On using Edge Computing for computation offloading in mobile network

Farouk Messaoudi*, Adlen Ksentini†, and Philippe Bertin*

*IRT b<>com, †Institut Eurecom
Email: *name.surname@b-com.com, †name.surname@eurecom.fr

***Abstract*—Mobile edge computing (MEC) emerges as a promising paradigm that extends the cloud computing to the edge of pervasive radio access networks, in near vicinity to mobile users, reducing drastically the end-to-end access latency to computing resources. Moreover, MEC enables the access to up-to-date information on users' network quality via the radio network information service (RNIS) application programming interface (API), allowing to build novel applications tailored to users' context. In this paper, we present a novel framework for offloading computation tasks, from a user device to a server hosted in the mobile edge (ME) with highest CPU availability. Besides taking advantage of the proximity of the MEC server, the main innovation of the proposed solution is to rely on the RNIS API to drive the user equipment (UE) decision to offload or not computing tasks for a given application. The contributions are two-fold. First, the design of an application hosted in the ME, which estimates current value of round trip time (RTT) between the UE and the ME, according to radio quality indicators available through RNIS API, and provide it to the UE. Second, the elaboration of a novel computation algorithm which, based on the estimated RTT coupled with other parameters (e.g., energy consumption), decide when to offload UE's applications computing tasks to the MEC server. The effectiveness of the proposed framework is demonstrated via testbed experiments featuring a face recognition application.**

I. Introduction

MEC is gaining lots of momentum, wherein the key idea is to empower mobile edge entities (e.g., radio access points such as eNodeBs and access gateways) with computation capabilities, allowing hosting and executing applications at the edge of mobile networks, rather than at a remote server in the Cloud or in the operator's core network domain. Combined with the 5G mobile access, which aims to drastically reduce the latency, MEC will enable a plethora of novel mobile services that require low latency to access data or computation capabilities. Among the envisioned services are computation-offloading-driven applications, which are able to offload part of the execution of their applications code to a remote server, benefiting from additional and highest CPU resources availability. Several works have been proposed in the literature to take advantage of nearby or remote servers to offload part of the computation from thin devices. Most of these works have been devised without considering MEC, since this concept is very recent. Moreover, they consider enforcing the offloading algorithms at the thin device side, ignoring all information on the device network environment, especially varying radio quality. This can lead to dramatic situations, as the thin devices may offload data even when the network conditions are bad (e.g., introducing very high RTT with the remote servers, not compatible with the devices application requirements). In this paper, we use MEC as an enabler for low-latency computation offloading-based applications, not only by hosting the remote server on the ME, but also by devising a ME service that continuously estimates the expected RTT in order to feed the device decision to offload parts of an application's computing tasks. Indeed, according to the European Telecommunications Standards Institute (ETSI) definition of MEC [1], the ME host is able to expose high level APIs to ME applications in order to provide real-time UE-relevant network state information, making possible the prediction of UE network state (particularly the RTT), and hence feeding the UE decision of when to offload, according to the network conditions. To the best of our knowledge, this is the first work that actively involves ME services in the UE's decision to offload part of an application for remote execution, by introducing interaction between the two components to drive the latter's decision, instead of relying on purely UE-local strategies.

This article is organized as follows. Section II reviews the state of the art in computation offloading in the context of edge computing, also describing the MEC reference architecture as defined by ETSI. Section III introduces our framework, by detailing each component (i.e., ME application, UE, and server). Section IV presents the obtained results and validates the proposed framework, before concluding the article in Section V.

II. Related Work

A. Computation offloading

Accelerated by the emergence of cloud computing, virtualization technology advances, and wireless network technology improvement (including, mobile broadband), computation offloading has attracted a tremendous amount of research works to make it a reality. Besides reducing energy consumption, computation offloading allows low-resource devices (e.g., smartphones) to run CPU-intensive applications like 3D gaming [2].

Usually, computation offloading consists in offloading a part or an entire application to a remote server in order to be executed there. This necessitates the design of algorithms able to decide which part of an application should be offloaded and

978-1-5090-5019-2/17/$31.00 ©2017 IEEE

when. Computation offloading requires four steps: (i) application profiling, (ii) application partitioning, (iii) execution of the decision algorithm, and (iv) data offload and communication with the remote server.

One of the big concerns of computation offloading is the packets delivery delay between the client and the server. High latency, as experienced today with servers hosted in the cloud (e.g., RTT around $200\,ms$), limits computation offloading applicability to delay-tolerant applications. Indeed, for delay-sensitive applications (e.g., interactive, and gaming), high latency is intolerable and causes the quality perceived by users to degrade. Hopefully, with the emergence of *Fog Computing* and *MEC*, it is possible to alleviate this constraint by reducing drastically the RTT with remote servers.

Fog is an extension of Cloud computing, bringing the network resources from the core network to the edge network. It is gaining a lot of industry support[1].

Although several works have been proposed to use Fog as an enabler for computation offloading [3], [4], [5], only few works have considered MEC. In [6], a power-constrained delay minimization problem for computation offloading based on Markov decision processes has been proposed and adapted to the MEC context. The authors have considered a mobile device running computation-intensive and delay-sensitive applications. The device is composed of a task buffer, a transmission unit, and a processing unit, and an algorithm has been proposed based on the average delay of each task and the average power consumption at the mobile device. This algorithm was used to solve the power-constrained delay model in order to find the optimal scheduling. The authors demonstrated their model with simulations. A multi-user resource allocation for MEC has been also proposed in [7]. The model is formulated as a convex optimization problem to minimize the mobile energy consumption considering the computation overhead and the capacity of the MEC. The model derives an offloading priority for each user according to its channel gain and energy consumption. A low priority derives a minimum offloading, while a high priority performs a complete offloading. Lastly, an extended study of computation offloading in a multi-cell environment was proposed in [8], wherein the authors considered a multiple inputs multiple outputs (MIMO) multi-cell system, with multiple users requests for computation offloading. The authors formulated the problem using a joint optimization of the radio and the computational resources for computation offloading in a dense deployment, with the presence of inter-cell interference. Clearly, none of these proposed solutions use the MEC services to drive the decision to offload or not. They all rely on local device's information to take such a decision. Our proposed framework overtakes this limitation by highly interacting with the MEC service to better predict UE's network quality of service, and thus considering up-to-date information to take offloading decision.

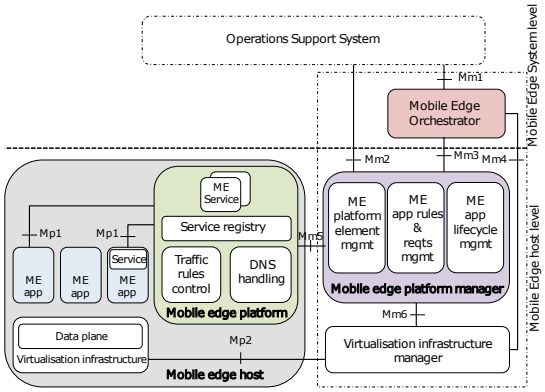

Fig. 1: **(Simplified) MEC architecture**

B. MEC Architecture

With MEC standard, the ETSI would mainly ease the deployment of UE applications requiring very low latency access to remote servers, which are usually hosted in the cloud. Examples of such targets are automotive systems, computation offloading, etc. Figure 1 portrays a simplified version of the MEC architecture as defined by ETSI. The ME management system comprises the ME orchestrator, the ME platform manager and the virtualization infrastructure manager (VIM). The ME orchestrator has the view on the whole ME system, as maintains the information about all the deployed ME hosts, the services and resources available in each host, the ME applications that are instantiated and the network's topology. The ME orchestrator is also responsible for installing the ME applications in the system, checking their integrity and authenticity and validating the policies associated to them. The ME host is the logical entity that contains the ME platform and the virtualization infrastructure on which the ME applications run. The ME platform contains a set of baseline functions that enable ME applications to run on a particular host, as well as to discover, and consume ME services, or to advertise and provide them through the service registry. The ME platform is also responsible for enforcing the traffic rules to route the data packets to/from the ME applications, as well as to maintain a DNS subsystem necessary to discover the ME applications. ME applications run on the ME host as virtual instances (i.e., virtual machine or container) and are designed to consume and/or provide ME services. The latter provide high level APIs that expose UEs status (radio and context information), which will be then used by the ME applications to optimize a registered service (e.g., offload computation). One of the most important API is the RNIS, which exposes radio access network (RAN) information such as radio quality indicators related to UE/eNodeB (eNB) layer 1/layer 2 parameters. It includes up-to-date information regarding the configuration and status of UEs and the access network. We may mention the following: UE's configuration information (e.g., public land mobile network identifier (PLMN ID), cell-radio network temporary

[1] https://developer.cisco.com/site/iox

identifier (C-RNTI), downlink/uplink (DL/UL) bandwidth); UE status information (e.g., global navigation satellite system (GNSS)); eNB configuration information (DL/UL radio bearer configuration, tracking area code, PLMN identity); eNB status information (GNSS, DL/UL scheduling information, number of active UE).

III. Proposed Framework

A. General description

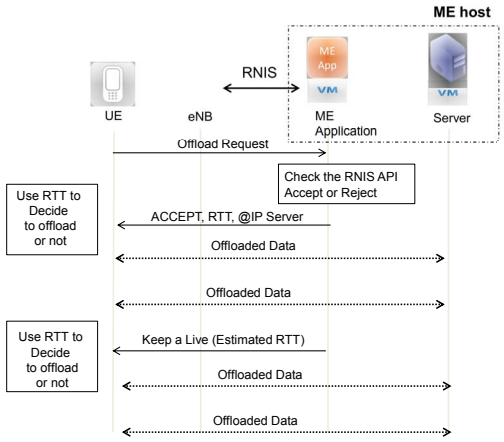

Fig. 2: **Global overview of the proposed framework**

The key idea of the proposed solution is to enable UEs to offload computation tasks to a remote server hosted in the ME host. As stated before, MEC allows the reduction of the end to end latency, which may ease the computation offloading process. Unlike the existing solutions, where UEs take locally (without the remote server help) the decision to offload or not a part of the computation task, in this work we rely on the MEC architecture, to not only host the remote server, but also drive the UE decision to offload or not a computation task.

One of the inputs usually considered to decide whether offload computation tasks, is the RTT between the remote server (hosted in the Cloud) and UE. In mobile networks, the RTT value is impacted by different parameters, which depend on both the core network and the wireless channel access delays. Note that hosting a server in the mobile edge to run the offloaded code prevents core network delay issue, any server located near eNBs being reacheable within a one hop connection. Therefore, only the wireless channel access may have impact on the RTT. On the DL direction, depending on the number of active UEs as well as the scheduler policy for the UE (i.e., guaranteed bit rate (GBR) or not) the packets towards UEs may experience delay. If the number of active UEs in the cell is high and the UE is using a no GBR bearer (e.g., the default bearer or a dedicated bearer with no GBR), the packets towards the UE might experience high delays at the eNB DL queue. In the UL direction, the same behaviour as for the DL may be seen (i.e., if the number of active UEs is high and the UE has no GBR, the packets from UE to the server may experience high latency, too). In addition, if the UE has very bad channel conditions it may not be scheduled often, which may also increase latency.

Estimating the value of the RTT in the wireless channel may be challenging as many types of information are required, such as the number of active UEs in the cell, the channel quality of the UE (channel quality indicator (CQI) and channel state information (CSI)), and the DL and UL bandwidth. But knowing that the ME application has access to these information via the RNIS API, it is possible to estimate the RTT and hence help the UE to decide to offload or not part of the computation tasks. A global view of the proposed solution is shown in Fig. 2. At the beginning, the ME application registers to use the RNIS API for specific UEs. Once a UE sends a request to offload data to a remote server, the ME application checks the RNIS API. If the ME application considers that it is better that the UE executes the application locally, then it sends a reject; otherwise it accepts the request, and includes the IP address of the ME server to connect to. Moreover, the accept message includes an estimation of the RTT, which will help the UE to decide to activate or not the computation offloading process. More details are given hereafter.

B. ME application Side

Once receiving the first request from a UE asking to offload data, the ME application checks the RNIS API to gather information on: the number of active UEs in the cell, the UE bearer type and the CQI of the UE. Obviously, if the UE's CQI is very bad, then it is better to execute the code locally as UE packets may experience high delays (due to the low modulation throughput), which may degrade the quality of the application run at the UE. Furthermore, if the UE is using a non-GBR bearer and the number of active UE is very high, then the offloading request is rejected. Even if the UE has a dedicated bearer, the ME application should check the associated QoS class identifier (QCI) in order to evaluate the packet delay budget allowed to this bearer. If the ME application estimates that the channel conditions are good or the UE has a dedicated bearer that guarantees low delay access, the UE is authorized to offload. The decision along with the remote IP address (i.e., if the offload is accepted) are included in the response of the ME application to UE. When the ME App decides to authorize the UE to offload data, it provides also an estimation of the RTT. The initial value could be the budget of packet delay associated with the bearer type of the UE (i.e., maximum tolerated latency). Then periodically, the ME application will further estimate the RTT according to the RNIS information and provide it to the UE.

Many works have tried to model the access latency in long term evolution (LTE) radio access. Among them, the authors in [9] proposed to rely on the remaining block signal (RBS), which is sent periodically by the UE to the eNB, in order to estimate the access delays. Though this solution shows a good approximation, it is difficult to use it at the ME application, since the RBS will arrive delayed to the ME application,

leading to wrong estimations. One straightforward solution on which we rely to estimate the RTT is based on the UL scheduling report obtained via the RNIS API. Indeed, the UL Scheduling information indicates the physical resource block (PRB) assignment per UE. The ME application will gather the UL information during a period of time, noted T, and analyze it to estimate the delay. We know that the transmission time interval (TTI) (or the smallest scheduling interval) in LTE is $1\,ms$. If we fix T to $1\,s$, then each period will include 1000 UL scheduling samples to be analyzed. If the UE has been scheduled X times during T, we can estimate the average access delay during the period T, noted by D, as T/X. Of course this calculation represents an average, which may not lead to an exact estimation, but at least it can be used easily to estimate the UE's packet delay, requiring only the UL scheduling information available through the RNIS API.

It is well known that the RTT could be estimated by multiplying the access delay by two, as the DL latency could not exceed the UL latency but would somehow be equivalent. Moreover, the backhaul latency may be omitted, as the server is located at the ME host. Then, the estimated RTT is equal to $2 \times D$. To avoid a sudden fluctuation of the estimated RTT, an exponential weighting moving average (EWMA) technique is used. The final value of the RTT, to be sent to the UE periodically is as follows:

$$RTT_{est} = \alpha \times RTT_{prec} + (1 - \alpha) \times RTT_{cur}, \qquad (1)$$

where $\alpha \in [0, 1]$, and RTT_{prec} and RTT_{cur} represent the most recent RTT estimate and RTT sample respectively.

C. UE Side

Basically, We assume an application composed by n classes. An application is represented using a valued and locality-labeled graph (VLG) with n vertices. Each vertex in the graph corresponds to one class in the application; it is labeled with a binary value to indicate whether the component can be offloaded or not, according to its dependencies on the system and/or hardware components like global positioning system (GPS), modem, screen, and I/O. Indeed, some parts of the application depend on the UE device hardware and cannot be offloaded. We used the execution time on the UE and the energy consumption to assign values to vertices. Each edge in the graph represents the interactions between two classes in the application and is weighted with the frequency of calls and the size of the exchanged data. The problem consists in identifying which classes of the application could be offloaded aiming at improving the response time and saving energy on the UE device. To this end, we propose to use the graph partitioning technique to obtain a minimum cut. Stemming from the fact that graph partitioning is an NP-hard problem (thus, its solution is prohibitive computationally), we propose to reduce the graph size by clustering the vertices that highly communicate.

From the valued and locality-labeled graph (VLG) we construct a reduced valued and locality-labeled graph (RVLG). The first step consists in transforming the VLG into a reduced graph (RG) using the strongly connected components (SCC) algorithm that constructs the clusters (detailed later), leading to reduce the size of the graph as well as the communication between components. Indeed, reducing the graph size permits to decrease the time duration for resolving it. Now, we assign values to the RG by computing the vertice weights and identify the vertice labels using the procedure *VerticesConstruct*. For the edge weights, we use the procedure *EdgesConstruct*. The result is a reduced valued and locality-labeled graph (RVLG).

a) Clusters Construction: The SCC algorithm constructs a cluster from a given vertex a. This cluster will at least contain the element a. To construct the cluster we proceed as follows. First, we initialize all the components of the graph to *non visited* (lines 3 and 9). Next, we create two sets of vertices (lines 4, 10). One of them will contain all the successors of all the vertices in this set (lines 4 to 8), and the other one will regroup all the predecessors of all the elements in this set (lines 10 to 14). Finally, we obtain a cluster which is the intersection between the two sets of vertices (line 15).

```
1:  procedure SCC(G(V, E), a ∈ V)
2:      begin
3:      Initialization: Set all the vertices with the label visited = false
4:      Create a set of vertices V1 initialized with a (i.e., V1 = {a});
5:      while there is a non visited vertex v in the set V1 do
6:          mark this vertex v with the label visited = true;
7:          for each edge (v, y), where the vertex y does not belong to V1
    do
8:              Add y to V1;
9:      Re-Initialization: All the vertices are set to non visited (i.e.,
    visited = false);
10:     Create a second set of vertices V2 initialized with a (i.e., V2 = {a});
11:     while there is a non visited vertex v in the set V2 do
12:         mark this vertex v as visited;
13:         for each edge (x, v), where the vertex x does not belong to V2
    do
14:             Add x to V2;
15:     return the intersection between the two sets V1, V2 (i.e., the cluster)
16:     end
```

b) Graph Reconstruction: Once the clusters are created, we reconstruct the graph with the obtained clusters. We start by defining the vertices of the reduced graph in the algorithm *VerticesConstruct*. Each vertex in the new graph is actually a cluster (with at least one element). Thus, the weights (time and energy) of each cluster are the cumulative weight of all the vertices in this cluster (lines 9 and 10). Then we assign a label to the cluster C_i, which depends on the labels assigned to the vertices composing the cluster C_i. If at least one vertex in the VLG that belongs to the cluster C_i, is labeled with the $l = 0$, then the cluster C_i should be labeled with $l_i = 0$, otherwise $l_i = 1$. In other words, if a class from the cluster C_i cannot be offloaded, then the whole cluster C_i cannot be offloaded (lines 12 and 13).

When all the vertices are defined, we construct the edges using the algorithm *EdgesConstruct*. For each edge in the original VLG, if the two vertices that constitute the edge belong to two distinct clusters, then an edge between the two clusters (if it does not exist, line 7) is created, and the weight

of this edge is incremented with the weight of the respective edge in the VLG (line 8). This process is repeated for all the clusters.

```
1: procedure VERTICESCONSTRUCT(G(V, E))
2:     begin
3:     G'(V', E') ← G(V, E);
4:     i ← 1;
5:     V_1 = {a}
6:     while (V' ≠ ∅) do
7:         Choose, randomly, a vertex v in V';
8:         Construct a cluster C_i with the vertex v by calling the algorithm
       SCC(G'(V', E'), v);
9:         Compute the execution time of the cluster C_i by accumulating
       the weights (times) of all the vertices in this cluster;
10:        Compute the energy consumption of C_i by accumulating the
       weights (energy) of all the vertices in this cluster;
11:        V' ← V' − C_i;
12:        if it exists a vertex in C_i initialized with (l_i = 0) then
       initialize the cluster C_iwith the locality l_{c_i} ← 0;
13:        else l_{c_i} ← 1;
14:        i++;
15:    return C, the set of the clusters;
16:    end
```

```
1: procedure EDGESCONSTRUCT(G(V, E), C = ⋃_{j=1}^{i} C_i)
2:     begin
3:     for (i = 1, i ≤ card(C), i++) do
4:         j ← 1;
5:         while ((j ≤ card(C)) & (j ≠ i)) do
6:             for each (((u = (v, w) ∈ E) & (v ∈ C_i)) & (w ∈ C_j))
       do
7:                 Create an edge (C_i, C_j) in the RVLG if it does not
       exists;
8:                 W(C_i, C_j) += W(v, w);
9:             j++;
10:    end
```

1) Decision Algorithm: Regarding the decision algorithm (i.e., offload a cluster or not), we propose two different models; *Global-View* and *Local-View*. In the Global-View model, we focus on the optimal solution. To this end we model the offloading decision problem using an integer linear programming (ILP). However, the resolution of the ILP may take time and drastically increase with the size of the application. Indeed, there is a local latency (overhead) of the model resolution to consider in the response time. In the Local-View model, we try to minimize this overhead by proposing a more simple but efficient heuristic solution. Both algorithms take as inputs the reduced version of the graph, the wireless network characteristics (bandwidth and RTT) and mobile resources (energy and computing resources), and give as output the offload decision.

a) Global-View Model: We consider that the total execution time of the application is split into three parts. The first part is the local execution time related to the execution of the clusters on the UE. The second part is the remote execution time, which considers the execution cost of the remaining clusters on the server. Finally, there is the communication cost, which includes the latency as well as the time to send and receive the results from the server. We define the energy threshold value E_{th}, as the maximum amount of energy that can be spent for the application processing. We set this value to 95% percent of the total consumption of the application when it is executed locally. This constraint is useful in our model: In the worst case, we gain 5% energy, and more importantly, this constraint prevents from wasting additional energy due to the network communication, as our objective function focuses on minimizing the execution time.

The algorithm steps are as follows. First, the graph size will be reduced using the cluster representation, by computing the $VerticesConstruct$ and $EdgesConstruct$ algorithms, which regroup the nodes of the VLG inside clusters and compute the weights of these clusters as well as the interaction between them, including the edge weight and locality. A RVLG is the result of this step, which should be written in matrix format in order to be given as input to an ILP. The proposed ILP aims to improve the execution cost of the overall program, while maintaining the energy consumption under a certain threshold (E_{th}). It is given by:

$$Min \sum_{i=1}^{n} \left((1 - x_i) T_i + x_i \left(\frac{T_i}{s} + \frac{d_i}{UL} + \frac{r_i}{DL} + 2 \times RTT \right) \right)$$
$$s.t. \begin{cases} \sum_{i=1}^{n} \left((1 - x_i) E_{cpu} + x_i (E_{idle} + E_{nic}) \right) \leq E_{th} \\ \forall i \in \{1, ..., n\}, \ x_i \in \{0, 1\}, \ l_i \in \{0, 1\} \\ x_i \leq l_i \end{cases} \quad (2)$$

where T_i represents the execution time of cluster C_i, d_i is the envisioned data (including the code and inputs) size to be sent to the remote server (i.e., if cluster i is offloaded), r_i represents the executed data (results) size sent from the remote server, s is the server CPU speed (this could be obtained along with the IP address of the remote server), and x_i is a binary variable that indicates the ILP output: $x_i = 0$ (respectively $x_i = 1$) denotes local (respectively remote) execution of this cluster. The constraint $x_i \leq l_i$ concerns the clusters that can not be offloaded, due to the hardware dependency, as stated before; therefore, if a cluster is labeled with $l_i = 0$ (i.e, non-offloaded cluster), then $x_i = 0$ (i.e., this cluster will not be offloaded). Next, the ILP is solved to compute the values of each x_i. After that, each cluster C_i is assigned to one of the two partitions (local partition or remote partition) based on the value of the corresponding x_i; that is, if $x_i = 0$, then the corresponding cluster is assigned to the local partition, otherwise it is assigned to the remote one. At the end, the algorithm returns the set of clusters that should be offloaded (remote partition), and those that should be executed on the UE (local partition).

b) Local-View Model: In the Local-View model, each cluster composing the application is treated separately from the others. The proposed algorithm dissects the possibility of offloading a cluster or not. with respect to the possibility to be offloaded. The purpose is to avoid using the ILP, in order to reduce the resolution time compared with the Global-View algorithm. Indeed, for some low-latency applications, there is a need to decide rapidly to offload a cluster or not, to avoid additional delays. Like the Global-View model, the proposed algorithm accepts as input a RVLG and generates

two partitions as output; the local partition and the remote partition. The concept of this algorithm is simple; for each cluster C_i, if either the latency or the cost of communication (i.e., exchanged data) with the ME is higher than the local execution time of this cluster, then offloading the cluster will neither improve the computation cost nor the consumption cost (i.e., $x_i = 0$). Otherwise, if both latency and the communication cost are less than the execution time, the algorithm checks if a gain is achieved when offloading the cluster. The *offloading gain* is the difference between the local cost and the offload cost (the offload cost includes the communication cost and the remote execution) and is defined as

$$TG_{off} = w \times \left(\frac{1}{s_m} - \frac{1}{s}\right) - \left(\frac{d_i}{UL} + \frac{r_i}{DL} + 2 \times RTT\right) \quad (3)$$

where w is the the amount of computation for cluster C_i, and s_m represents the UE processing speed.
If offloading a cluster will achieve a gain, then it will be offloaded (i.e., $x_i = 1$).

c) Communication between server and UE: Once the decision algorithm has been executed (i.e., the computation offloading process is launched), the UE and the remote server interact through the communication channel that has been established just after receiving the authorization to offload from the ME application. For reliability reasons, this communication channel is based on TCP and implements a control protocol which allows the UE or the server to ask for intermediate data. For example, a cluster in the MEC end can request the results of the execution of another cluster located on the UE, or the results obtained by the cluster in the MEC side could be used by another cluster in the UE.

IV. Results

To evaluate the proposed framework, we developed a testbed composed of a UE running a face recognition application based on the OpenBr[2] framework on top of an Ubuntu 14.04 operating system, and a MEC server. The UE was emulated by a virtual machine hosted on a Dell M4800 laptop, powered by an Intel Core i7 processor (clocked at $2.8\,Ghz$) running Ubuntu 14.04, also including an Nvidia Quadro K2100M GPU card using 2 GB. The server is 8 times faster than the UE, while the UL bandwidth is around $50\,Mbps$, and the DL bandwidth is $60\,Mbps$. The aim is to compute the gain of both proposed solutions, when offloading a part of the application tasks. After the profiling step, 96 classes are used by the application. These classes were then grouped into clusters. Figure 3 illustrates the number of clusters to offload regarding the latency between the UE and the MEC server. We make two main observations here.

- *The number of clusters to offload is inversely proportional to the latency.* This remark is applicable for both proposed algorithms (i.e., Global-View and Local-View). When the network latency is increasing, there is less offloading gain, hence less clusters to offload. Basically, the Global-View (respectively Local-View) algorithm minimizes (respectively maximizes) the model defined by equation 2 (respectively the gains through equation 3). To minimize equation 2, the second part of the sum (i.e., $x_i \left[\frac{T_i}{s} + \frac{d_i}{UL} + \frac{r_i}{DL} + 2 \times RTT\right]$) should be equal to zero when the latency (RTT) is high (i.e., $x_i = 0$). For equation 3, there is a gain ($TG_{off} > 0$) when the RTT is less than $\left(\frac{1}{2}\right) \times \left[w\left(\frac{1}{s_m} - \frac{1}{s}\right) - \left(\frac{d_i}{UL} + \frac{r_i}{DL}\right)\right]$. Once the latency reaches $110\,ms$, the performances of both algorithms converge; less than 10 clusters are offloaded. The number of offloaded clusters is merely zero when offloading a cluster will not generate gains.
- *The Global-View algorithm identifies more clusters to offload than the Local-View.* Deciding where to execute each cluster subject to latency constraints is very challenging, as it requires a global view of the program's behavior. The solution is optimal only when the decision strategy is globally optimal (i.e., across the entire program) rather than locally optimal (i.e., relative to a single cluster invocation). The Global-View algorithm solves an ILP, which helps to derive the optimal solution. Thus, for each latency value, the ILP finds the maximum number of clusters that minimizes the cost (i.e., the optimal number of clusters to offload). The Local-View algorithm, on the other hand, focuses on each cluster independently of the others, searching only for local solutions. A cluster is offloaded if and only if a gain (i.e., TG_{off}) is obtained. We believe that the difference between the two curves is due to the communication cost of some clusters which do not result in a local gain as the communication cost exceeds the computation one but improve the global gain.

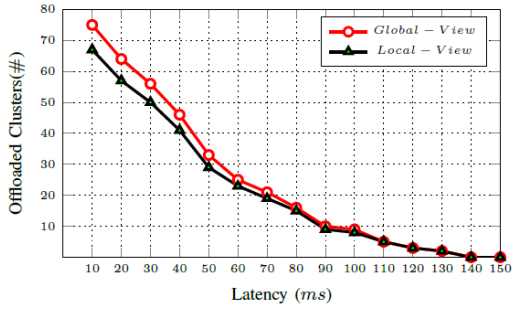

Fig. 3: **Number of offloaded clusters by report to the RTT between the UE and the MEC sever**

Now we turn our attention to the gain that could be achieved for users. For this purpose, we focus on two important metrics regarding improving user experience: (i) the time needed to execute the face recognition application, and (ii) the energy consumed at the user end.
Figure 4 presents the response time using three approaches: (i) the whole application is computed locally on the UE, (ii) the application is computed using the Local-View algorithm, and (iii) the application is executed employing the Global-View algorithm. The two last approaches are subject to the

[2]http://openbiometrics.org/

network latency. The total execution time for face recognition takes around $9.5\,s$. This time corresponds to the elapsed time between the moment when the web camera of the laptop is launched and the moment of viewing the result. Inside this delay interval, different actions are performed, including (i) launching the web-camera, (ii) capturing a video (a set of 300 frames), (iii) communication with a graphical QT interface to display the video in real-time, (iv) detection (detecting eyes, face, key-points, and landmarks) for each frame to have the illusion of tracking, (v) normalization (applying color conversion, enhancement, and filtering), (vi) representation (computing binary patterns, key-point descriptors, orientation histograms, and wavelets), (vii) extraction (using clustering, normalization, and quantization), and (viii) matching (using classifiers, distance metrics, and density estimation).

The Global-View algorithm achieves the best performance for all the latency values. For a $10\,ms$ latency, the Global-View algorithm (respectively Local-View) offers almost 69 (respectively 62) percent of gain compared with the local execution. For small latency ($10\,ms$ to $60\,ms$), the Global-View algorithm is almost $1.13\times$ optimal compared with the Local-View Algorithm. For a latency higher than $110\,ms$, the two algorithms offer the same performance, since both identify the same clusters to offload. As the latency is high, the two algorithms converge to the same results.

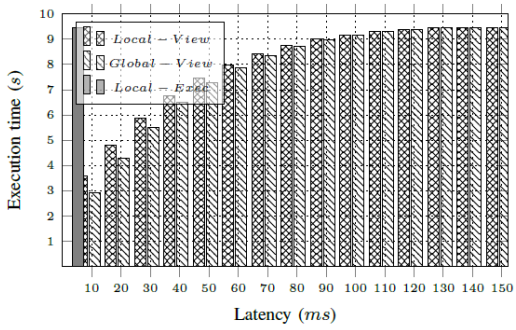

Fig. 4: **The execution time by report to the RTT between the UE and the MEC sever**

Figure 5 depicts the energy consumed by the face recognition application for the three scenarios (i.e., local execution, Global-View algorithm, and Local-View algorithm). Computing the whole application locally consumes $7.564\,J$. For $10\,ms$ latency, the Global-View (respectively Local-View) algorithm offers almost 93.4 (respectively 90.5) percent better performance than the local execution. This gain is inversely proportional to latency. Indeed, when latency increases, the number of clusters to offload decreases, in turn decreasing the achieved gain, which depends on the number of clusters to offload. The Global-View algorithm is almost 3 percent better than the Local-View algorithm. For latencies higher than $140\,ms$, the achieved gain by the both proposed algorithms is null.

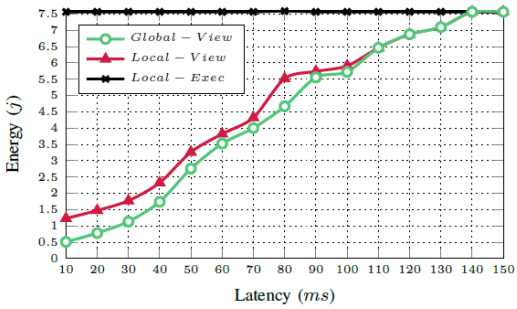

Fig. 5: **Energy consumption versus the network latency**

V. Conclusion

In this paper, we proposed a novel framework for computation offloading, which uses the MEC architecture defined by the ETSI, taking advantage of its RNIS API to estimate network latency between the UE and a server hosted in the ME. This estimation, coupled with other parameters, is used to derive an offloading decision for the UE. We formulated two models operating at different granularities (application component vs. component cluster level). Whilst the first model searches for an optimal offloading, the second searches for a fast solution. The effectiveness of our design, tested using a face recognition application, unveils an execution 62 percent faster than the local execution, and can save up to 93.5 percent of energy for the UE.

References

[1] ETSI, "Mobile edge computing (mec); framework and reference architecture," *ETSI GS MEC*, vol. 3, p. V1.1.1, March 2016.

[2] F. Messaoudi, G. Simon, and A. Ksentini, "Dissecting games engines: The case of unity3d," in *Proceedings of the 14th ACM Netgames Workshop, Zagreb, Croatia, December 3-4*, 2015, pp. 1–6.

[3] M. A. Hassan, M. Xiao, Q. Wei, and S. Chen, "Help your mobile applications with fog computing," in *12th Annual IEEE International Conference on Sensing, Communication, and Networking Workshops, SECON Workshops 2015, Seattle, WA, USA, June 22-25, 2015*, 2015, pp. 49–54.

[4] S. Yi, Z. Hao, Z. Qin, and Q. Li, "Fog computing: Platform and applications," in *Third IEEE Workshop on Hot Topics in Web Systems and Technologies, HotWeb 2015, Washington, DC, USA, November 12-13, 2015*, 2015, pp. 73–78.

[5] N. Chen, Y. Chen, Y. You, H. Ling, P. Liang, and R. Zimmermann, "Dynamic urban surveillance video stream processing using fog computing," in *IEEE Second International Conference on Multimedia Big Data, BigMM 2016, Taipei, Taiwan, April 20-22, 2016*, 2016, pp. 105–112.

[6] J. Liu, Y. Mao, J. Zhang, and K. B. Letaief, "Delay-optimal computation task scheduling for mobile-edge computing systems," *CoRR*, vol. abs/1604.07525, 2016.

[7] C. You and K. Huang, "Multiuser resource allocation for mobile-edge computation offloading," in *Proc. IEEE Globecom*, 2016.

[8] S. Sardellitti, G. Scutari, and S. Barbarossa, "Joint optimization of radio and computational resources for multicell mobile-edge computing," *IEEE Trans. Signal and Information Processing over Networks*, vol. 1, no. 2, pp. 89–103, 2015.

[9] A. Baid, R. Madan, and A. Sampath, "Delay estimation and fast iterative scheduling policies for LTE uplink," in *10th International Symposium on Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks (WiOpt), Paderborn, Germany, May 14-18, 2012*, 2012, pp. 89–96.