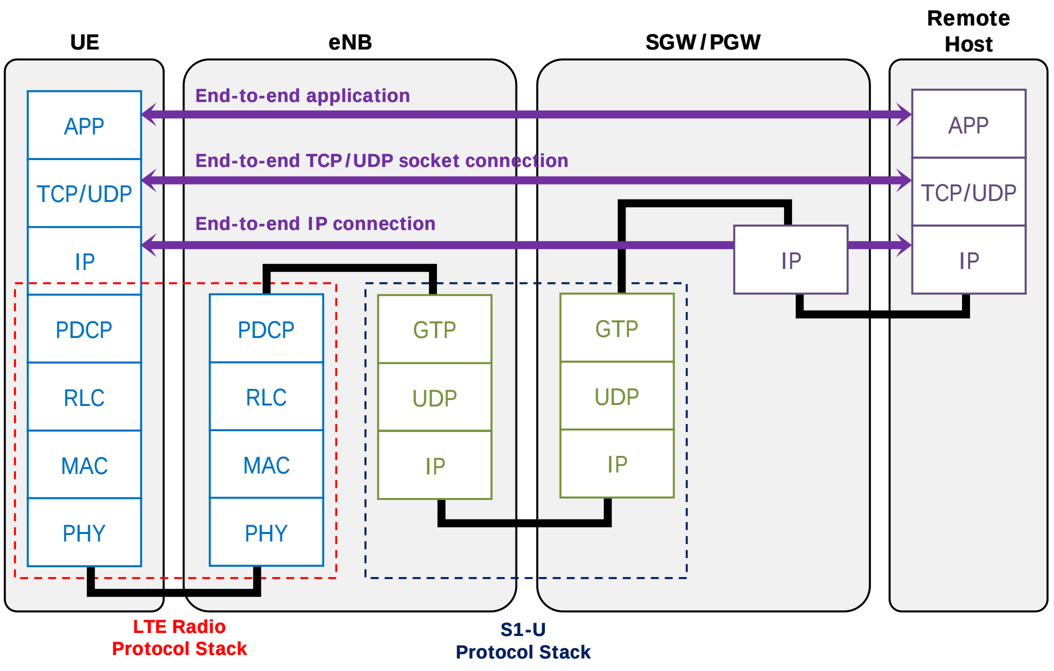


Figure 16. EPC data-plane

5.1.1.1 Downlink data flow

The data flow in downlink, that is, from an internet node to the UE, can be described through the following steps:

1. The packets are generated from a generic remote host, and addressed to one of the UE devices;
2. The internet routing protocol takes care of forwarding the packet to the *NetDevice* of the SGW/PGW node which is connected to the internet;
3. The SGW/PGW node has a *VirtualNetDevice* to which is assigned the gateway IP address of the UE subnet. The device starts the GTP/UDP/IP tunneling procedure, by forwarding the packet to a dedicated application, called *EpcSgwPgwApplication*;
4. This application does the following operations:
   a) it determines the eNB node to which the UE is attached;
   b) it classifies the packet using Traffic Flow Templates (TFTs) to identify to which EPS Bearer it belongs (this operation provides the GTP-U Tunnel End-point Identifier (TEID) of the packet);
   c) it adds the corresponding GTP-U protocol header to the packet;
   d) it sends the packet over an UDP socket to the S1-U point-to-point *NetDevice*, addressed to the eNB to which the UE is attached;
5. The end-to-end IP packet with newly added IP, UDP and GTP headers is sent through one of the S1-U links to the eNB, where it is received and delivered locally (as the destination address of the outmost IP header matches the eNB IP address).
6. The local delivery process forwards the packet, via an UDP socket, to a dedicated application called *EpcEnbApplication*. This application then performs the following operations:
   a) it removes the GTP header and retrieves the Tunnel End-point Identifier (TEID) which is contained in it;
   b) it determines the Radio Bearer ID (RBID) to which the packet belongs and is adds a tag to the packet;
   c) it forwards the packet to the *LteEnbNetDevice* of the eNB node via a raw packet socket.

7. When the *LteEnbNetDevice* receives the packet from the *EpcEnbApplication*, it performs the final action for the delivery (at this point, the outmost header of the packet is the end-to-end IP header, since the IP/UDP/GTP headers of the S1 protocol stack have already been stripped):
    a) it determines the Radio Bearer instance (and the corresponding PDCP and RLC protocol instances) which are then used to forward the packet to the UE over the LTE radio interface
    b) the packet is sent to *LteUeNetDevice*;
8. Finally, the *LteUeNetDevice* of the UE will receive the packet, and delivery it locally to the IP protocol stack, which, in turn, deliveries it to the application of the UE, which is the end point of the downlink communication.

5.1.1.2 Uplink data flow

The data flow in the opposite direction, that is, from the UE to an internet node is synthesized in the following sequence of operation:

1. Uplink IP packets are generated by a generic application inside the UE, and passed to the local TCP/IP stack;
2. The packets are forwarded to the *LteUeNetDevice* of the UE which performs the following operations:
    a) it classifies the packet using TFTs and determines the Radio Bearer to which the packet belongs (and the corresponding RBID);
    b) it identifies the corresponding PDCP protocol instance, which is the entry point of the LTE Radio Protocol stack for the packet;
    c) it sends the packet to the eNB over the LTE Radio Protocol stack.
3. The eNB receives the packet via its *LteEnbNetDevice*. Since there is a single PDCP and RLC protocol instance for each Radio Bearer, the *LteEnbNetDevice* is able to determine the RBID of the packet;
4. The RBID tag is added to the packet and the *LteEnbNetDevice* then forwards the packet to the *LteEnbNetDevice* via a raw packet socket.
5. The packet is received by the eNB where the *EpcEnbApplication* performs the following operations:
    a) it retrieves the RBID and it determines the corresponding EPS Bearer instance and GTP-U TEID;
    b) it adds a GTP-U header on the packet, including the TEID determined previously;
    c) it sends the packet to the SGW/PGW node via the UDP socket connected to the S1-U point-to-point *NetDevice*.
6. The packet is received by the corresponding S1-U point-to-point *NetDevice* of the SGW/PGW node, where the following operations are performed:
    a) it forwards the packet to the *EpcSgwPgwApplication* via the corresponding UDP socket;
    b) the *EpcSgwPgwApplication* removes the GTP header and forwards the packet to the *VirtualNetDevice*; at this point, the outmost header of the packet is the end-to-end IP header.
7. If the destination address within the header is a remote host on the internet, the packet is sent to the internet via the corresponding *NetDevice* of the SGW/PGW. Otherwise, in case the packet is addressed to another UE, the IP stack of the SGW/PGW will redirect the packet again to the *VirtualNetDevice*, and the packet will go through the downlink delivery process in order to reach its destination UE.

5.1.2 EPC control plane

The model of the control plane for the EPC, shown in the figure below, implements the S1-AP, the X2-AP and the S11 interfaces, but with different accuracy.

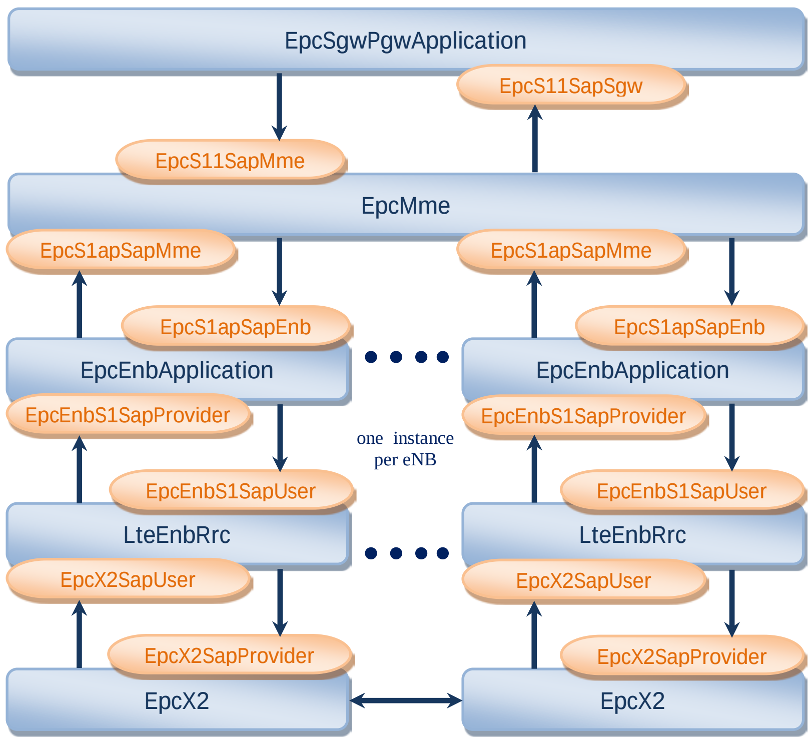


Figure 17. EPC control-plane

Object name	Description
EpcSgwPgwApplication	This class implements the SGW/PGW functionality.
EpcMme	This object implements the MME functionality.
EpcEnbApplication	This class is the same described in the eNB model. It models the application, installed inside eNBs, that provides the bridge functionality between the radio interface and the S1-U interface.
LteEnbRrc	This class models the LTE Radio Resource Control entity at the eNB; it's the same already used in the eNB model.
EpcS11SapSgw	This class implements the SGW side of the S11 Service Access Point (SAP) for managing the S11 messages received by the SGW.
EpcS11SapMme	This entity models the MME side of the S11 Service Access Point (SAP); it provides the MME utility to manage the S11 messages received by the MME.
EpcS1apSapMme	This class provides the Service Access Point (SAP) of the S1-AP at MME side, in order to manage the messages received by MME on such interface.
EpcS1apSapEnb	This class is the eNB side of the S1-AP Service Access Point (SAP); it allows the eNB to manage the received S1-AP messages.
EpcEnbS1SapProvider	These classes implement the Service Access Point (SAP) between the LteEnbRrc and the EpcEnbApplication. They are the same already described in the eNB model.
EpcEnbS1SapUser	
EpcX2SapUser	These two classes model the interface between the X2 entity and the RRC entity; they are already described in the eNB model.
EpcX2SapProvider	

Table 10. Description of the EPC control-plan classes

The S1-AP and the S11 interfaces are modeled in a simplified manner, by using the interface classes to model only the interaction between the involved nodes (the eNB and the MME for the S1-AP interface, and the MME and the SGW for the S11 interface). In detail this means that, in the model, the interfaces are directly mapped to a function call between the two entities without actually implementing the encoding of the messages and without actually transmitting any PDU on any link.
The X2-AP interface, on the other hand, is being implemented with a more realistic model, using protocol data units sent over the X2 link (modeled as a point-to-point link).

5.2 Extended EPC model

Most of the activity on the EPC model has been done on its *helper* for the creation of the proper interconnections of the SCM and the cluster of computing Small Cells with the nodes of the scenario that have to interact with them. The implementation details concern only the generic mechanisms of the *ns-3 framework* and are not described here.

The second kind of activity on the *EPC model* has regarded the introduction of the backhaul model obtained with the measurement campaign performed in WP5. It allows the modelling of the network congestion as backgroung traffic that produces as effect an additional delay and jitter in the exchange of the packets with remote nodes and, therefore only in the offloading towards the Centralized Cloud.
Its implementation is straightforward with the random reading of the values provided by the backhaul model, stored in a file.

6 CLOUD MODEL

The definition *Cloud model* encompasses all the objects involved in the simulation of a cluster of nodes with a virtualized environment for the processing of an application. These objects are the cluster of SCeNBces and its manager, the SCM, but regarding the SCeNBce, only the simulation of the virtual platform is included in this model, while the simulation of all the radio layers is still part of the *LTE model*. The *Cloud model* from this initial purpose has been extended to the simulation of a conventional Centralized Cloud for a comparison with the remote processing offered by the usual Cloud Provider.

6.1 Small cell Cloud Manager (SCM)

The Small Cell Cloud Manager is implemented as a centralized “entity” and hence it lies entirely in a single node, with “point-to-point” links created with all the SCeNBces of the cluster, as showed in Figure 3.

The setup of these communication links is exclusively for the purpose of the creation of the simulation scenario, but it does not implement a true star topology as the mentioned figure might suggest. The label “point-to-point”, too, states only the creation of a link between the *NetDevices* of a couple of nodes, not the the protocol used in these links. Section 10 explains in detail that the simulator needs to define, for every node in the scenario, the type of channel and protocol used by the *NetDevice* installed on each node. Whatever the topology of the cluster, the "link" between each SCeNBce and the SCM has to be configured, in order to properly simulate the IP communications. The simulator uses then the so called "*global routin*g" functionality to allow the SCeNBce to reach the IP address of the SCM.
The only simplification is the absence of a router as a separate node. In a scenario with a router implemented, all the SCeNBces would have had this link with the router instead of with the SCM and this latter would have had a link with the router. The difference is negligible.

These connections are created and configured, in terms of IP address, port and “channel” characteristics, during the initial setup in order to build a cluster immediately ready, without the need of its dynamic creation at run-time.
The assumption underlying this approach implies that, at the starting of a simulation, the SCM already has all the information about the nodes to manage and, mainly, the initial negotiation and admission control is considered already done.

The deployment of the Virtual Machine on the SCeNBce and the destruction at the end of its usage, is not implemented. During the setup of the cluster, each SCeNBce is created with all the Primary Virtual Machines associated to every UE with cloud services enabled. The Virtual Machines are ready for running UE applications, without any action performed by the SCM. This is the result of the deployment of the UEs inside the scenario. As described in sections 9 and 10, at the beginning of the simulation all the UEs are attached to their serving cell, included the UEs with cloud capability, and no new UEs are added to the scenario at run-time, so there isn’t the need of implementation of such functionalities.

The SCM implements two different algorithms for the selection of the best configuration for the offloading parameters, while the algorithms proposed to define the Service Model of the SCM are better evaluated in the second activity of WP6, thanks to the implementation in the “proof of concept” where real devices with virtualized platform are used.

6.1.1 SCM model

The Small cell Cloud Manager basically responds to two different kinds of events:

- <u>An offloading request from a UE</u>: this is the first message sent by a UE every time a new application starts and it is needed to obtain the best configuration for the offloading parameters. The optimal configuration is pinpointed using a set of information that depends on the type of algorithm has been selected for the simulation; the data are provided by the UE and / or the Hypervisor on the SCeNBce. The offloading decision result, that is the optimal configuration to use, is then sent to the UE through the attached SCeNBce and to the Hypervisor as well.

- A status message from the SCeNBce: this can be the answer to a request sent by the SCM itself, by means of the *Monitoring module* which periodically (dT) sends a request to all the nodes of the cluster to obtain the status. This is the case only if *reactive mode monitoring* is enabled. Otherwise (*proactive mode monitoring*) the status message is spontaneously dispatched by the SCeNBce when the status of the resources allocation change significantly. In this latter case, *VMInfoRequest()* with its auto-scheduling mechanism in the *system event-queue*, shown in Figure 18 is not active.

The SCM makes use of a database for managing the information about all the Small Cells belonging to the cluster and their UEs, constantly updated by means of the status messages and through the information contained inside the offloading request messages coming from the UEs. Its structure is briefly described in the following section.

The decision about the optimal parameters for performing the offloading is taken by the SCM using one of the two possible algorithms, selected at the startup by means of a specific configuration attribute. The first one, the *Energy-Latency Tradeoff* algoritm is directly implemented, while the second, the *Joint Radio-Cloud Optimization* algorithm make use of the extern library described in section 7.1.3.

The main classes and their most important methods are described in Table 11 while Figure 18 shows the interaction among them.

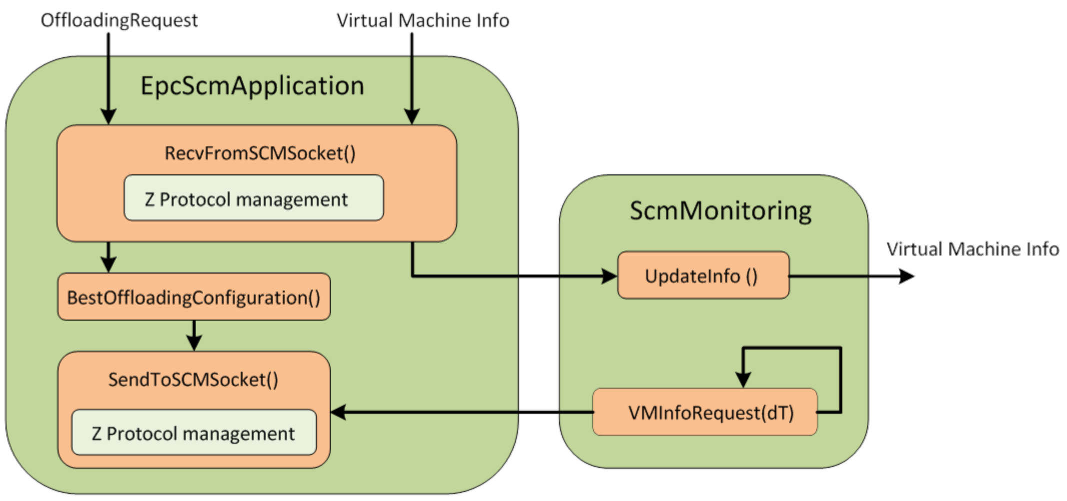


Figure 18. SCM model

Inside the two methods *RecvFromSCMSocket()* and *SendToSCMSocket()* (class *EpcScmApplication*) the incoming and outgoing packet headers are managed (Z-Protocol) to detect the packet type and the related info.

Class::Method	Description
EpcScmApplication:: RecvFromScmSocket (Ptr <Socket> socket)	The method allows the SCM to receive messages from the SCeNBce. According to their contents they are dispatched to the Hypervisor (Message "VM info") to update the SCM knowledge about the VM current status or used to define the optimal offloading configuration.
EpcScmApplication:: BestOffloadingConfiguration (InfoRequestMessage infoReq)	The method selects the optimal offloading configuration on the basis of the processing times (inside and outside the UE), of the energies spent for processing (inside the UE) and of the energy spent for communications if portions of the processing are performed outside the UE.
EpcScmApplication:: SendToScmSocket (Ptr <Socket> socket)	The method allows the SCM to send messages (offloading decision, VM info request) to the SCeNBce.
MYEpcScmApplication:: ExtLibOffloadParamsUpdateCallback (std::vector<double> *outTxPow, std::vector<double> *outCompRes, uint16_t ueCellId, double appLatency, int ueNum, std::vector<uint16_t> ueId, std::vector<double> ueTxBit, std::vector<double> ueInstrNum)	This method is the callback between SCM and *Extern Library Manager* for the implementation of the *Joint Radio-Cloud Optimization*. For the description of the input parameters see section 7.1.3
ScmMonitoring:: UpdateInfo (void)	The method updates the SCM data-base including the current status of the VM of each SCeNBce.
ScmMonitoring:: VMInfoRequest (double dT)	The method activates at fixed rate (dT) an info request toward the SCeNBce in order to obtain the current status of the VM of each SCeNBce.
void MYEpcScmApplication:: AddSmallCellToDb (uint16_t cellId, uint16_t ueNum)	This method is used during the initial setup for the creation of the Small Cell database.

Table 11. SCM model: classes and methods

6.1.2 Small Cell Database

The database is structured as a *Container* of objects defined *Scm Small Cell Info*. The class *container* is one of the tool made available by the *ns-3 framework*. The table below describes the main method.

Method	Description
uint32_t GetN (void)	This method provides the current number of *Scm Small Cell Info* object stored in the container
Ptr <ScmSmallCellInfo> Get (uint32_t index)	The method gives back the *Scm Small Cell Info* at the position indicated by index provided.
void Add (Ptr < ScmSmallCellInfo > scInf)	The method simply adds a new *Scm Small Cell Info* object inside the container.
void Clear (void)	This method removes all the *Scm Small Cell Info* objects from the container.
Ptr< ScmSmallCellInfo > GetScInfoFromCellId (uint16_t cellId)	This method allow to retrieve the *Scm Small Cell Info* object starting from the cell ID.
void StoreStatusData (hypervisorStatusMsg hypStat)	The method is in charge of storing inside the database the information sent by the Hypervisor.
void ScInfoSetUeTotalNum (uint16_t ueNum)	These are a set of method used for storing and getting a single specific information from the database
uint16_t ScInfoGetUeTotalNum (void)	
void ScInfoSetUeActiveNum (uint16_t ueNum)	
uint16_t ScInfoGetUeActiveNum (void)	
std::vector<uint16_t> GetActiveUeId (void)	
std::vector<double> GetActiveUeTxBit (double taskTxBit, uint16_t taskNum)	

Method	Description
std::vector<double> GetActiveUeInstrNum (double taskInstrNum, uint16_t taskNum)	

Table 12. Small Cell Database: class methods

6.2 SCeNBce model (virtualized environment)

The SCeNBEce, for the "cloud side", is composed of the following four main objects (classes), according to the model proposed in [TROPIC-D51]:

1. Hypervisor
2. Scheduler
3. Virtual Machine
4. Virtual CPU

In addition to these classes, the *Helper* class has been implemented in order to provide a set of methods for creating the virtualized environment inside the SCeNBce node, following the methodology of the *ns-3 framework*.

In the following sub-sections these classes are illustrated through a brief description of their main methods, while the interaction among them during the processing of the applications is explained in section 6.3, where the whole implementation is described in detail.

6.2.1 Hypervisor model

The class Hypervisor is in charge of the management of the Virtual Machines deployed in the node; this task is performed using a special class, named *container*, used to store the Virtual Machines of the node. The *virtual machine container* allows to easily access to the status and the resources of the Virtual Machines and perform the needed operations on them.

The container of Virtual Machines is the tool used to store such simulation objects, part of the Cloud model. The Hypervisor leverages on additional items for managing all the information associated to the Virtual Machines, without the need of continuously consult each Virtual Machine to obtain them. During the initial setup, the Hypervisor creates, for each Virtual Machine a *VirtualMachineInfo* element and stores them in a kind of "database" that has a structure similar to the container class. The *VirtualMachineInfo* is used to store, for instance, some information contained in the messages received from the "communication module", which have to be inserted inside the answer message, sent at the end of the processing.

As already mentioned, the Hypervisor is "installed" during the initial setup, as well as the Virtual Machines associated with each UE. This means that, unlike the real tasks, the simulated Hypervisor does not really deploy and destroy the Virtual Machines, but simply coordinates the processing requests and the monitoring process.

The Hypervisor interacts with the "communication module", belonging to the Radio part of the SCeNBce, from which it receives the input of new actions to accomplish and to which it provides the results of the requests. In detail, the Hypervisor performs the following actions:

1. receives the task processing requests and assigns them to the proper Virtual Machine, depending on the UE ID;
2. sends back the output when a task completes the processing;
3. collects the information about the Virtual Machines for monitoring purposes;
4. provides the estimation of processing time for the offloading algorithm.

Table 13 describes the main methods of the class *Hypervisor*. The auxiliary methods, such as those required for accessing the private variables of the class, are not shown in the table, but only those that implement the main functionalities of the Hypervisor. Not even the standard methods, such as “constructor” and “destructor”, are shown in the table. The full description of the interaction with the other classes of the SCeNBce model is reported in section 6.3.

Method	Description
void AddVirtualMachine (Ptr<VirtualMachine>)	These methods are used by the *helper* for adding and removing a Virtual Machine to/from the private container of the Hypervisor, during the initial setup.
void DeleteVirtualMachine (Ptr<VirtualMachine>)	
void SetScheduler (Ptr<ProcessingScheduler>)	The method initializes the scheduler after its creation to allow the Hypervisor to interact with it.
void requestManager (reqInfo)	This method manages all the requests coming from the radio side. It is called by means of the *callback* mechanism and receives the information about the incoming request through the structure *reqInfo*. In case a new task to process arrives, it starts the actions described in section 6.3.
void requestCompleteNotifier (reqInfo)	The method has the task to inform the communication module about the completion of the received request, in order to send back the output. The structure *reqInfo* contains the information for the correct dispatching of the data.
void nodeMonitoring (void)	This method is used to gather the information about the status of all the Virtual Machines for the purpose of periodic monitoring of the cluster performed by the SCM. The data are sent to the communication module to forward them to the SCM.

Table 13. Hypervisor: class methods

6.2.2 Scheduler model

The Scheduler implements the allocation of the computational resources to the active Virtual CPU to allow them to process the assigned task. The shared resources are allotted to the VCPUs using the algorithm *simplified fair work-conserving CPU scheduler* (with variable VCPU timeslice), described in [D51]. The implementation of the algorithm is slightly different from the original proposal for adapting it to the *ns-3* characteristics, reducing the number of simulation events. In the original formulation, most of the events are not essential because they are related to the update of the internal status, before the end of the processing and, therefore they don't produce any reaction in the simulation. The implementation of the scheduling algorithm is explained in detail in section 6.3.

The core of the Scheduler is the function *ProcessingEvent()* which manages the two possible processing event:

1. a task, previously assigned to a VCPU, has completed (case *END_TASK*);
2. the request of a new task to process in a VM (case *NEW_TASK*).

The first one is an event that the Scheduler itself has added to the simulator event queue, evaluating in advance the time required for completing the processing, on the basis of the current resources allocation among the active VCPUs. The second one, instead, is a completely asynchronous event, subject to the offloading requests from the UEs and the following decision taken by the SCM.

In this latter case the status of all the VCPU and the scheduling queue has to be updated because of a new VCPU to which part of the computational resources have to be assigned. Being an asynchronous event, the Scheduler has to find out which is the VCPU currently scheduled in order to correctly define the new sequence of scheduling for the VCPUs.

The case *END_TASK* as well, implies the update of the status of all the active VCPUs, but the following sequence of scheduling is most easily defined because all the events were already known.

The Scheduler as well, makes use of the special class *container*, tailored for the VCPU, in order to manage the VCPU with “running” task. The *Virtual CPU container* is used by the Scheduler as a scheduling queue where a VCPU is added when it starts the processing and is removed at the completion.

Method	Description
void ProcessingEvent (eventType newEvent, eventInfo info)	The method accomplishes the whole sequence of operations needed to manage one of the two processing events, the case *END_TASK* and the case *NEW_TASK*.
Ptr<VirtualCpu> NextProcessingEvent (uint32_t *nextEventTime)	This method is in charge of defining which VCPU, among the active ones, will be the next to complete the processing of the assigned task. Using the current values of timeslice and computational resources assigned, it finds out the VCPU that requires the minimum number of timeslice. It also computes the corresponding time of the event, in order to add it to the event queue of the simulator.
void UpdateProcessingParameter (void)	The method computes and updates the parameters *timeslice* and *computational resources* assigned to all the active VCPUs.
void UpdateProcessingStatus (void)	This method imposes to all the active VCPUs the update of their processing load (the residual number of instructions), through the method *VirtualCpu::VcpuUpdateProcessingLoad()*, providing them the information about the number of scheduled timeslices since the last event.
void FindLastScheduledVcpu (void)	The method is used to find out the last VCPU scheduled when the request of processing a new task comes to the node. Using the current timeslice assigned to the active VCPUs, the time elapsed since the last event and the scheduling order, it locates the VCPU that has the availability of the shared resources, at that time. This method is used only in case of *NEW_TASK* because, in case of *END_TASK*, the needed information is computed in advance when the next event is scheduled.
void SetHypervisor (Ptr<Hypervisor> hypervisor)	The method initializes the connection with the Hypervisor to allow the Scheduler to interact with it.

Table 14. Scheduler: class methods

6.2.3 Virtual Machine model

The class Virtual Machine abstracts the physical resources of the node where it is deployed, according to the simulation model proposed in [D51], and derived from [D42] and [D52]. The Virtual Machine selects the Virtual CPU to which a task to process must be assigned when the request is received from the Hypervisor and it manages the status of the parameters involved in the scheduling. A private *container* is used for accessing to the VCPUs.

The most important methods are described in the following table, while, for simplicity, the large number of methods created to manage all the parameters of the model (private), are not shown.

Method	Description
void VmResourcesAllocation (int virtualCpuNum, int virtualMemory, int virtualDisk, int bandwidth)	This method is in charge of configuring the resources assigned to the Virtual Machine at creation time.
void VmSetScheduler (Ptr<ProcessingScheduler> scheduler)	The method initializes the interface with the Scheduler in order to interact with it.
void VmAddVcpu (Ptr<VirtualCpu> newVcpu)	This method inserts a VCPU in the internal container.
void VmTaskStartProcessing (procRequestInfo newTaskInfo)	The method is in charge of starting the processing of a new task. It selects the VCPU to which the task must be assigned, updates the internal status of the VM and then signals the event to the scheduler.
void VmTaskEndProcessing (void)	This method is the dual one for the previous. It manages the actions to accomplish when a VCPU ends the processing of the assigned task. After the update of the internal status, it informs the Hypervisor for sending to output to the UE.
void VmUpdateVcpuWeight (void)	This method computes the new value of the weight and updates the parameter in all the active VCPUs.

Table 15. Virtual Machine: class methods

6.2.3.1 Virtual Machine container

The *Container* class is a typical simulation object of *ns-3 framework*, created with the purpose of holding together homogeneous elements, providing the tools to facilitate access to them. With the same approach, the *Virtual Machine Container* class has been built, mainly for the management of the Virtual Machine deployed in the node by the Hypervisor. The main methods of the class are shown in the table below.

Method	Description
uint32_t GetN (void)	This method provides the current number of Virtual Machines stored in the container
Ptr<VirtualMachine> Get (uint32_t index)	The method gives back the Virtual Machine at the position indicated by index provided.
void Add (Ptr<VirtualMachine> vm)	This method adds a Virtual Machine to the container.
void Clear (void)	The method removes all the Virtual Machine from the container.

Table 16. Virtual Machine Container: class methods

6.2.4 Virtual CPU model

The Virtual CPU class models the computing units of a Virtual Machine. According to the proposed simulation model, the processing of an Application (its component tasks) doesn’t implement its execution, but simply considers the time required for processing the amount of instructions of the task, related to the computation power available and the timeslices assigned to it by the scheduler. Each Virtual CPU manages and updates its status and processing parameters, making them available to the Scheduler and the Virtual Machine to which belongs.

Method	Description
void VcpuSetScheduler (Ptr<ProcessingScheduler> scheduler)	The method sets up the interface towards the Scheduler for the interaction with such a class.
void VcpuSetVirtualMachine (Ptr<VirtualMachine> virtualMachine)	This method configures the interface with the Virtual Machine the VCPU belongs to.
void VcpuUpdateProcessingLoad (uint32_t executedTimeslice)	The method receives from the Scheduler the number of timeslices executed (how many times the VCPU has been scheduled), needed to compute, along with the assigned computational resources and timeslice, the residual number of instructions.
void VcpuStartProcessing (uint32_t taskId, float taskInstrNum)	This method updates the internal processing variables, in particular the status (from FREE to BUSY), the computational load and the ID of the task assigned.
void VcpuEndProcessing (void)	This is the method scheduled by simulator through the system events-queue at the end of the processing of the assigned task. It informs the Scheduler and its Vitual Machine about the end of the processing.

Table 17. Virtual CPU: class methods

6.2.4.1 Virtual CPU container

The container of Virtual CPUs is used by the Virtual Machine to keep its VCPUs for a simplified access and management; this container is populated during the setup of the Cloud model and is a permanent private object. A VCPU container is used by the Scheduler as well, but the utilization is different. The container acts as a processing queue, so the VCPUs are added when they get active, after a processing task is assigned to them, but are then removed from the container when the processing is complete.

Method	Description
uint32_t GetN (void)	This method provides the current number of Virtual CPUs stored in the container
Ptr<VirtualCpu> Get (uint32_t index)	The method gives back the VCPU at the position indicated by index provided.
void Add (Ptr<VirtualCpu> vcpu)	The method simply adds a new VCPU at the end of the processing queue (container), when a VCPU gets active after a task processing request.
void Clear (void)	This method removes all the Virtual CPU from the container.
void Remove (Ptr<VirtualCpu> vcpu)	The method removes a specific VCPU from the container.
void SwapOrder (uint32_t swapIndex)	The method updates the VCPU order inside the scheduling queue on the base of the position index of the last scheduled VCPU. The VCPUs with higher index are moved at the beginning of the queue, because not scheduled before the last event, while the VCPUs with lower index are moved at the end.

Table 18. Virtual CPU Container: class methods

6.2.5 Virtual processing model Helper

The *Helper*, as well as the *Container* class, is a typical tool of *ns-3 framework*, used for creating and configuring a simulation object in the main script. It is important to highlight that role of this class is provide the proper means for creating the objects of the Cloud model, Hypervisor, Scheduler and Virtual Machines, during the initial setup. It is no more used at run-time. The main methods of this class are described in the following table.

Method	Description
void CreateVirtualNode (void)	This method is the main one in the creation of the virtual environment on the SCeNB. It uses the other methods described below to accomplish all the required setup operations.
Ptr<Hypervisor> InstallHypervisor (void)	This method creates and configures the Hypervisor in the node.
Ptr<ProcessingScheduler> InstallScheduler (void)	The method is used to provide the node shared computing resources management.
Ptr<VirtualMachine> CreateVirtualMachine (int virtualCpuNum, int virtualMemory, int virtualDisk, int bandwidth)	This method is in charge of the creation and configuration of a Virtual Machine. It should be noted that, a Virtual Machine once created, creates, in turns, its Virtual CPUs as private objects.
Ptr<ProcessingScheduler> GetScheduler (void)	This method is used to provide the access to the scheduler to all the simulation objects of the model.

Table 19. Virtual processing model Helper: class methods

6.3 Processing on the SCeNBce

6.3.1 Implementation of the Scheduler

The simulation of an application running on a virtualized platform is based on the two concepts of *task* and *timeslice*, according to the simulation models described in [D51]. Just for clarity, a brief summary is provided below.

The *task graph model* considers the application composed by a set of "units of computation", named *task* and, depending on the accuracy of the model, a *task* may correspond to the entire application or one of its functions. The tasks can be executed in parallel or in serial manner and, in the execution chain, a task receives input data from the one executed before and provides its output data to the next task to be executed.

The *timeslice*, in the *simplified fair work-conserving CPU* model, is the amount of time assigned by the scheduler to a CPU (or *Virtual CPU* in case of virtualized environment) for using the computational resources, shared with other concurrent VCPUs with a task in execution. Moreover, the "variable timeslice" is used to take into account the priority of the VCPUs. Beyond the algorithm used for managing the scheduling order, the essence is that, at the end of every timeslice, the current VCPU frees the resources, allowing another VCPU to take the control to accomplish a piece of its processing.

As long as no new VCPU requires access to shared resources and no VCPU completes its task, the allocation of the resources remains unchanged and the progression of the processing of all the active VCPU is totally predictable.

This is the reason why, from the point of view of the simulator, it's not necessary update the status of the SCeNBces with a time scale based on the timeslice because, for most the timeslices, the allocation of the computational resources among the VMs/VCPUs doesn't change.

As mentioned, the only events to take into account are:

1. the beginning of a new elaboration task (*NEW_TASK*),

2. the end of a running task (*END_TASK*).

The first one is an external event that happens when a new application from a UE gets accepted, while the second one is an internal event, scheduled (in the *simulator event queue*) a priori and computed using the knowledge of the resources allocation status and the scheduling policy.

These events cause a change in the number of active VMs/VCPUs and hence a new allocation of the computational resources among the active Virtual Machines and their VCPUs is required. At the occurrence of an event, the Scheduler has to update:

1. the weight of the VCPUs,
2. the timeslice assigned to the VCPUs,
3. the computational resources allocated to VCPUs,
4. define the new scheduling order for the VCPUs in the queue.

In the interval of time between two events, all the processing parameters are constant and known. This way the simulator can avoid the generation of a huge number of simulation events (each timeslice for all the SCeNBce).

6.3.2 Model details

The model *simplified fair work-conserving CPU scheduler* (with variable VCPU timeslice), described in [D51], can be implemented with the following steps, executed at the occurrence of one of the events above described. The formulas used for evaluating the new processing parameters are shown. Most of them correspond to those of the [D51], but in some cases they have been adapted to the approach followed in the implementation.

1. update of the number of active VMs/VCPUs and their weights

2. for each of *L* active $VCPU_i$ evaluation of the **new timeslice** (TS_i) (equation (50) in [D51]):

$$VCPU_i\ timeslice\ (ms) = \frac{30 * VCPU_i\ weight * L}{\sum_{k=1}^{L} VCPU_k\ weight}$$

   Where:
   1. 30 (ms) is the default timeslice,
   2. The VCPU weight is the VM weight divided the number of VCPU active on the its VM,
   3. *L* is the total number of the active VCPUs (sum of all the VCPU of all the VMs)

3. for each of *L* active $VCPU_i$ evaluation of the **new computational capability** (CC_i). Using one general queue rather than individual queues for each physical core, the computational capability should be expressed as a weighted percentage of the total computation capability of the CPU ($C * SP$, where SP is the speed (MIPS) of the C cores):

$$CC_i = \frac{SP * C * VCPU_i\ weight}{\sum_{k=1}^{L} VCPU_k\ weight}$$

4. for each of *L* active $VCPU_i$ evaluation of the **residual elaboration** NI_i (number of instructions), based on the time elapsed from the previous event, the number of timeslices executed, the previous values of timeslice and computation capability;

5. for each of *L* active $VCPU_i$ computation of the **number of timeslices** (NT_i) needed to complete the elaboration, using the new values of TS_i, CC_i and NI_i

$$NT_i = \frac{NI_i}{CC_i * TS_i}$$

(basically this is the residual elaboration time divided by the timeslice)

6. The minimum value of NT_i (NT_{MIN}) is used to compute and schedule the next event, that is, the end of the elaboration of the VCPU° corresponding to NT_{MIN}. Considering that <u>not all the active VCPU will execute their timeslice NT_{MIN} times, depending from the scheduling order of the VCPUs</u>, the next event will occur at time T_{NEXT} given by:

$$T_{NEXT} = NT_{MIN} * \sum_{k=1}^{L1} TS_k + (NT_{MIN} - 1) * \sum_{k=1}^{L2} TS_k$$

Where:
L1 is the number of VCPU scheduled before VCPU° (VCPU° included)
L2 is the number of VCPU scheduled after VCPU°
L = L1 + L2

It should be noted that the next event computed as described, will be the next event scheduled by the simulator if, and only if, there won't be in the meantime a new processing request.

The subsequent section will show a numeric example, with graphical representation as well, of the implementation, just described, of the scheduler model. With reference to this example, the sixth point can be explained better. In the example, the *event #2* corresponds to the end of VCPU11, after 3 TS. VCPU31 and VCPU32 will execute 3 TS as VCPU11, while VCPU12 will execute only 2 TS. After the *event #1*, the scheduler updates all the processing parameters and computes the next event. The event #2 will be added to the event queue of the simulator at the time:

$$T_{NEXT} = 3 * (TS_{11} + TS_{31} + TS_{32}) + 2 * TS_{12}$$

6.3.3 Numeric example

The numeric example illustrated in this section helps to verify the correct interpretation of the model and explain how the simulator can implement it using the events associated to the tasks instead of the events produced by the timeslice scheduling. Table 20 shows a short sequence of processing events in a SCeNBce with three VMs, as depicted in Figure 19. The initial values of the parameters correspond to the status of a generic elaboration time with all the five VCPUs are active; after each event the status of all the active VCPU is updated. The coloured boxes in the event rows, represent the repetition of the timeslices scheduled for each active VCPU, and show how the duration of the timeslice is changed according to the new conditions.

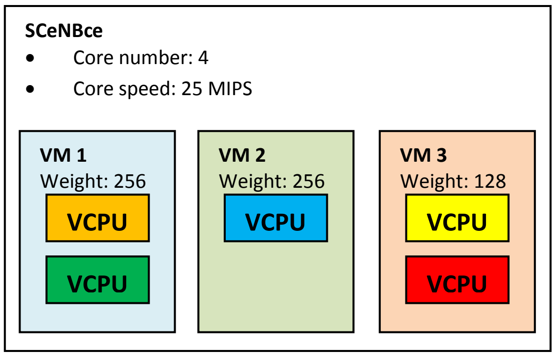


Figure 19. Scenario of the example

This example has been implemented inside the simulator as well, in order to test the proper behaviour of the source code written.

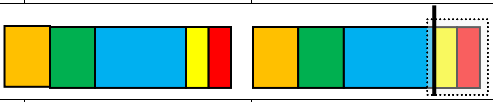
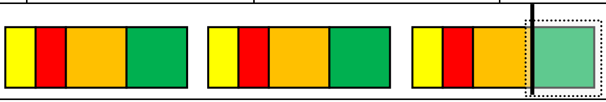
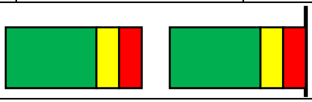
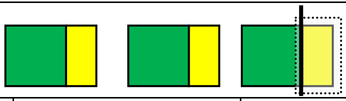
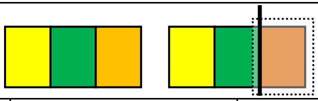
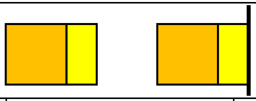
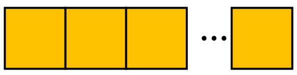
	Weight	Timeslice [ms]	Computation capability [MIPS]	Computational load (residual) [MI]	Processing per TS
VCPU11	128	30	20	5.19	0.6 MI/TS
VCPU12	128	30	20	21.80	0.6 MI/TS
VCPU21	256	60	40	4.8	2.4 MI/TS
VCPU31	64	15	10	6.26	0.15 MI/TS
VCPU32	64	15	10	1.64	0.15 MI/TS
Event #1 (END_TASK)	**VCPU21 completes the elaboration in 2 TS** 				
VCPU11	128	40	33.3	3.99	1.33 MI/TS
VCPU12	128	40	33.3	20.60	1.33 MI/TS
VCPU21	No more active				
VCPU31	64	20	16.6	6.11	0.33 MI/TS
VCPU32	64	20	16.6	1.49	0.33 MI/TS
Event #2 (END_TASK)	**VCPU11 completes the elaboration in 3 TS** 				
VCPU11	No more active				
VCPU12	256	60	66.6	17.94	3.99 MI/TS
VCPU21	No more active				
VCPU31	64	15	16.6	5.12	0.25 MI/TS
VCPU32	64	15	16.6	0.50	0.25 MI/TS
Event #3 (END_TASK)	**VCPU32 completes the elaboration in 2 TS** 				
VCPU11	No more active				
VCPU12	256	40	66.6	9.96	2.66 MI/TS
VCPU21	No more active				
VCPU31	128	20	33.3	4.62	0.66 MI/TS
VCPU32	No more active				
Event #4 (NEW_TASK)	**VCPU11 gets active (event not pre-computed)** 				
VCPU11	128	30	33.3	18.5	0.99 MI/TS
VCPU12	128	30	33.3	1.98	0.99 MI/TS
VCPU21	No more active				
VCPU31	128	30	33.3	3.30	0.99 MI/TS
VCPU32	No more active				
Event #5 (END_TASK)	**VCPU12 completes the elaboration in 2 TS** 				
VCPU11	256	40	66.6	17.51	2.66 MI/TS
VCPU12	No more active				
VCPU21	No more active				
VCPU31	128	20	33.3	11.32	0.66 MI/TS
VCPU32	No more active				
Event #6 (END_TASK)	**VCPU31 completes the elaboration in 2 TS** 				
VCPU11	256	30	100	12.19	3 MI/TS
VCPU12	No more active				
VCPU21	No more active				
VCPU31	No more active				
VCPU32	No more active				
	VCPU11 is scheduled until the end 				

Table 20. Example of processing events

6.3.4 Description of the processing and interactions among the classes

The two processing events, indicated with labels *NEW_TASK* and *END_TASK*, cause a sequence of actions, in the *Hypervisor* and in the *Scheduler*, similar, but not completely. The differences in the event management arise from the fact that in one case, *END_TASK*, all the processing parameters are known because computed when the occurrence of the successive event is assessed and it is inserted into the *system events-queue*. Conversely, in case of *NEW_TASK*, the event is totally asynchronous and this implies that (1) the event previously estimated to be the next to occur, has to be removed from the *events-queue* and (2) all the processing parameters of the active VCPUs have to be computed again considering the additional VCPU introduced in the processing queue.

This implies that the Scheduler has to accomplish almost (but not exactly) the same actions in the two cases, but they are performed in a different order, as detailed in the next sub-sections and graphically shown in Figure 20 and Figure 21.

6.3.4.1 A new task starts the processing

The request of processing a new task comes to *hypervisor::requestManager()* from the communication modules in the Radio part. It locates the Virtual Machine to which the request is addressed and signals the event by means of *VirtualMachine::VmTaskStartProcessing()*.

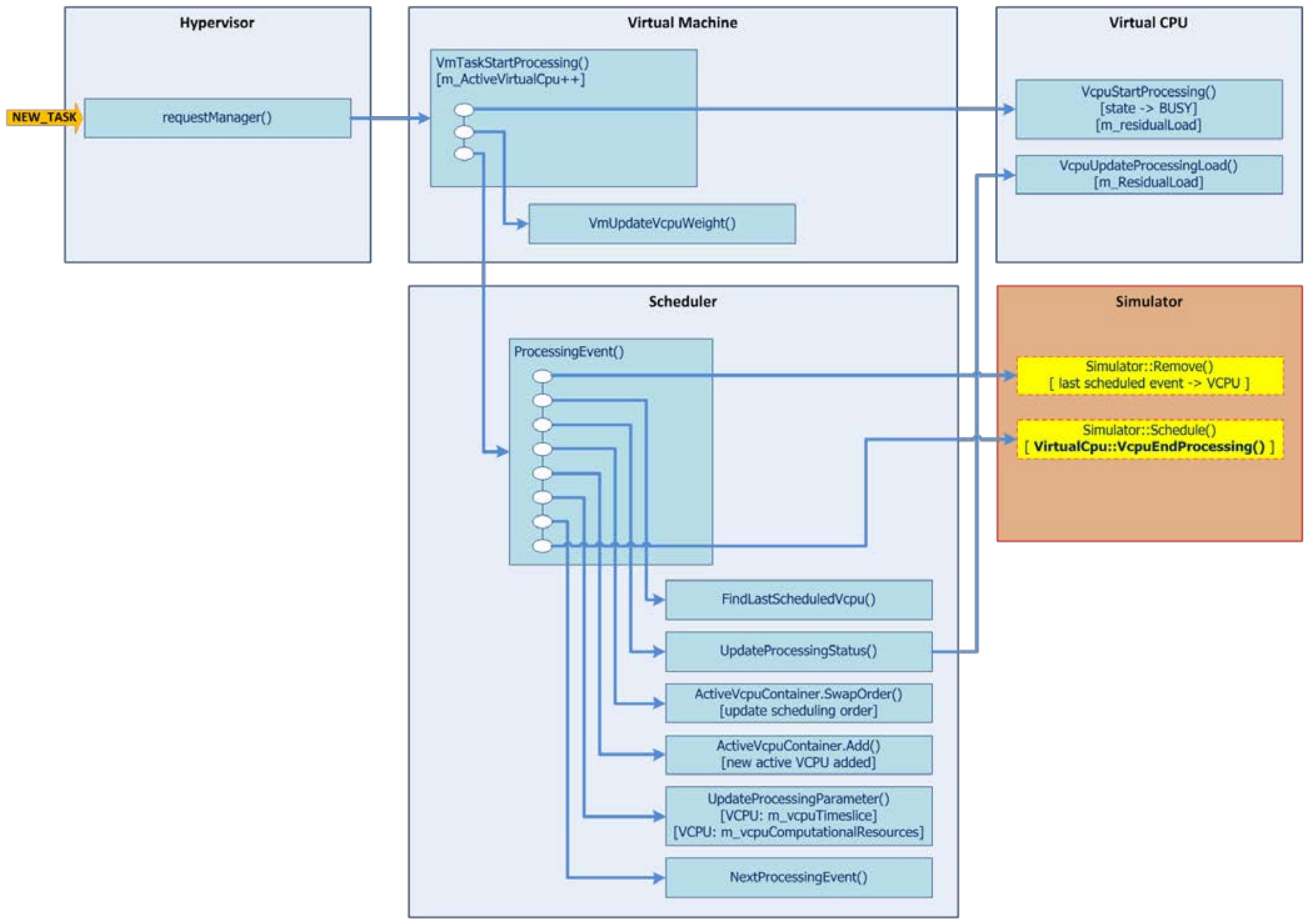

Hypervisor
NEW_TASK
requestManager()
Virtual Machine
VmTaskStartProcessing() [m_ActiveVirtualCpu++]
VmUpdateVcpuWeight()
Virtual CPU
VcpuStartProcessing() [state -> BUSY] [m_residualLoad]
VcpuUpdateProcessingLoad() [m_ResidualLoad]
Scheduler
ProcessingEvent()
FindLastScheduledVcpu()
UpdateProcessingStatus()
ActiveVcpuContainer.SwapOrder() [update scheduling order]
ActiveVcpuContainer.Add() [new active VCPU added]
UpdateProcessingParameter() [VCPU: m_vcpuTimeslice] [VCPU: m_vcpuComputationalResources]
NextProcessingEvent()
Simulator
Simulator::Remove() [last scheduled event -> VCPU]
Simulator::Schedule() [VirtualCpu::VcpuEndProcessing()]

Figure 20. Processing event management: NEW_TASK

VirtualMachine::VmTaskStartProcessing() determines the VCPU to which assign the processing request and then:

1. alerts the VCPU to start the processing → *VirtualCpu::VcpuStartProcessing()*: the VCPU configures its internal status and the processing parameters (status flag, time of starting, computational load, task information);
2. updates the VCPU weights → *VirtualMachine::VmUpdateVcpuWeight()*: computes the new value of the weight and updates all the VCPUs through *VirtualCpu::VcpuSetWeight()*;
3. signals to the scheduler the additional VCPU that requires access to additional computing resources → *ProcessingScheduler::ProcessingEvent().*

ProcessingScheduler::ProcessingEvent() has to perform the following action in order to manage the new VCPU become active:

1. first of all, removes from *system event-queue* the event previously scheduled → *Simulator::Remove()*;
2. finds out the VCPU that is currently accessing to the computational resources (scheduled when the event *NEW_TASK* occurs) → *ProcessingScheduler::FindLastScheduledVcpu()*: the result depends on the time from the last event, the timeslice assigned to each VCPU and the current processing order;
3. updates the processing status (residual number of instructions) of the active VCPUs → *ProcessingScheduler::UpdateProcessingStatus()*: assesses the number of timeslices executed by each VCPU, that is the number of times the VCPU has been scheduled, and calls *VirtualCpu::VcpuUpdateProcessingLoad()* passing this value for updating the residual load;
4. updates the order of scheduling in the processing queue → *VirtualCpuContainer::SwapOrder()*: changes the position of the VCPUs in queue (*container*), according to the timeslices executed;
5. adds the VCPU with the new task to process in the processing queue → *VirtualCpuContainer::Add()*;

The following last three actions complete the management of this event, but are made also in case of *END_TASK* event:

6. updates the processing parameters → *ProcessingScheduler::UpdateProcessingParameter()*: computes the new values of timeslice and computation capability assigned to each VCPU and updates them by means of the dedicated *VirtualCpu* methods;
7. finds out the next event → *ProcessingScheduler::NextProcessingEvent()*: the next event (*END_TASK*) is determined by the VCPU that needs the smaller number of timeslice to complete the processing of the assigned task. The evaluation is made with parameters just updated and provides the corresponding time instant of occurrence;
8. adds the event to the *system event-queue* → *Simulator::Schedule()*: the end of processing of the VCPU pinpointed at the previous point, after the interval time computed, will be signaled by *VirtualCpu::VcpuEndProcessing()*, scheduled by the simulator.

6.3.4.2 A task completes the processing

When a VCPU completes the processing of its task, as above mentioned, the *VirtualCpu::VcpuEndProcessing()* is executed. After the reset of the status and the internal processing parameters, the VCPU has to inform both the Virtual Machine and the Scheduler about the completion of the activity, therefore, *VirtualMachine::VmTaskEndProcessing()* first and *ProcessingScheduler::ProcessingEvent()* then, are called.

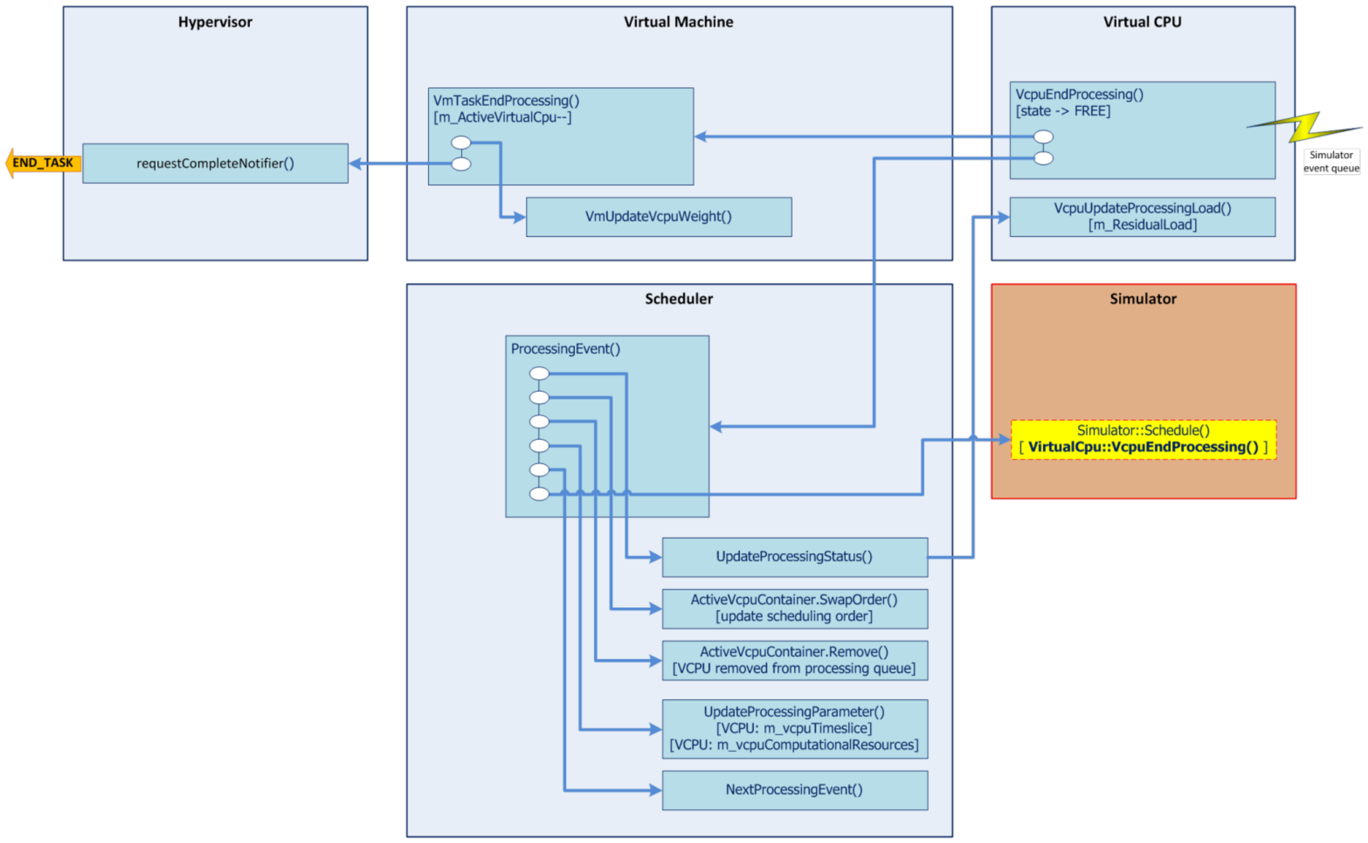


Figure 21. Processing event management: END_TASK

VirtualMachine::VmTaskEndProcessing() has to accomplish two actions for managing this event:

1. updates the weight of the VCPUs by means of *VirtualMachine::VmUpdateVcpuWeight()*, which in turn calls *VirtualCpu::VcpuSetWeight()*; this is needed to allow the Scheduler to properly find out the next processing event;
2. sends the information about the completed activity to the Hypervisor (*Hypervisor::requestCompleteNotifier()*) for the proper dispatching of the message through the communication module.

ProcessingScheduler::ProcessingEvent() manages the event of the processing completion performing the following actions:

1. updates the processing status of the active VCPUs, that is the residual number of instructions, as described at point 3 in the previous sub-section (*ProcessingScheduler::UpdateProcessingStatus()*);
2. updates the scheduling order of the VCPUs in the processing queue, also this described at point 4 (*VirtualCpuContainer::SwapOrder()*);
3. remove the VCPU that raised the event from the processing queue → *VirtualCpuContainer::Remove()*;

At this point the actions for managing the *END_TASK* event become the same of an event of type *NEW_TASK*, here only listed:

4. updates the processing parameters → *ProcessingScheduler::UpdateProcessingParameter()*;
5. finds out the next event → *ProcessingScheduler::NextProcessingEvent()*;
6. adds the event to the *system event-queue* → *Simulator::Schedule()*.

6.4 Centralized Cloud model

A Centralized (Remote) Cloud is implemented in the simulator for obtain a comparison of the offloading on the Small Cell Cloud with the services offered by a conventional Cloud Provider.

The Centralized Cloud is reached by the UEs through the EPC gateways (P-GW / S-GW) and is implemented, for simplicity, as a single node on Internet. A detailed model of the whole servers farm of the Cloud Service Provider is out of the scope of this project.

The virtual environment for the processing of an application, instead, is modeled with the same accuracy of the one built for the Small Cell Cloud, with the implementation of all the components: the Hypervisor, the Scheduler and the Virtual Machines with their Virtual CPUs. The same model is, indeed, deployed on the SCeNBce nodes and, with minor modifications, on the Centralized Cloud node. The main differences arise from the inevitably different implementation of the interface for the interaction with the other entities of the scenario, while the management of the computing resources is exactly the same in all the nodes and won't be described again in this section (refer to section 6.2 for details).

6.4.1 Centralized Cloud configuration

It's important to highlight that, despite the model is the same, a different configuration of the computing resources is used for the Remote Cloud with respect to the SCeNBce nodes.
Even though the Centralized Cloud is implemented in a single node, the higher computing resources available in this kind of cloud are taken into account. At the same time, for keeping comparable the results in all the simulations with different number of user, the resources on the Centralized Cloud are not fixed, but allocated using a scale factor proportional to the number of UEs that require access to it. This approach has been needed because the model of the Scheduler, implemented in the Hypervisor, always assigns all the available resources to the active VMs (their VCPUs).

In this way, using a scale factor, the computing resources assigned to each UE are almost the same in all the simulation configurations whit a number of UE from few units to tens. Moreover, an upper limit is used when the number of running Virtual CPUs is little, expecially at the beginning of the simulation. These mechanisms avoids the assignment of huge computing resources to the UEs in those simulations where their number is small or very poor resources when the number is very big, altering completely the results with unrealistic processing time.

Another important difference is that the processing request from a UE towards the Remote Cloud is always for the whole application: there is no optimization algorithm for partitioning the application because this task is performed by the SCM for the Small Cell Cloud, while the traditional cloud usually doesn't foresee it.

6.4.2 Remote Cloud application

The main task of the *Remote Cloud Application* is the management of the interface for the communication with the other nodes of the scenario. It is composed basically by two main objects (*RemoteCloudReception* and *RemoteCloudTransmission*) that manage, respectively, the reception socket (acting as a server) and the transmission socket (acting as a client). They interact with the *Hypevisor* when a new processing event occurs:

- a request is received from a UE: the information of the application to process is delivered to the *Hypervisor* for starting the service required;
- a processing task is completed: the *Hypervisor* requests the transmission of the data back to the UE.

At the startup the application is responsible of the establishment of the two communication socket through the usual mechanisms of *bind()* and *listen()* / *connect()*.

The following table shows the main methods of the reception part of the application (*RemoteCloudReception*)

Method	Description
Ptr<Socket> RemoteCloudReception::GetListeningSocket(void)	This set of methods are used for the basic functionality management of the socket (as server).
std::list<Ptr<Socket>> RemoteCloudReception::GetAcceptedSockets (void)	
Void RemoteCloudReception::HandlePeerClose (Ptr<Socket> socket)	
Void RemoteCloudReception::HandlePeerError (Ptr<Socket> socket)	
Void RemoteCloudReception::HandleAccept (Ptr<Socket> s, const Address& from)	
Void RemoteCloudReception::StartApplication (void)	These two methods perform all the operations needed for opening and closing the socket, at the startup and the end of the simulation.
Void RemoteCloudReception::StopApplication (void)	
Void RemoteCloudReception::HandleRead (Ptr<Socket> socket)	The method manages the incoming packets and delivers them to the Hypervisor.

Table 21. Remote Cloud Reception: class methods

The table below describes the main methods of the transmission part of the application (*RemoteCloudTransmission*)

Method	Description
void RemoteCloudTransmission::SetRemote (Address ip, uint16_t port)	The methods is used at the startup of the scocket (as client).
void RemoteCloudTransmission::StartApplication (void)	These two methods are in charge of the creation and closure of the socket, when the simulation starts and at the end.
void RemoteCloudTransmission::StopApplication (void)	
void RemoteCloudTransmission::Send (Ptr<Packet> pkt)	This method simply send on the socket the packet prepared by the Hypervisor.

Table 22. Remote Cloud Transmission: class methods

6.4.3 Remote Cloud Helper

The *Helper* of the Remote Cloud model is in charge of the installation of the applications that manage the two sockets for receiving and transmitting the messages exchanged with the UEs. The installation of processing environment is performed, instead, by the helper of the Cloud model already described in section 6.2.5.

It is split in two different components, one for each of the two sockets of the node, but both perform the basically the same setup operations:

- Configure the socket parameters;
- Install the application for the reception/transmission from/to the socket;
- Create a connection between the dual applications for letting the sharing of information
- Setup the tracing for data logging

The table below describes the main method of the *Remote Cloud Helper*.

Method	Description
void RemoteCloudReceptionHelper::SetAttribute (std::string name, const AttributeValue &value)	The methodsallow setting the parameters for opening the sockets.
void RemoteCloudTransmissionHelper::SetAttribute (std::string name, const AttributeValue &value)	
ApplicationContainer RemoteCloudReceptionHelper::Install (NodeContainer c	This methods install the applications managing the reception and transmission socketsof the node.

Method	Description
)	
ApplicationContainer RemoteCloudTransmissionHelper::Install (NodeContainer c)	
void RemoteCloudReceptionHelper::SetRemoteCloudRxName (Ptr<Node> remCloudNode, Ptr<Node> ue)	The methods are used for setting the name of the applications in the *NamesDomain* of ns-3. This is needed for the proper collection of the data.
void RemoteCloudTransmissionHelper::SetRemoteCloudTxName (Ptr<Node> remCloudNode, Ptr<Node> ue)	
void RemoteCloudReceptionHelper::EnableLatencyTracing (Ptr<Node> remCloudNode, Ptr<Node> ue)	This couple of methods sets up the "callback" for the data collection (tracing) on the Centralized Cloud.
void RemoteCloudTransmissionHelper::EnableLatencyTracing (Ptr<Node> remCloudNode, Ptr<Node> ue)	

Table 23. Remote Cloud Helper: class methods

7 PRECOMPILED EXTERNAL LIBRARIES

TROPIC has investigated a wide number of solutions for the optimization of the resources ranging from the lower layers of the LTE stack, up to the processing of applications on a virtualized environment. The algorithms proposed range from the reduction of the energy consumption or the interference mitigation, related to PHY layer parameters, up to the best assignment of the computing resources, at an abstraction level really far from the previous ones.

Many of these algorithms use PHY layer parameters that are not available in the *LTE model* of a system level simulator as *LENA-ns3*. Moreover, in some cases, also the corresponding abstraction model, required to describe the behaviour of the physical layer at a higher level, wasn't available.
This issue has been overcome by the use of an external library, properly compiled in the environment where the algorithm has been studied, and integrated inside the simulator. In this way the simulator, linking the library, is able to execute complex functions that perform the needed PHY layer estimations.

In some cases, moreover, the algorithms make use of complex mathematical tools, provided by computing environment like Matlab, that are not available in C++, the language of ns-3 framework. Also for these algorithms the creation of a precompiled library is the solution for the implementation inside the simulator.

7.1 External Libraries Manager

The introduction of an external library inside the ns-3 framework has required the development of an "object" with characteristics completely different from all the others active during the simulation, the *External Library Manager*. It doesn't interact with the other nodes exchanging messages by means of a *NetDevice*, but it acts as a collector of the information provided to and obtained from the library, making them available when necessary.

The data exchange with the simulation nodes is achieved through the *callback* mechanism, for both the input and the output data. A node, in correspondence of the proper simulation event, calls the *External Library Manager* for updating the input data for the library; it stores them inside an internal database and calls the library when the whole set of input data has been updates from all the nodes involved. The output computed by the library is stored again inside the internal database and made available to all the nodes when they ask them, always through a *callback*. At this point the simulator can use the data computed by the algorithm implemented inside the Matlab library instead of the simulated data.
The *External Library Manager*, moreover is in charge of the setup the software layer between the C++ used by the simulator and the Matlab environment, thanks to the run-time module and libraries made available by Matlab itself and linked by the compiler. In particular, the *External Library Manager* prepares the data structures needed to pass the input data to the library and receive back the results, that is the Matlab *mxArray* type (refer to the Matworks documentation for detail on this data type).

Two are the libraries integrated inside the simulator, one for the *Interference Management algorithm* and the second one for the *Joint Radio-Cloud Optimization algorithm*. They share most of the structure built by the *External Library Manager*, while specific methods have been developed for each library in order to match their input data and the interaction with the nodes involved.

This simulation object is created only if one of the two libraries has been enabled by means the corresponding configuration parameter and they cannot by executed simultaneously. At the startup, the operations performed are:

1. the creation and initialization of the internal database for storing the data exchanged with the Matlab library;
2. the initialization of the library by calling the appropriate function provided by the library itself; this action is mandatory to be able to call the functions inside the libraries.

Method	**Description**
void doInitializeSmallCell (void)	This method prepares the information objects associated to each Small Cell for allowing the access to the database.
void ComputeScUeLinkLoss (Ptr<Node> SmallCell)	The method is appointed to update the total loss of all the link among the Small Cell (in input) and all the UEs in the scenario. It makes use of the same methods utilized by the simulator at PHY layer.
std::vector<double> GetUeRbSinr (Ptr<Node> ueNode)	This method is specific for the *Interference Management* library. It is the *callback* by means the UEs can get the SINR at the reception of a sub-frame, as described in 7.1.2
void UpdateInterfManagInputData (uint16_t cellId, std::list<Ptr<LteControlMessage>> ctrlMsgList, uint8_t rbgSize)	The method is the *callback* by means the SCeNB updates the input data for the *Interference Management* library. It calls the library (when is the case) and manages the input and output data as explained in 7.1.2
uint16_t GetUeRnti (Ptr<Node> ue)	This three methods are used exclusively for the *Interference Management* library for handling the UEs through their RNTI. They are built because at the PHY layer of a Small Cell the RNTI is the only identifier available, while at the higher level of the *External Library Manager* all the nodes described by a *Node-ID*.
int16_t SearchUeFromRnti (NodeContainer cellUes, uint16_t ueRnti)	
bool CheckUeRnti (void)	
void GetJointRadioCloudOffloadingParameters (std::vector<double> *outTxPow, std::vector<double> *outCompRes, uint16_t ueCellId, double appLatency, int ueNum, std::vector<uint16_t> ueId, std::vector<double> ueTxBit, std::vector<double> ueInstrNum)	The method is the interface between the simulator and the library for the *Joint Radio-Cloud Optimization* algorithm. It called by the SCM with the *callback* mechanism and is in charge of managing the input and the output data.
std::vector<double> ComputeActiveUePathloss	This is the method used for computing the pathloss to be provided to *Joint Radio-Cloud Optimization*

Method	Description
(std::vector<uint16_t> ueId, uint16_t ueCellId)	library. It simply uses *ComputeScUeLinkLoss()* for computing the pathlosses and gets, from the array with all the UEs, only the subset of UEs with active offloading.

Table 24. External Library Manager: class methods

7.1.1 Library Database

The class has the purpose of storing and keeping update all the information about the Small Cells and their users. All the input data, needed to both the two external libraries, are saved in the database right after a Small Cell updates them and read back just before calling the library. In the same manner, the outputs produced by the algorithms are stored and made available when requested.

The database is structured as a *Container* of *Small Cell Info* objects. The former class is a typical *ns-3* object, while the latter is a specific one built for managing the information about the Small Cell. The table below describes the main method.

Method	Description
uint32_t GetN (void)	This method provides the current number of *Small Cell Info* object stored in the container
Ptr <SmallCellInfo> Get (uint32_t index)	The method gives back the *Small Cell Info* at the position indicated by index provided.
void Add (Ptr <SmallCellInfo> vcpu)	The method simply adds a new *Small Cell Info* object inside the container.
void Clear (void)	This method removes all the *Small Cell Info* objects from the container.
Ptr<SmallCellInfo> GetScInfoFromNodePtr (Ptr<Node> scNode)	These three methods allow to retrieve the *Small Cell Info* object starting from different information about the Small Cell: the node pointer, the node ID or the cell ID.
Ptr<SmallCellInfo> GetScInfoFromNodeId (uint16_t scId)	
Ptr<SmallCellInfo> GetScInfoFromCellId (uint16_t cellId)	

Table 25. External Library Database: class methods

7.1.2 Interference Management Library

The purpose of this library is to implement the algorithm, studied in WP3, for the coordination of the radio resources (in downlink) assigned by the Small Cells to their UEs at every sub-frame, in order to reduce the interference and improve the downlink rate. For this reason it is called every millisecond, when a new sub-frame has to be transmitted from a Small Cell to a UE.

The library needs from the simulator the input data described in the following table. The output is shown as well.

Name	Type	Description
inputRbMatrix	mxArray	[Input] Two-dimension vector (total number of UE x small cell bandwidth) that contains the RBs assigned to each UE by its SCeNB. In every row (a UE) the i-th element is filled with "1" only if the corresponding RB index has been assigned to such UE ("0" otherwise).
inputLossMatrix	mxArray	[Input] Two-dimension vector (total number of small cell x total number of UE) that contains the pathloss of all the UEs in the scenario, with respect to every Small Cell, not only the one to which they are attached to. In every row (a SCeNB) the i-th element is filled with the pathloss of the i-th UE towards such SCeNB, irrespective of whether the UE is attached to it or not. The order of the UE indicates the belonging to the Small cells (in each row, the first *n* UEs belongs to the first SCeNB, the second *n* UEs belongs to the second SCeNB and so on).
inputScNum	mxArray	[Input] Simple value indicating the number of SCeNB
inputUeTotNum	mxArray	[Input] Simple value indicating the total number of UE
inputScBand	mxArray	[Input] Simple value that contains the bandwidth assigned to the Small Cell (number of RBs - it is equal for all the SCeNB)
outputSinrMatrix	mxArray	[Output] Two-dimension vector (total number of UE x small cell bandwidth) that provides the SINR in each RB, computed by the library. In each row (a UE) the i-th element provides the SINR at the RB of the corresponding index, if the such RB was initially assigned to the UE and if, after the interference management, the RB has to be still used.

Table 26. Interference Management Library: class methods

In order to allow the proper operation of the library, with the appropriate data in every sub-frame, the *External Library Manager* have to create callback mechanisms with the lower layer of the LTE stack of both the Small Cell and the UE. In detail, it establishes a callback with the PHY layer of the SCeNB (in the class LteEnbPhy) for updating the input data of the library and with the PHY layer of the UE (in the class LteSpectrumPhy) for allowing the UE to get and use the output data of the library.

At the transmission of every sub-frame each Small Cell calls *UpdateInterfManagInputData()* in *External Library Manager* with the list of the Control Messages of the current sub-frame from which the active UEs and the allocated RBs can be obtained. Such information is stored inside the internal database along with the pathloss of the SCeNB with respect of all the UEs deployed (by means of *ComputeScUeLinkLoss()*). Each SCeNB that complete this task updates its status and checks the status of all the other Small Cells. Only the last SCeNB will find the task complete for all the SCeNBs and will call the library for proceding with the computing of the output data and storing them inside the database.

When a UE receives the a sub-frame, it calls *GetUeRbSinr()* in *External Library Manager* and retrieves the vector with the SINR of every RBs used for the current sub-frame.
The vector of SINR represents the point where the library enters inside the communication chain of the simulator. This vector, in fact, along with the RBs map, is given in input to the module in charge of computing the *effective SINR* and the corresponding BLER for the evaluation of the status of the reception (if the subframe has been received correctly or not). It takes into account the retransmissions of the HARQ mechanism as well, using a specific set of BLER curves when retransmissions occur, as described in [NS3-Model].
The effective SINR is used for computing the user throughput (maximum, Shannon capacity), as metric for the assessment of the performance, and the results are illustrated in section 11.3.

In order to allow a fair comparison of the performance, the library can be executed in two different ways, *"Coordinated"* and *"Uncoordinated"* allowing the collection of the statistics, in same environment, when the coordination is active and not. By means of a configuration parameter can be selected the type of execution for the library and the corresponding function is called by the *External Library Manager*. This is the only difference: all the mechanisms and data structures described above are exactly the same in the two cases.

7.1.3 Joint Radio-Cloud Optimization Library

This Matlab library allows the simulator to implement the algorithm for the optimization of the resources assigned by the Hypervisor to each UEs, that is, to their Virtual Machines, along with the power to be used for the trasmissions towards the Small Cell. The algorithm uses Matlab optimization tools not available inside the ns-3 framework and this is the reason of the use of a precompiled library.

The library is called by the SCM at every new offloading request by a UE, by means of the *External Library Manager*. In input it needs the set of parameters described in Table 27, all available at the SCM that stores the information about every SCeNBce belonging to the Small Cell Cloud; the status is always updated by each node through the dedicated status message, sent when the conditions change, following the proactive approach. In this way when the libray has to be called, the SCM has only to take the information from its database.

Once the optimization of the parameters has been performed, the SCM send the results to the Hypervisor on the SCeNBce so that the new allocation of the computing resources is used for all the UE with an application running on the Small Cell. Regarding the transmission power to be used by the UEs, the one that has sent the new request receive it directly in the answer message from the SCM, while for all the other the new value is sent by the Hypervisor inside the message with the data results at the end of the processing of a task.

Even though the mechanisms described above are completely general and can work with scenarios populated by multiplu cells, the library is built for testing the single Small Cell case.

Name	Type	Description
inputUeNum	mxArray	[Input] Single value indicating the number of active users, that is, the UEs with the application running on the SCeNB. The information is provided by the SCM.
inputBand	mxArray	[Input] Single value that contains the bandwidth of the Small Cell.
inputUeTxBit	mxArray	[Input] One-dimension vector that provides the number of bit to be still transmitted by each UE. This information, provided by the SCM, is updated considering the processing status of the application for all the UE.
inputAppLat	mxArray	[Input] Single value indicating the latency constraint of the application.
inputScCompRes	mxArray	[Input] Single value that contains the total computing resources of the SCeNBce.
inputUeInstrNum	mxArray	[Input] One-dimension vector that contains the number of instruction to be still executed for each UE. As for the bit to transmit, this information is provided by the SCM and is updated taking into account the status of the running application.
inputUeMaxTxPow	mxArray	[Input] Single value. It indicates the maximum transmission power of the UEs.
inputNoiseVar	mxArray	[Input] Single value that contains the noise power.
inputUePathloss	mxArray	[Input] One-dimension vector that provides the pathloss of all the UEs with respect to the SCeNBce.
inputBerCorr	mxArray	[Input] Single value. It contains the BER correction to be used by the algorithm.
outputUeTxPow	mxArray	[Input] One-dimension that provides the transmission power to be used by each active UE for sending to the SCeNBce the tasks to be processed.
outputUeCompRes	mxArray	[Input] One-dimension vector containing the computing resources to be assigned to the UE (its Virtual Machine) by the Hypervisor.

Table 27. Joint Radio-Cloud Optimization Library: class methods

8 SIMULATION SCENARIO

The purpose of the simulator developed in WP6 is the jointly evaluation of the radio aspects along with the cloud capabilities implemented over a cluster of the LTE Small Cells, therefore all the entities described in section 3.4 are deployed in the simulation scenario.

In particular, the radio coverage to all the UEs in the whole simulation area is guaranteed by eNBs and Small Cells (SCeNB) as in the standard LTE simulations with *LENA-ns3*. The UEs attached to this cells are not enabled for the application offloading on the SCC, but they act as radio interference sources towards the enhanced Small Cell with computing capabilities (SCeNBce) and the UEs attached to them.

From the point of view of the Cloud, all the SCeNBce belong to the same cluster, managed by a single SCM and, implemented as centralized node; the inter-cluster cooperation, with the deployment of multiple SCMs, is not addressed by the simulator. Moreover, all the UEs performing the offloading towards the Small Cell Cloud have a dedicated Virtual Machine on their serving SCeNBce.

The scenario is completed by the EPC nodes and the Remote (Centralized) Cloud on Internet, whose access is possible by means of the EPC gateways (the S-GW/P-GW node in the simulator), with the purpose of endpoint of the IP traffic for the UEs not enabled for the SCC services.

Basically, two types of scenario have been used for the evaluation of the performance, with a variable number of objects deployed:

1. multiple SCeNBce, with multiple UEs attached which execute an application with different type of processing: locally on their device and remotely with offloading towards the Small Cell Cloud and the Remote Cloud. This is the scenario mostly utilized during the simulations.
2. Single SCeNBce, with multiple UEs attached, performing only SCC offloading, used for the evaluation of the *Joint Radio-Cloud Optimization* algorithm.

They main deployement characteristics of the simualtion scenarios are summarized in the table below.

Simulation type	**Scenario setup**	
Basic configuration	• eNB • SCeNBce (outdoor) • Remote Cloud • UE @ SCeNBce type: - Local processing - Small Cell Cloud offloading - Remote Cloud offloading	
Offloading based on *Energy-Latency Tradeoff algorithm*	• SCeNBce: • UE per SCeNBce: - local processing: - Small Cell Cloud: - Remote Cloud:	4 3 to 15 1 to 3 3 to 15 3 to 5
Offloading based on *Joint Radio-Cloud Optimization algorithm*	• SCeNBce: • UE per SCeNBce: - Small Cell Cloud only	1 3 to 12
Interference Management algorithm	• SCeNBce: • UE per cell: - application processing not enabled	3 3 to 12

Table 28. Simulation scenarios

9 SIMULATION CAPABILITIES AND ASSUMPTIONS

The description of the setup process and the sequence of actions performed for building the simulation scenario, object of the previous sections, allows to understand which are the features that the simulator is able to model.

This section poses greater emphasis on this topic and summarizes the most important characteristics, assumptions and limitations and the way the main features are modelled.

- **Simulation entities in the scenario -** The number of the entities in the scenario is fixed and is established before starting the simulation. This applies not only to the number of eNBs deployed, but also to the number of UEs and SCeNB(ce)s, because the possibility of adding or removing (turning on/off) new UEs or small cells (Home eNB) at run-time is not foreseen. This is the consequence of the required setup actions performed at the beginning. Before starting, the simulator also fixes the number of each different type of entities dropped in the scenario; the corresponding parameters define:
  - the number of eNBs,
  - the number of SCeNB,
  - the number of SCeNBce,
  - the total number of simple UEs,
  - the number of UEs with SCC offloading capabilities,
  - the number of UEs with local processing,
  - the number of UEs with offloading towards the Centralized Cloud.

- **Number of UE attached to eNBs and SCeNB(ce) -** The number of UEs attached to each eNB and SCeNB(ce) is fixed and defined by a configuration parameter. In this way the simulator knows the number of UEs to be attached to each cell (eNB, SCeNB, SCeNBce), when, at the beginning, it builds the scenario. This number may change during the simulation, for some nodes, only if the handover is enabled. Regarding the SCeNBces, the portion of the total number of UEs with cloud capabilities is specified as well.

- **Connection of the UEs to their serving cell -** The UEs are attached to their serving cell, eNB or SCeNB(ce), without any negotiation phase. During the initial setup the simulator creates all the connections and configures all the parameters of the LTE-EPC model (i.e. creates the radio links and activates the EPS bearers) as if the initial operations, usually performed when a UE asks to attach to a cell, had been already accomplished.

- **SCM and cluster dimension -** The SCM is centralized, that is, the functionalities of cluster management are located in a single node, and the dimension of the cluster is fixed. The number of SCeNBces belonging to the cluster is known a priori, defined by a configuration parameter and it isn't foreseen the possibility of adding or removing a node to/from the cluster at run-time.

- **Cluster creation -** The cluster of SCeNBces is built without any initial operations performed by the SCM. As already said, the number of nodes belonging to the cluster is fixed and this allows building the cluster before starting the simulation, through the creation and configuration of all the communication links among SCeNBces and SCM.

  The SCM doesn't perform any setup operation or cluster selection and, at the start of the simulation, it already has all the information about the physical resources of the SCeNBces and their UEs (those enabled for accessing to the service). This also is a consequence of the several configuration actions that the simulator needs to perform before running.

- **Virtualization infrastructure on the SCeNBce** - The simulator prepares the virtualized environment in every SCeNBce node of the cluster during the initial configuration, therefore no installation operation is simulated. When a simulation starts, all the SCeNBces have a Hypervisor active, with a working Scheduler, and are ready for managing the offloading requests from their UEs.

- **Virtual Machine deployment** - The Virtual Machines are deployed on the SCeNBce nodes during the preliminary setup. Thanks to the fact that the number of UEs of the SCeNBce is fixed, the simulator can deploy the proper number of Virtual Machines.

  The SCM doesn't perform the process of Virtual Machines creation and deployment, but when the simulation starts, all the SCeNBce nodes already have a number of working Virtual Machine equal to the number of their UEs with SCC capabilities.
  For each UE, the corresponding primary Virtual Machine is deployed exclusively on its serving cell (see D42 and D52), while a secondary Virtual Machine is not deployed and therefore, is not involved in the remote processing.

- **Admission control** - The admission control is not implemented. It is supposed that the UEs with cloud capabilities have been already admitted to the Cloud services. When the simulation starts the SCM already knows the group of users that are enabled to process their application on a remote node.

- **Offloading** - Two are the algorithms used in the offlloading decision: the *Energy-Latency Tradeoff* and the *Joint Radio-Cloud Optimization* (see D311 and D51). In both the cases the decision is taken by the SCM, but using a different set of information received by the UE and SCeNBce. For the first one the following information are used:
  - estimations of the time for local processing;
  - estimations of the energy spent for data upload and download;
  - estimations of the time for remote processing.

  Such estimations are produced for a predefined set of possible processing configurations (application catalogue) and among them the SCM evaluates the optimal configuration, that is, type of tasks execution, partition between local and remote processing and transmission power.

  In the second algorithm the information are the following:
  - radio and computing resources available;
  - channel condition (pathloss only) of all the UEs with active offloading;
  - application status (bit to be transmitted and instruction to be precessed) of all the UEs with active offloading;

  The algorithm produces the optimal values for the transmission power and the computing resources for every UE.

- **Remote processing on a Virtual Machine** - The tasks, the computing unit of an application, may be executed in serial or parallel manner, depending on the optimal configuration selected by the offloading algorithm. Below some implementation details according to the simulation model described in D51:
  - the processing time of a task is determined by the number of instructions to be processed and the computational resources assigned by the Scheduler to the VCPU in charge of its processing;
  - every task is assigned to only one VCPU: the parallel processing refers to the simultaneous execution of two tasks on two distinct VCPU, not the splitting of a task on two VCPUs;

  - every VCPU processes only one task a time: the assignment of a VCPU is intended in an exclusive way, that is, a VCPU cannot process two different tasks, even if belonging to the same application;
  - the processing of a task begins with a message sent from the UE and a message is sent back from the SCeNBce at the end: only when a task is ended a new one can be started. The messages exchanged contains information about the task and the data to be processed.

- **Local processing on a user device -** The processing of the portion of application to be executed locally is still modeled by means tasks as basic computation unit, but in this case no scheduling mechanism is implemented. The processing time is evaluated in the same manner as the remote processing, as ratio of the number of instructions to process and the computation power.

- **Security -** The absence of initial negotiation and admission control for the access to the cloud services implies that the security algorithms are not implemented and security attacks are not simulated and evaluated. The secured connections (tunneling) among the nodes are not implemented because in this scenario they would have been modeled simply as an overhead in the header of the packets exchanged.

- **Centralized Cloud -** The conventional Cloud is modelled has single node without a detailed implementation of the server farm of the Provider. In this node is “installed” exactly the same *Cloud model* of the SCeNBce, therefore all the assumptions listed for the SCC are still valid in this case.

10 SIMULATION SETUP

The simulator, before starting the execution, needs to “build” the scenario. All the entities previously described are created and configured at the beginning of the main script, then the communication links are established for each couple of nodes between which there is an exchange of packets.

In detail, the complete setup of a Node implies, from the point of view of the communications, the creation and the configuration of two key abstractions: the *NetDevice* and the *Channel*. In the real world, these terms correspond roughly to peripheral cards and network cables (talking about cabled networks) and, therefore, they completely define the characteristics of the channel between two nodes, that is, the model used to simulate the medium. For this reason, every communication interface in a Node has to be specialized for the type of medium used.

This is the setup of a *ns-3* simple network simulation. The *LTE-EPC model* has to define and arrange many other elements, in order to create the complex simulation structure of the LTE standard as illustrated in the following sub-sections. Moreover, the *Cloud model*, as well, has to be configured to create the simulation objects of the Cloud side of TROPIC.

10.1 Setup of the LTE model

The process of setting up the *LTE model* typically involves the following main steps:

1. create Node objects for the eNBs, SCeNB(ce)s and the UEs;
2. install the *NetDevice(s)* and define the corresponding channel(s);
3. configure the spatial characteristics, that is, position and mobility (if applicable);
4. install and configure the LTE protocol stack;
5. install and configure the IP protocol stack;
6. attach the UEs to an eNB or SCeNB(ce) and create an RRC connection between them;
7. install an Application for traffic generation on the UEs.

Furthermore, the *LTE helper* allows to enable, as optional feature, the handover during the simulation. If the handover has to be simulated, the setup process accomplishes the additional step of adding the X2 interface to the eNBs. Through this interface the cells exchange the information required to complete the operation and it is not created if the handover is not simulated. Only the creation of the X2 interface is left to the *LTE model*, the handover during the simulation is managed by *EPC model*.

The configuration of the LTE stack (point 4) by means of the *LTE Helper*, allows the selection, among many other parameters, the following simulation characteristics:

a. Transmission Mode: the LENA model doesn’t implement all the Transmission Modes of the LTE standard. Among the available modes there is MIMO, the one used for the TROPIC simulations. It should be noted that the characteristic of having multiple antennas at transmitter and receiver side is modeled simply as the gain, at PHY layer, that MIMO schemes bring in the system from a statistical point of view, with respect to the SISO case with no correlation between the antennas.

b. AMC algorithm: the LENA model provides two different algorithms for Adaptive Modulation and Coding; without entering in the implementation details, the one based on the physical error model is the most appropriate for TROPIC. See the official documentation for further information.

c. MAC scheduler: the LTE model implements two scheduler algorithms, the *Round Robin* and the *Proportional Fair*. The default *Round Robin* is used for the project simulations in absence of specific requirements.

Finally, for a more complete overview of the *LTE model*, it should be said that it considers the FDD only, and implements downlink and uplink propagation separately.

The above are the operations for setting up standard *LENA-ns3* LTE simulations. For TROPIC simulations, additional steps are required in order to enable the new characteristics implemented.

- The module for the supervision of the Cloud related operations is installed on the UEs with access to the Cloud services and on the SCeNBces.
- The Offloading module and processing model is installed in the UEs: these are specific features of the UEs that use the Cloud services (attached to SCeNBces).
- The battery consumption model is installed in the UE: the model provides the information about the impact of radio communications and local execution of an application on the battery of the device.

It should be explicitly said that the *LTE helper* is also in charge of the creation (not the complete setup) of the nodes belonging to Cloud model, the SCeNBces and SCM, as explained in section 10.3.

Regarding the Application (point 7) the default one (client/server for packet exchange) is enabled in the UEs attached to eNBs and SCeNB, with the purpose of IP traffic generation. For the UEs with Cloud services enabled, instead, a special Application has been developed in order to manage the offloading request and the processing performed locally and remotely, as illustrated in section 4.2.3.1.

10.2 Setup of the EPC model

The LENA module provides the EPC entities in a separate model because they don't need to be necessarily deployed in a LTE simulation, as for instance, in simulations focused on the radio aspects. The *EPC model*, therefore, has to be enabled and configured explicitly by means of its *helper*.

The boundary between LTE and EPC models, however, is not well-outlined, therefore some of the items needed by the *EPC model* have to be configured by the *LTE helper*. Only whether the EPC is enabled, for instance, the *LTE helper* takes care of the installation and configuration of IP device and stack, used for the communication of the LTE nodes with the entities of the EPC network.

The creation and configuration of the *EPC model* entities is different with respect to the deployment described in the previous sub-section because of its great complexity and its criticality on the proper execution of the simulation. Only a few optional configurations are left to the users, while the *EPC helper* takes care of creation and configuration of all the EPC nodes and the interfaces among them.
The simple instantiation of the *EPC helper* automatically performs the following operations:

1. create and configure the P-GW/S-GW node,
2. create and configure the MME node,
3. connect the eNB/SCeNB(ce) to the Core Network through the S1 interface,
4. create the S11 interfaces among the S-GW and MME (the S5/S8 interface is not modeled because P-GW and S-GW are modeled as a single node),
5. activate the default EPS bearers

The setup of the *EPC model* is completed by a few manual, and in some cases optional, operations:

1. configuration of the UEs for IP networking: assignment of the address and routing configuration. The only Packet Data Network (PDN) type supported is IPv4 (the IPv6 type is implemented in ns-3, but it hasn't been integrated by LENA in their module).
2. creation and configuration of one or more Remote Host: this allows the simulation of the entire communication chain, from a UE to Internet (and vice-versa);

3. activation of dedicated EPS bearer: a dedicated bearer is activated for the UE with Cloud capabilities, used for the Cloud related communication.

10.3 Setup of the Cloud model

The nodes belonging to the *Cloud model*, the SCM and the SCeNBces, are not deployed by means of specific “tools” of its *helper* but they are created and configured using the helper of the *LTE-EPC model*. These node objects have, in fact, the same characteristics of all the other simulation entities and, therefore, their setup doesn’t require additional operations with respect to those performed using the existing helper. This is the same approach used by the LTE and EPC model: some items are created and configured in the *LTE model* even if they belong to the *EPC model*.

The deployment of the SCM and the SCeNBces, performed by the *LTE-EPC helper*, implies the following operations:

1. create the SCM with the networking capabilities (device and stack);
2. install an additional IP *NetDevice* (with the IP stack) on the SCeNBces for the communication with the SCM;
3. create a P2P link among the SCM and all the SCeNBce of the cluster;
4. configure all the IP *NetDevice* with address and routing information.

Once the deployment of cluster and its manager is done, the setup of the *Cloud model* can be completed by the specific helper. The *Cloud helper* is in charge of creating the virtualized computing platform on all the small cells that make up the cluster (the SCeNBce); in detail, on each of them:

1. create the Hypervisor;
2. create the Scheduler;
3. create a Virtual Machine for each UE with cloud capabilities attached to the small cell. Every new Virtual Machine, in turn, create its Virtual CPUs.

Regarding the *Centralized Cloud*, the setup is not different from the SCeNBce node described above. First of all, the node, with its *NetDevice*, is created and configurared for allowing the communication with the other nodes (those involved). The operations are performed mainly using the *EPC Helper* because it is reachable only through the EPC gateway, that is the P-GW/S-GW node.

After the initial setup, the node is completed with the “installation” of the *Cloud model* by means of its *helper* that builds the full virtual environment with exactly the same elements of the SCeNBce. Only the configuration parameters are different because the *Remote Cloud* uses its own specific application for the management of the communications and it has to be enabled.

10.4 Additional models

The previous sections have been dedicated to the description of the most important models. The full setup of a simulation implies the configuration of many other aspects for which *ns-3* offers several different models. The most important for the project purposes are the following:

1. Channel and propagation: these models provide path loss and fading used for the simulation of the radio channel among the eNB, the SCeNB(ce) and the UE. *LENA-ns3* makes available several channel models, most of which implements the ITU and 3GPP models, therefore they are perfectly aligned with the requirements of TROPIC.

2. Antenna: the models allow to define the proper radio and geometric characteristics of the antenna used by eNB and SCeNB. The available models (*isotropic antenna*, *cosine antenna* and *parabolic antenna*) are enough for the purposes of the project.

It is to be noted that the simulator *LENA-ns3* creates a multi-sector site by means of different nodes placed in same position, with the appropriate configuration of the orientation and beam width.

3. Building: the model is used to create buildings inside the scenario, configuring their geometry (room size, number of rooms and floors) and position, but also the characteristics of internal and external walls because this impacts on the propagation model. The Building model, in fact, works closely with the Mobility model and Propagation model in order to manage the UEs positions and correctly take into account the indoor-outdoor communications.
   The corresponding *helper* allows to shape complex scenarios with the definition of each single buildings or the use of default building blocks, like the *dual stripe model*, frequently used in this kind of simulations [R4-092042].

4. Mobility: define the way the UEs move inside the scenario. The model provides appropriate tools for controlling the position of the UEs in specific area, like inside a building, in order to avoid the continuous movement from outside to inside and vice-versa.

10.5 Simulation parameters

The simulation entities and the models illustrated in the previous sections use a very large number of parameters for the complete definition of their characteristics and behaviour. The whole set of configuration parameters may be divided in the following main classes:

- Geometry parameters: they define size and position of the simulation objects; thanks to these parameters the simulator builds the scenario by placing, for instance, buildings with specified dimensions and characteristics and antennas with the proper inter-site distance. The definition of this set of parameters allows running simulations with the same scenario deployment.

- Population parameters: they define the number of the simulation objects placed in the scenario (UE, eNB, SCeNB, SCeNBce) and their distribution, like the density of indoor Small cells or the number of UE attached to the cells, for every kind of cell.

- Radio parameters: these are, basically, the parameters used for the configuration of the LTE model. They span from the transmission power, to the frequency and band or the scheduler algorithm at the LTE MAC.

- Cloud parameters: they define the characteristics of the entities of the cloud model in term of computational resources, both physical and virtual, like the number of CPU and the speed but also the number of Virtual CPU of the Virtual Machine or the *weight*, a parameter of the Virtualized SCeNBce model (D51), that impacts on the processing scheduling and the allocation of the computation resources.

- Network parameters: these parameters are used for the configuration of the *EPC model*, but also to define the characteristics of the links among the nodes belonging to cluster and the SCM. In this class, in fact, are included the parameters of the backhaul model.

- Application parameters: they define the model of the applications running on the Virtual Machine (but also the local part on the UE), like the code length, the number of instructions, the data input and output of each tasks in which the application is split.

Each of the above macro-classes encompasses a lot of simulation models, each of which is characterized by a wide number of parameters, for a huge total number of possible configuration options of the simulator. Most of them have been left to their default value, expecially in the LTE and EPC models. The configuration parameters mainly changed during the assessment of the performance have been those of the specific models introduced for TROPIC and are described in the results section.

11 SIMULATION RESULTS

The simulator allows the assessment of the overall process of offloading from the UE towards the SCC with the modelling of the radio communication between UE and Small Cell, with the whole LTE-EPC infrastructure, and the processing of the application performed on the virtual environment built over the SCeNBce. For the communication part of the this process there hasn't been the need to modify the default setting used by *LENA* module for the multitude of configuration parameters.
As regards the Cloud model the main parameters are the computing resources and the characteristics of the application to be executed.

Table 29 shows the characteristics of the application used in the simulations of the following subsections, while Table 30 describes the physical resources of UE and SCeNBce.
Regarding the computing resources of the user device, the implemented UE model makes available for the local processing of an application only the 20% (with a small random oscillation around this value) of the total amount, in order to take into account the OS overhead and the background load due to other active applications (as described in the task graph application model).

Application parameter	Value	Unit	Note
Task number	8		
Instruction Number (each task)	15	MI	
Data In (each task)	26	KB	Size of the data trasmitted
Data Out (each task)	5	KB	Size of the data received
Maximum latency	1.5	s	Total latency constraint for the application

Table 29. Application parameters

Physical resources	Value	Unit	Note
UE			
Core number	2		
Core speed	500	MIPS	Only a percentage of total resources are available for processing the application (to take into account all the functionalities active in a device)
SCeNBce			
Core number	4		
Core speed	1000	MIPS	

Table 30. Computing physical resources

Finally, as explained in section 6.4.1, the physical resources on the Remote Cloud cannot be keept constant for avoiding the allocation to the VCPUs an amount of resources too high when the number of UEs is small and very low computing resources when the number of UEs is big. Therefore, in order to have a fair configuration in all the simulations, a scale factor is used on the Centralized Cloud with respect to the SCeNBce.

11.1 *Energy-Latency Tradeoff algorithm*

The results shown in the following subsections have been obtained using the *Energy-Latency Tradeoff algorithm* in the offloading decision, in a set of simulations where the number of UEs per Small Cell is gradually increased. For a full comparison, in each SCeNBce a fraction of the UEs are configured for both local processing and offloading towards the Centralized Cloud.

It's worth to remember here that the *Energy-Latency Tradeoff algorithm* is based on data-partition. In order to allow a more effective analysis of the results, the "catalogue" of the possibile configurations for the partition of the UE application has been reduced with respect to the initial list foreseen in [D51]. The possible configurations among which the SCM can choose the best option in each offloading request are described in the table below. The main simplifications can be summarized in having reduced the data partition percentages and having neglected the configurations without parallel computing.

Configuration index	UE percentage	SCC percentage	Type of processing
0	100 %	0%	parallel
1	75%	25%	parallel
2	50%	50%	parallel
3	25%	75%	parallel
4	0%	100%	parallel

Table 31. Application partition configurations

This set of configurations has been established after numerous simulations performed during the development, with a wide range of different setups, having noticed that:

- a finer step in the partition percentage doesn't produce significant differences in the estimated parameters used in the offloading decision, but the wider number of them makes less easy the understanding of the overall behavior;
- the options where the parallel processing of the tasks is not enabled are never selected by the SCM because, as obvious, they produce always the worst estimations of the overall time. Moreover, considering the model of the processing, the test case without parallel computing can be simply obtained by doubling the number of instruction of each task, because the unique impact is on the processing time.

The UEs that performe offloading towards the Remote Cloud, instead, always send the whole task, that is, the configuration 4 is always applied, because the SCM is not involved in the processing on the conventional Clouds.

11.1.1 Mixed application processing: 8 UEs per SCeNBce

In this setup, each Small Cell has a small number of UEs and only half of the total are active with offloading requests towards the SCeNBce. The latency constraint can be met and thanks to the computing available on the Hypervisor, in 3 requests out of 4 the applications is processed totally on the SCC as shown in Figure 24.

The energy spent by the UE with the processing on the SCC is (in most of the cases) lower than UEs that process the application locally and is comparable with those spent for the processing on the Centralized Clod (Figure 22). This latter result is due to the basically the same type of activity for this two type of UEs, considering what said about the offloading configuration. The higher latency of the Remote Cloud UEs (Figure 23), instead, can be explained with the additional delay due to the backhaul.

Processing type	UE number
Local processing	1
Remote Cloud offloading	3
Small Cell Cloud offloading	4

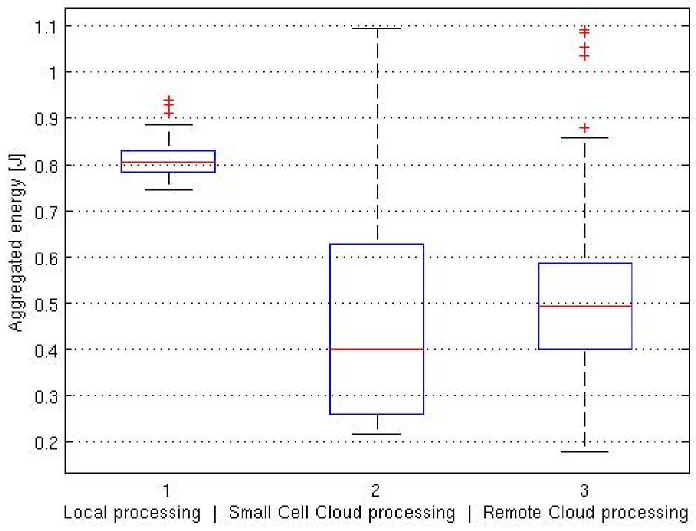


Figure 22. UE energy consumption (case 8 UE per SCeNBce)

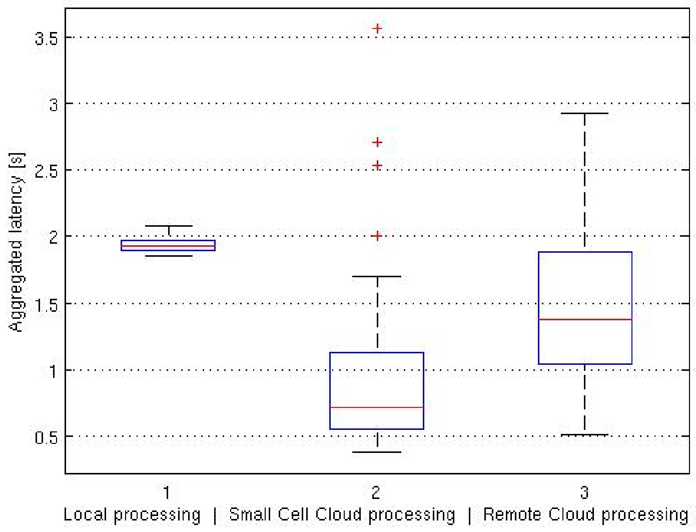


Figure 23. UE application latency (case 8 UE per SCeNBce)

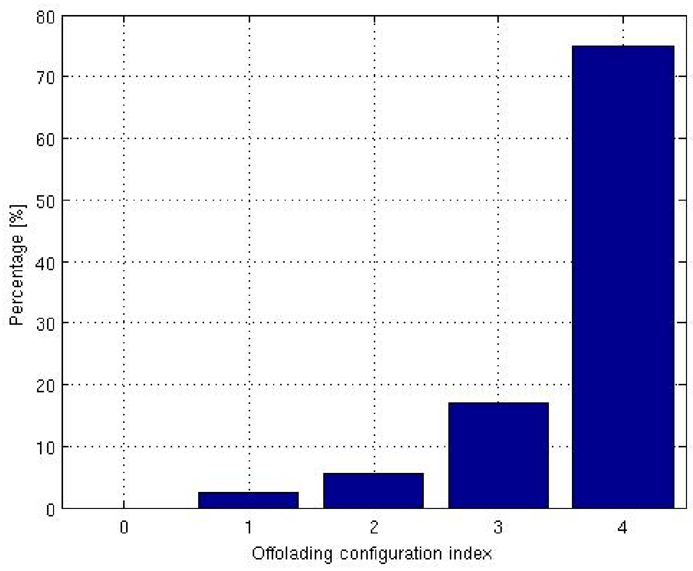


Figure 24. Application configuration selected by the SCM (case 8 UE per SCeNBce)

11.1.2 Mixed application processing: 12 UEs per SCeNBce

Increasing the number of UEs of the Small Cell, and the corresponding number of them that requires processing on the SCC, the offlading configuration changes in favour of a lower percentage of remote processing. This produce an increase of both the latency and the energy spent by the UE, while continuing to be, for both, below those of the local processing.

Processing type	UE number
Local processing	2
Remote Cloud offloading	4
Small Cell Cloud offloading	6

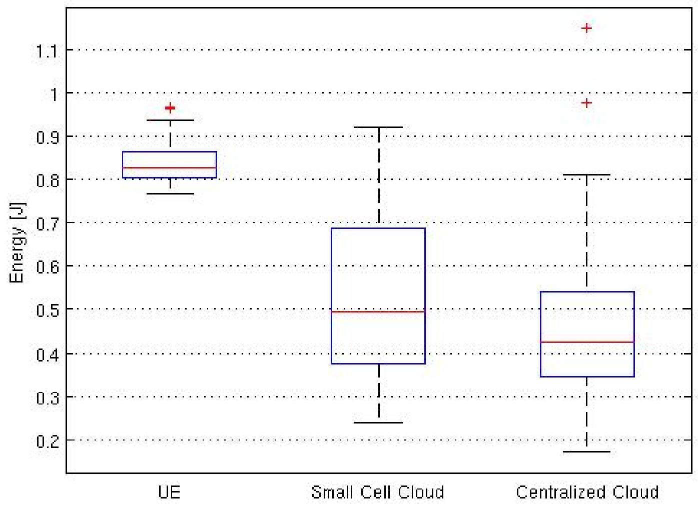


Figure 25. UE energy consumption (case 12 UE per SCeNBce)

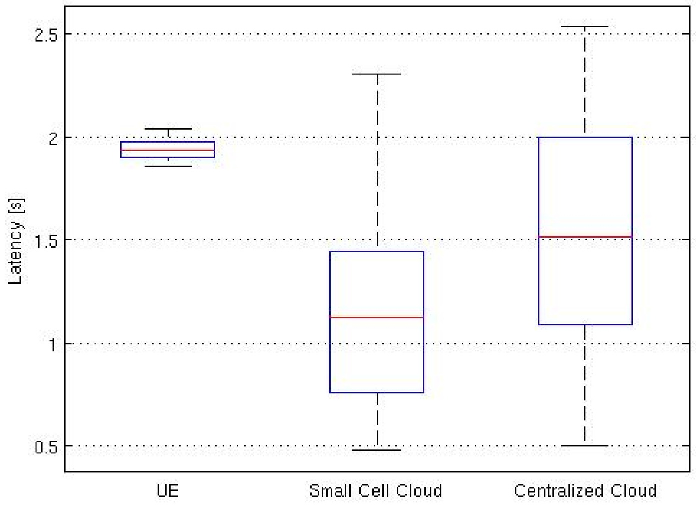


Figure 26. UE application latency (case 12 UE per SCeNBce)

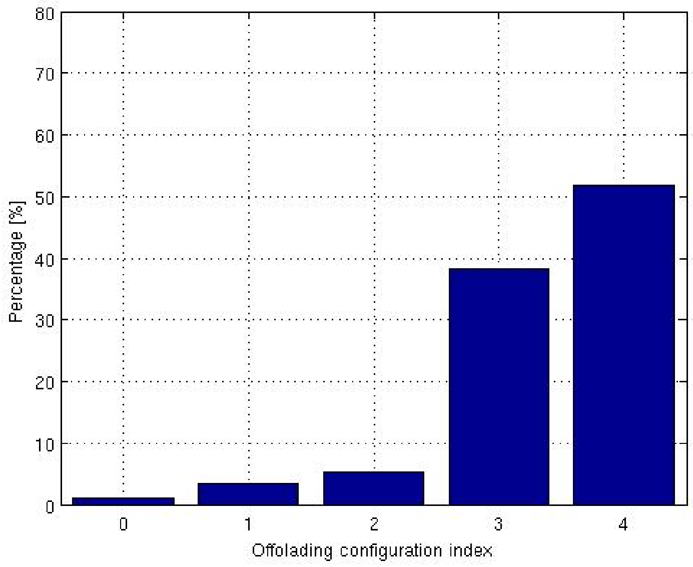


Figure 27. Application configuration selected by the SCM (case 12 UE per SCeNBce)

11.1.3 Mixed application processing: 16 UEs per SCeNBce

The trend shown in the previous test case continues with the further increasing the number of UEs, although this is less evident from the figures of latency and energy, but is confirmed by Figure 30. In The latency constraint for the application can be still met.

Processing type	UE number
Local processing	3
Remote Cloud offloading	5
Small Cell Cloud offloading	8

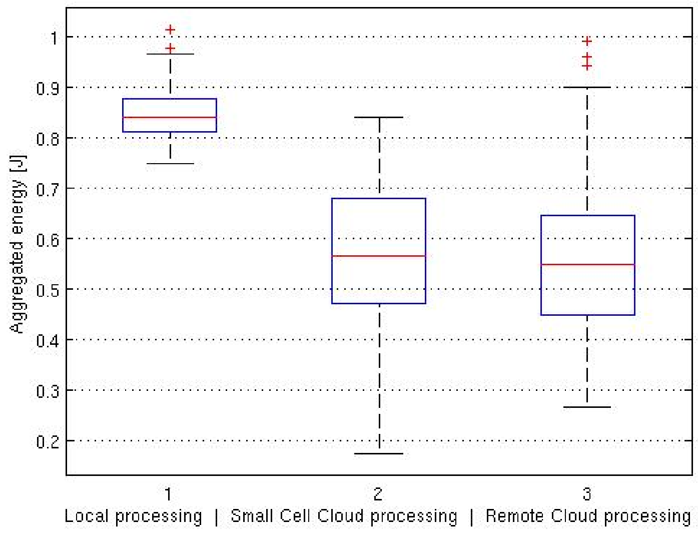


Figure 28. UE energy consumption (case 16 UE per SCeNBce)

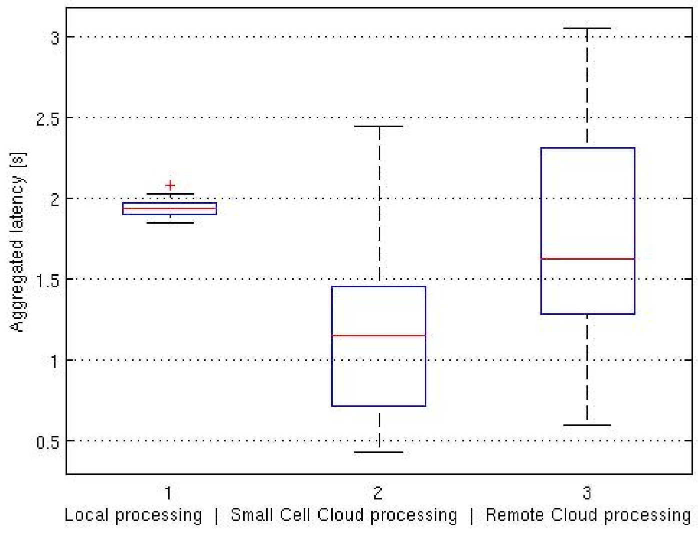


Figure 29. UE application latency (case 16 UE per SCeNBce)

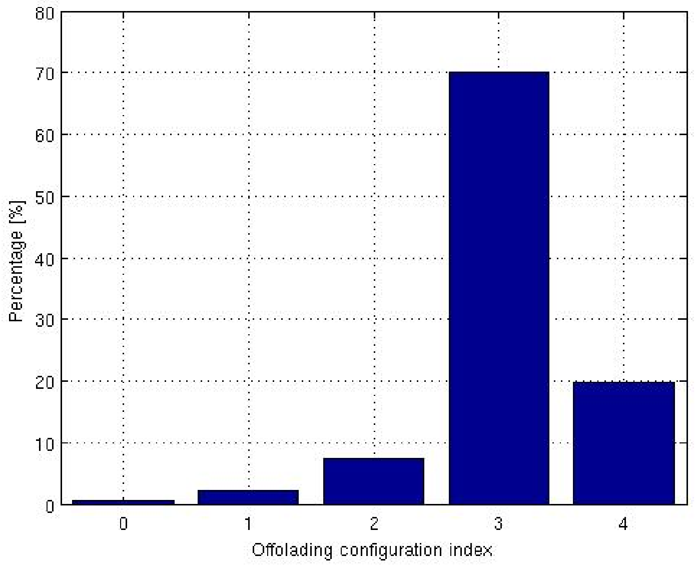


Figure 30. Application configuration selected by the SCM (case 16 UE per SCeNBce)

11.1.4 Small Cell Cloud offloading only

For a deeper analysis of the system behavior, a further set of simulations has been executed, focusing solely on the offloading towards the Small Cell Cloud, again increasing the number of UEs attached to the SCeNBce. In these simulations therefore, all the UEs have always requested the offloading to their Small Cell and their number ranges from 3 to 15, against a maximum number of 8 UEs (SCC offloading) reached in the test cases described in the previous sections.

Figure 32 shows how the latency constraint for the application cannot be satisfied, in the current setup, when the number of a UE accessing to the SCC service becomes greater than 9, considering that the processing time is larger when performed on the local device (as shown in the test cases above). From the point of view of the energy, instead, in most of the scenarios the performances continue to be better with respect to execute the application locally, as shown in Figure 31.

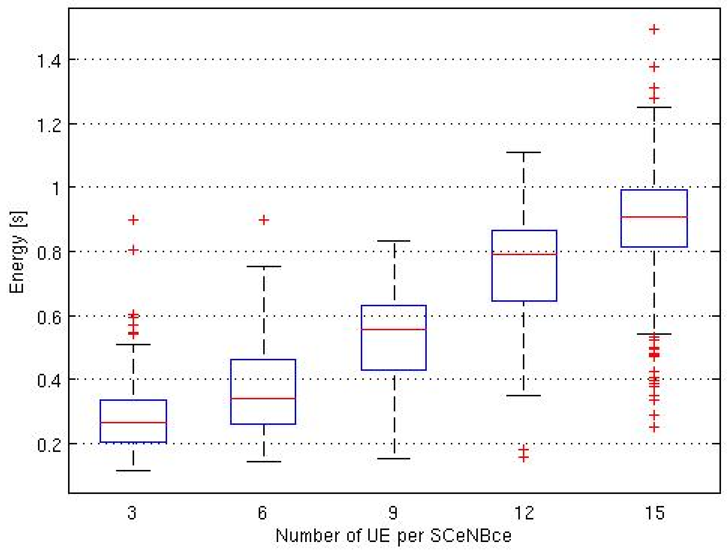


Figure 31. Energy consumption for different number of UE per SCeNBce (ELT algorithm)

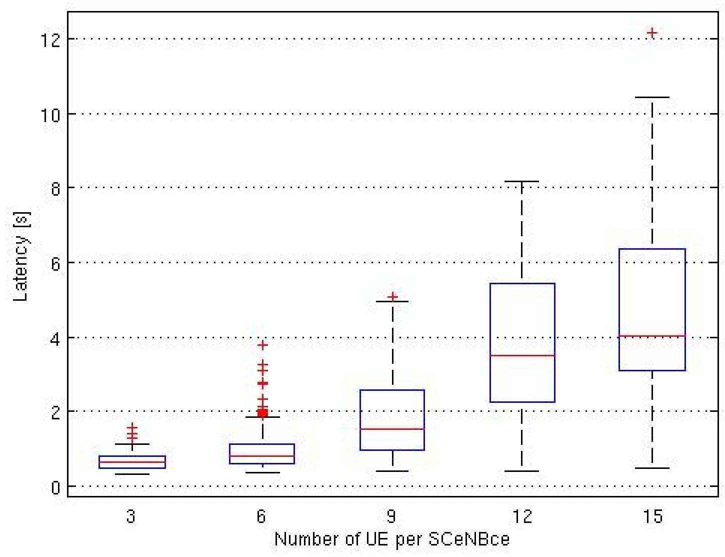


Figure 32. Application latency for different number of UE per SCeNBce (ELT algorithm)

Figure 33 shows how the partition of the application, between local and remote processing, is distributed when the SCeNBce gets more populated. In this set of simulations, again, the configuration selected by the SCM tends to shift from the configuration with high percentage of processing on the SCC (number 4) towards those in which this percentage is lower. This trend in the partion of the application is coherent with the behavoiur observed for the energy spent (Figure 31): the more the number of UEs increases, the higher is the percentage of application running locally and therefore the energy required.

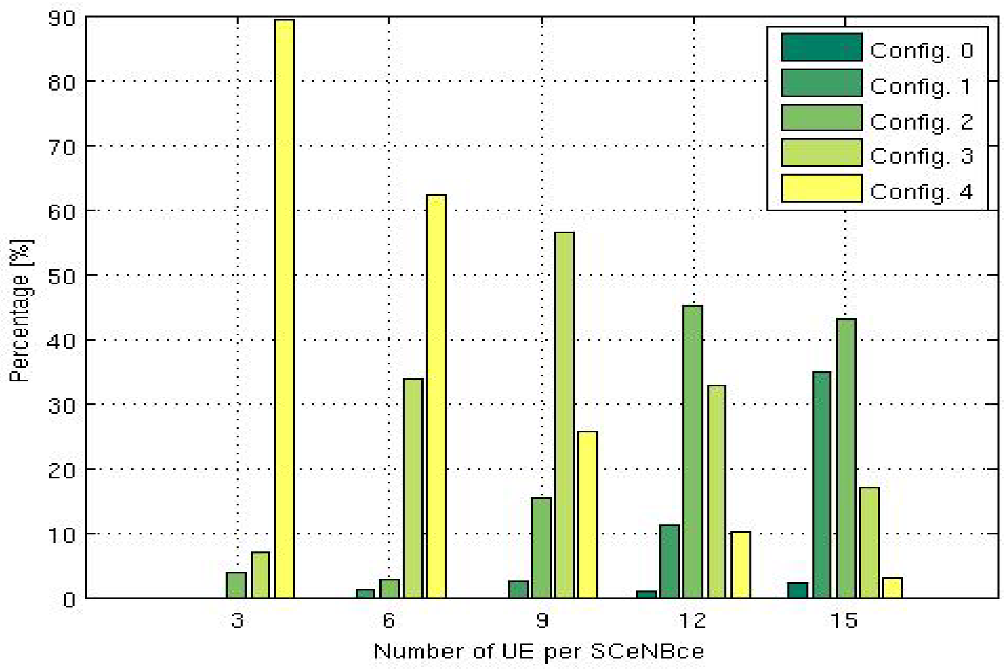


Figure 33. Application configuration selected by the SCM for different number of UE per SCeNBce

11.2 Joint Radio-Cloud optimization Algorithm

In this section the results are obtained using a different algorithm for the offloading with respect to the previous sections, the The *Joint Radio-Cloud Optimization*, implemented by means an external library as described in 7.1.3.

It's important to highlight the following main differences between the two offloading algorithms.

- The *Energy-Latency Tradeoff* algorithm provides the optimal partition for the application between local processing and processing on the SCC, not the second one. For the comparison of the results, with the *Joint Radio-Cloud Optimization* algorithm the partition configuration number 3 (Table 31) has been used. The choice has been based on the results of the previous set of simulations that shows how this configuration seems to be suitable in most of the test cases (Figure 33).
- The *Joint Radio-Cloud Optimization* algorithm provides the amount of computing resources to be allocated by the the Hypervisor to all the active UEs, while with the *Energy-Latency Tradeoff* algorithm this task is left to the Scheduler on the SCeNBce. However, the status of the computing resources on the SCeNBce is taken into account during the decision in the *Energy-Latency Tradeoff* algorithm as well, thanks to the estimations about the processing provided by the Hypervisor.
- Both the algorithms are executed every time the SCM receive a new offloading request from a UE, but while the *Joint Radio-Cloud Optimization* algorithm updates the computing resources and the transmission power for all the UEs, with the other algorithm the transmission power of the UEs already active (with an application already offloaded) never change until the application processing is completed. The computing resources are, instead, always updated by the Scheduler depending on the number of running Virtual Machines.

The configuration of the application and the physical resources of UEs and Small Cells used in this set of simulation is the same of the previous, described in Table 29 and Table 30. Local processing and offloading towards the Remote Cloud have not been envisaged in these simulations because the results for these kind of UEs don't change with the algorithm used for the offloading and they have been already evaluated in the previous test cases.

Observing the results for energy (Figure 34) and latency (Figure 35) can be noticed that:

- The latency constraint can be sasfied only until the number of UEs per SCeNBce is less than about ten, as with the *Energy-Latency Tradeoff* algorithm. This is the limit imposed by the resources available.
- With the *Joint Radio-Cloud Optimization* algorithm, however, both the energy and the latency show less dependency from the number of UEs, at least until the constraint on the latency can be met (Figure 36).

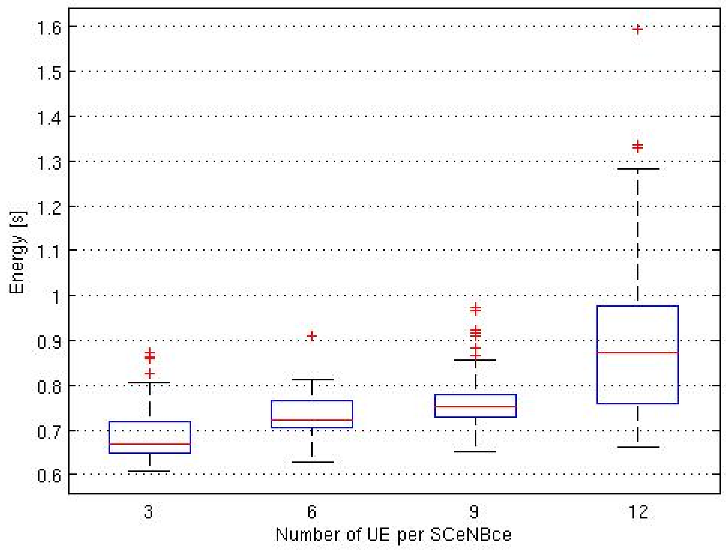


Figure 34. Energy consumption for different number of UE per SCeNBce (JRC algorithm)

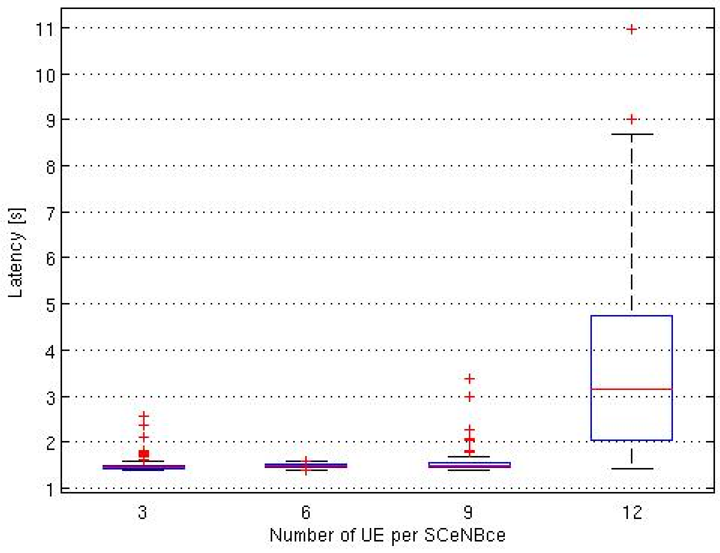


Figure 35. Application latency for different number of UE per SCeNBce (JRC algorithm)

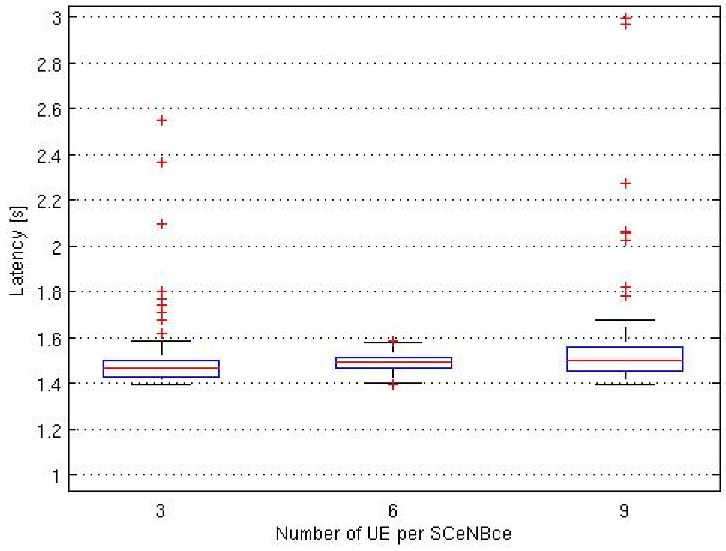


Figure 36. Application latency (detail)

The following three figure (Figure 37 to Figure 39) illustrate the distribution of the computing resources assigned to the active Virtual Machines (therefore to each UE), when the two offloading algorithms, *Energy-Latency Tradeoff* and *Joint Radio-Cloud Optimization*, are used by the SCM for chosing the set of parameters for the offloading.

With both the algorithms, as the number of UEs increases, obviously the computing resources assigned to each Virtual Machine decreases, but the *joint Radio-Cloud* algorithm keeps a smoother distribution with respect to the *Energy-Latency Tradeoff*.

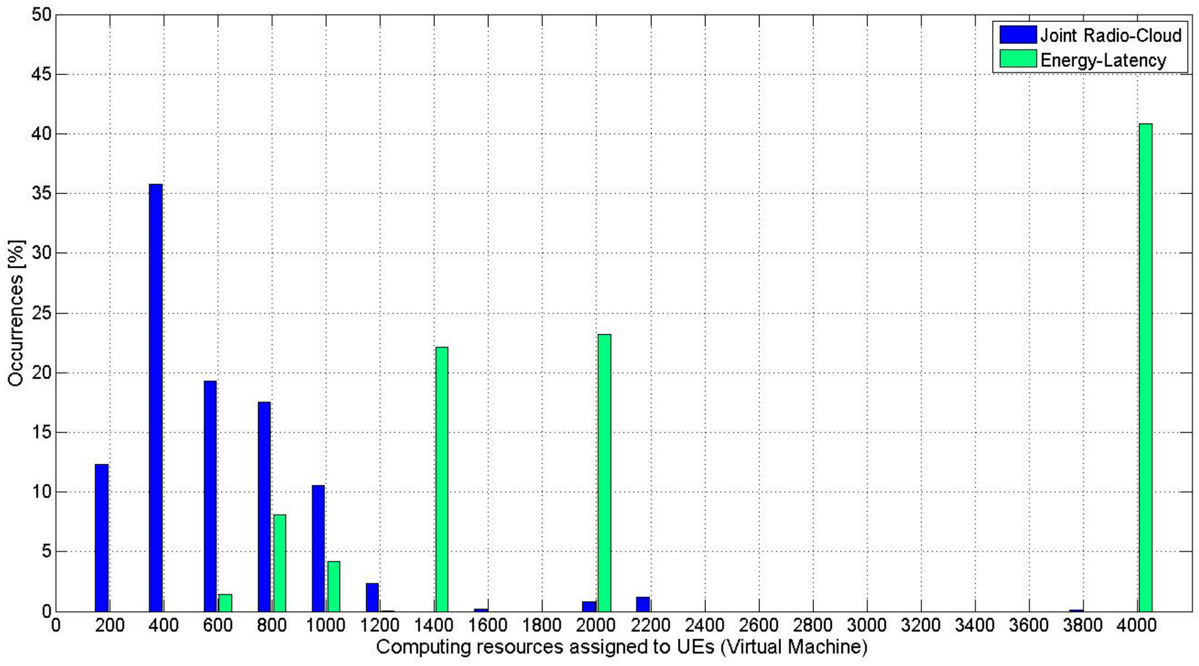


Figure 37. Computing resources distribution (case 6 UE per SCeNBce)

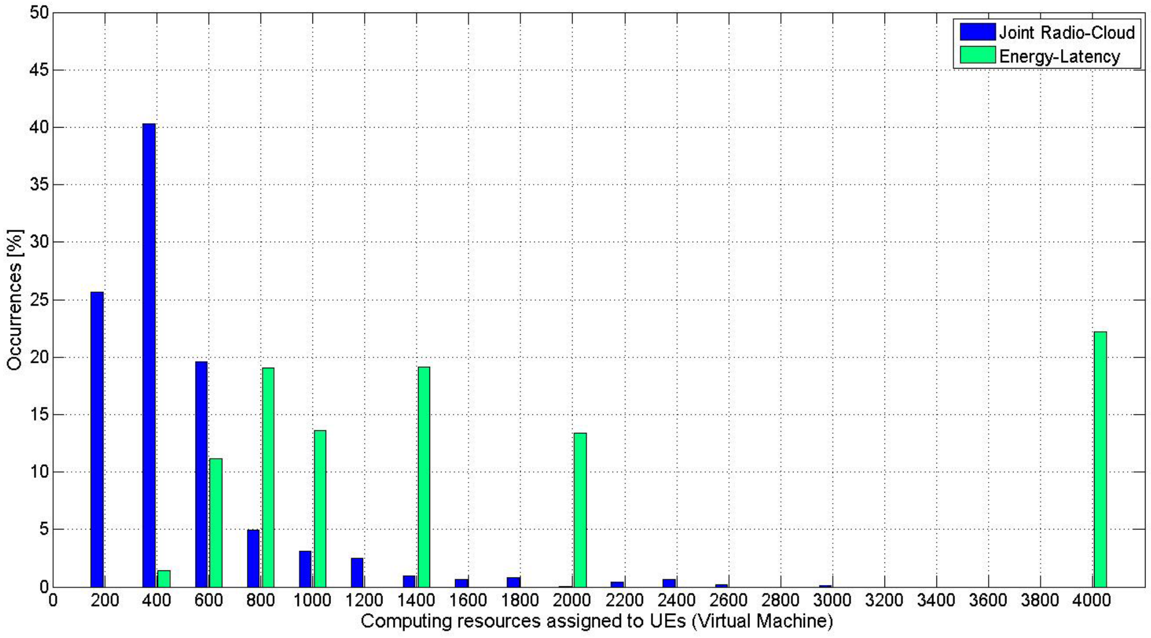


Figure 38. Computing resources distribution (case 9 UE per SCeNBce)

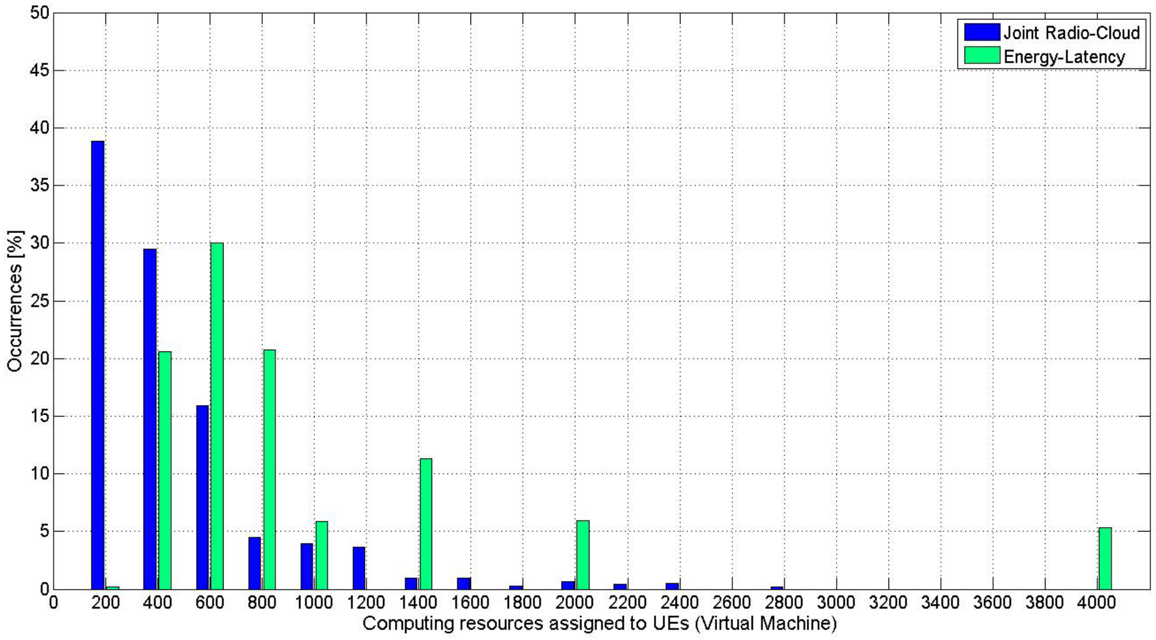


Figure 39. Computing resources distribution (case 12 UE per SCeNBce)

The latter shows a highly discontinuous distribution, that comes from the resources allocation implemented by the Scheduler: it assigns the whole computing resources splitting them among the running Virtual CPUs, taking into account their *weight*. This leads to obtain a behavior that reflects the distribution of the number of UEs simultaneously active with the offloading.

Moreover, with the *Joint Radio-Cloud Optimization* algorithm the assignement of the entire resources to a single Virtual Machine is an event very rare.

11.3 Interference Management Algorithm

In this section the performance of the MP2MP interference management procedure based on CoMP-BF with coordinated sounding developed in 3A2 is evaluated. The interference management procedure is detailed in Section 5.2.2.4 of deliverable D322. Said technique addresses interference management in small cell networks. The inter-cell interference mitigation is done in a decentralized manner by sensing the sounding reference signals (SRS) from UEs served by other Small Cells and processing the received signal. The technique is referenced in the following as Interference Management – Coordinated.

The *Interference Management* is implemented in the simulator by means of an external library that allows the execution of the algorithms developed with Matlab, as it is detailed in Section 7.1.2. The C++ environment of the simulator is able to interface with this library thanks to the *run-time engine* provided by Matlab, but this highly slows down the execution of the simulations. Moreover, because of the kind of optimization performed, it has to be executed at every sub-frame, with a period of 1 ms.
For these reasons the simulations have taken very long time for completing (from 18 minutes per second of simulation to a maximum of 44 minutes per second of simulation) and therefore the simulation time has been inevitably reduced. Nevertheless, even with duration not longer than a minute, the statistics of the data collected is wide enough to allow the evaluation of the algorithm, thanks to the very high repetition period.

Furthermore, in order to reduce the simulation events and obtain a more effective evaluation of the algorithm, the UEs have been configured for executing a special application that continuously exchange messages with the Small Cell, sending a new one right after having received the answer from its cell. In this way, the probability of simultaneous access to the radio channel is highly incremented, because the gap between two trasnsmissions, due to the task processing, is removed. The message size, 10 KB, is the same in Uplink and Downlink.

The deployment parameters are described in Table 28. A deployment with 3 Small Cells is considered, and then we vary the number of UEs per Small Cell from 3 to 12. Users active on a given sub-frame for a given Small Cell are distributed within the available RBs.

The same simulation setup has been executed two times for gathering the results from two different approaches:

- *Coordinated*: interference coordination among Small Cells at each RB is performed by following the MP2MP interference management technique developed in 3A2. The technique is used to adjust transmit precoders per RB (including power, spatial shape, and number of streams) at each Small Cell in a coordinated manner with the rest of Small Cells.

- *Uncoordinated*: each Small Cell decides about the transmit precoders per RB in a selfish manner without taking care of the generated inter-cell interference. This approach does not apply any interference managment with the neighbour Small Cells and is used as baseline for comparison.

The external library provides to the simulator the SINR per RB of each UE being served by each Small Cell. These SINRs are used in the simulator to compute the BLER, as detailed in Section 7.1.2, which enters inside the communication chain of the developed simulator. The metric used for comparing the results is the user throughput, measured in bits/s/Hz.

In Figure 40 the average user throughput is shown when increasing the number of UEs per Small Cell frome 3 to 12. Figure 41 displays the percentiles of the user throughput (10%-tile, 50%-tile and 90%-tile from upper to bottom) versus the number of UEs per Small Cell.

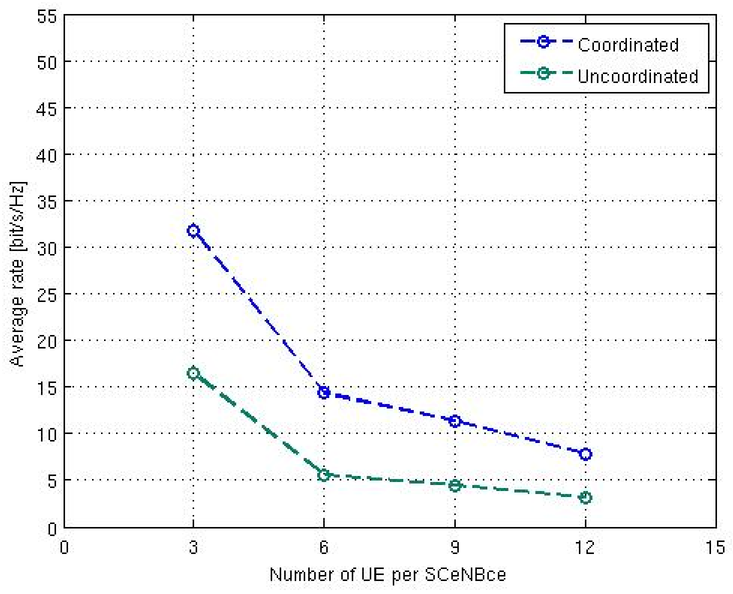


Figure 40. Average throughput for different UE number per SCeNBce

We can observe that significant user throughput gains are obtained with the proposed interference management procedure, even when it is evaluated not only at the physical layer (as it was done in 3A2) and all the chain in the simulator is used.

The gains are remarkable in average, 10%-tile, 50%-tile and 90%-tile. Furthermore, the gains are also appreciable and significant when the number of UEs per Small Cell is increased (i.e. the network load and the interference are increased). This shows that the proposed technique is effective at managing the created inter-cell interference in a small cell network, as it allows enhancing the downlink spectral efficiency and hence reducing the time needed to receive a packet at the UE.

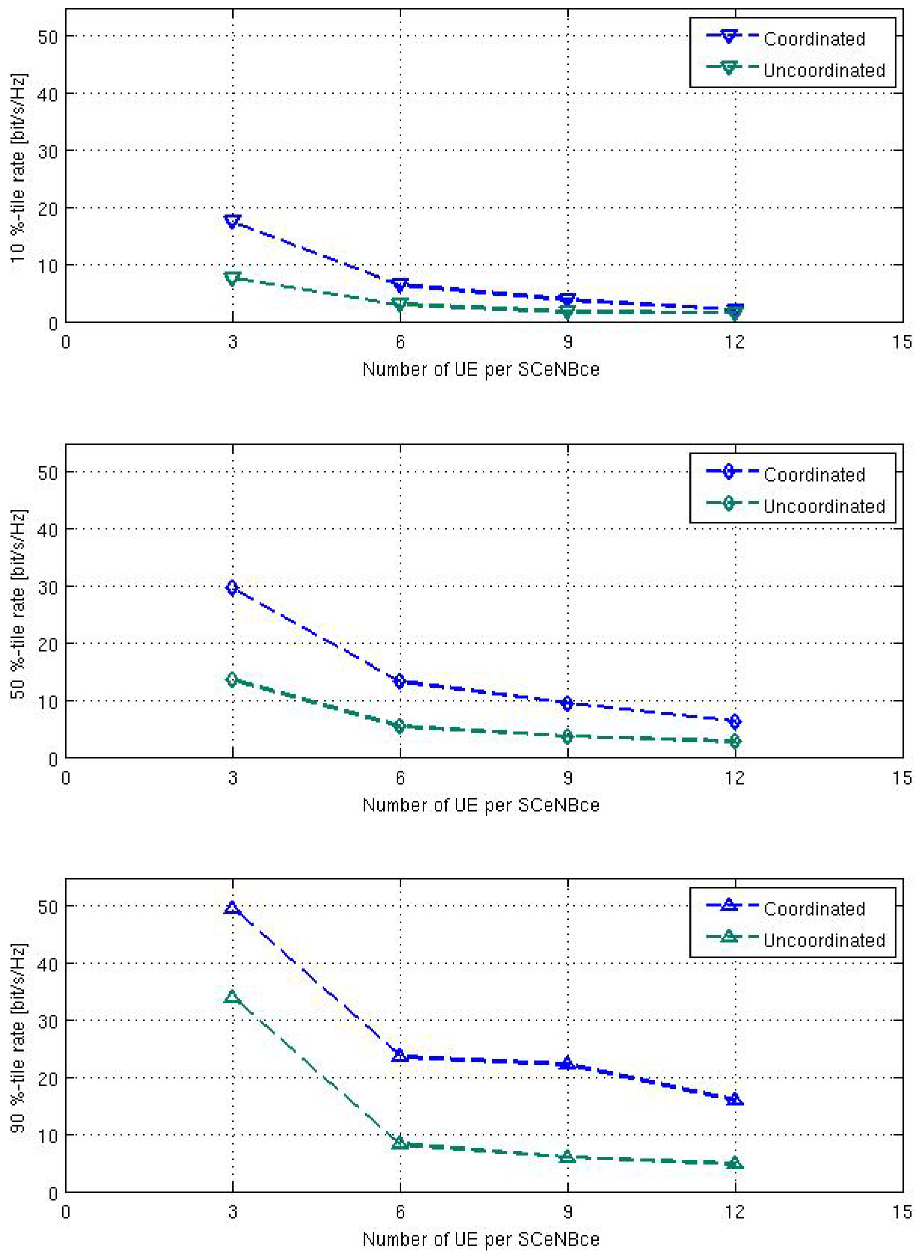


Figure 41. User throughput percentile: 10%-tile (upper) - 50%-tile (middle) - 90%-tile (bottom)

12 CONCLUSIONS

This deliverable describes the simulator developed in WP6, activity 6A1, as part of the demonstrator of the project. The purpose of the simulator is the evaluation of the system envisaged by TROPIC by modelling both the LTE-EPC infrastructure and the Cloud processing implemented on a cluster of Small Cells empowered with computing capabilities. This enhanced Small Cells are the core of the TROPIC innovations and, therefore, the simulation approach is only possible for evaluating the Cloud platform inside the standard LTE.

The simulator has been developed starting from the *ns-3 framework* with the *LENA* module for the LTE-EPC model. It makes available accurate models for both the radio LTE and the network communications, allowing the implementation of models and algorithms studied during the project, without the need of creating the model on which they are based. On the contrary, the whole Cloud platform, with the virtualized computing environment, has been implemented totally from the scratch according to the models proposed in WP5.

Two offloading algorithms have been implemented and evaluated *Energy-Latency Tradeoff* and the *Joint Radio-Cloud Optimization*, studied in WP3 and WP5. The later uses some mathematical tools for the optimization, available inside Matlab environment, but not inside the *ns-3* platform, based on the standard C++. For this reason, it has been implemented has an external Matlab library, enclosed and linked by the simulator. The access to the algorithm has required the integration of run-time Matlab libraries and the creation of the data structure used in such environment.

The same mechanism has been used also for the *Interference Management* algorithm, because of its extensive usage of link-level parameters not available a system level simulator as *LENA-ns3* is.

By means of the WP6 simulator, the algorithms studied in workpackages in WP3, WP4 and WP5 can be evaluated, using metrics different from those used for their specific analysis already performed, allowing a more general assessment inside the system complete with all its components.

The evaluation of the system performances are basically focused on two aspects that TROPIC expects to improve: the energy spent by the UE and the latency of demanding applications, with the offloading on the Small Cell Cloud.

In section 11.1 the *Energy-Latency Tradeoff* is used in the offloading decision on the SCM, while the results with the *Joint Radio-Cloud Optimization* algorithm are shown in section 11.2. With both the algorithms, it can be observed how the offloading on the Small Cell Cloud produces an improvement of the QoE, in terms of a reduced overall latency with respect to the processing on the user device. However, this observation is no longer valid when the number of UEs per SCeNBce increases.

With the *Energy-Latency Tradeoff*, the imposed latency constraint can be satisfied with a number of UEs per Small Cell smaller than around nine (Figure 32), but also with a bigger number of UEs, the latency continues to be lower than the latency experienced with the local execution of the application, evaluated in sections from 11.1.1 to 11.1.3.

The offloading, moreover, allows the UEs to reduce the energy spent to run the application, besides the additional energy consumption for the communications with the SCeNBce. From the point of view of the energy the trend is similar to the latency when the number of UEs is increased (Figure 31), but the amount of energy spent for local processing is reached with a higher number of UEs attached to each Small Cell.

With the offloading based on the algorithm *Energy-Latency Tradeoff* the behavior of the latency and the energy with the number of UEs per Small Cell is influenced by the progressive reduction of the percentage of application processed on the SCeNBce and, therefore, the increase of the portion to be executed locally, as Figure 33 shows.

The performance are also compared with the processing on a convetional Cloud (in sections from 11.1.1 to 11.1.3), even though with some limitations on the implemented model, described in 6.4. Despite the greater computing resources available on this Cloud with respect to SCC, in this case an important role on the latency is played by the backhaul, with its delay due to the background congestion. The latency shows a wider variability, but for most of the application sent it is lower than the local processing and higher than the processing on SCC. In this case the energy is spent by the UE only for the radio communications and the absence of the component for processing produces a result comparable with the offloading towards the Small Cell Cloud.

The *Joint Radio-Cloud Optimization* algorithm confirms the the limits when the number of UEs is increased, observed with the previous algorithm. However, the different approach in the selection of the offloading parameters and the continuous update with the status of the offloading of all the UEs, produce a lower dependency with the number of UEs of the SCeNBce, at least until the latency constraint can be satisfied (Figure 34 and Figure 35).

The results about the *Interference Management* algorithm are shown in Figure 40 and Figure 41. Even though it has not been possible to evaluate this algorithm together with the offloading (for a more effective evaluation) due to the lack of resources, the gains in terms of user throughput (in downlink) can further improve the system performance reducing the component of the latency due to the reception of the tasks processed from the SCeNBce.