Resource Calendaring for Mobile Edge Computing in 5G Networks

Bin Xiang  
Politecnico di Milano  
bin.xiang@polimi.it

Jocelyne Elias  
University of Bologna  
jocelyne.elias@unibo.it

Fabio Martignon  
University of Bergamo  
fabio.martignon@unibg.it

Elisabetta Di Nitto  
Politecnico di Milano  
elisabetta.dinitto@polimi.it

Abstract—Mobile Edge Computing (MEC) is a key technology for the deployment of next generation (5G and beyond) mobile networks, specifically for reducing the latency experienced by mobile users which require ultra-low latency, high bandwidth, as well as real-time access to the radio network. In this paper, we propose an optimization framework that considers several key aspects of the resource allocation problem for MEC, by carefully modeling and optimizing the allocation of network resources including computation and storage capacity available on network nodes as well as link capacity. Specifically, both an exact optimization model and an effective heuristic are provided, jointly optimizing (1) the connections admission decision (2) their scheduling, also called calendaring (3) and routing as well as (4) the decision of which nodes will serve such connections and (5) the amount of processing and storage capacity reserved on the chosen nodes. Numerical experiments are conducted in several real-size network scenarios, which demonstrate that the heuristic performs close to the optimum in all the considered network scenarios, while exhibiting a low computing time.

Index Terms—Calendaring, Network slicing, Network Design, Edge computing, Joint Optimization

I. INTRODUCTION

Next generation (5G and beyond) mobile networks are currently being deployed, and need to provide services characterized by ultra-low latency, high bandwidth, as well as real-time access to the radio network. To achieve these goals, Mobile Edge Computing (MEC) is envisaged to provide an IT service environment and cloud-computing capabilities at the edge of the mobile network, within the Radio Access Network and in close proximity to mobile subscribers; through this approach the latency experienced by mobile users can be considerably reduced. However, the computation power that can be offered by an edge cloud is limited if compared to a remote cloud. Considering that 5G networks will be likely built in an ultra-dense manner, the edge clouds attached to 5G base stations will also be massively deployed and connected to each other in a specific mesh topology. Thus, by exploiting the cooperation among multiple edge clouds and by carefully allocating edge resources to each connection, we can provide a solution to the limitations of a single MEC unit.

In this paper, we provide an optimization framework (an exact model as well as an efficient heuristic approach) that considers several key aspects of the resource allocation problem in the context of Mobile Edge Computing. Specifically, our proposed model and heuristics jointly optimize (1) the admission decision (which connections are admitted and served by the network, based on the profit they can potentially generate with respect to the required resources for serving demands), (2) the scheduling of admitted connections, also called calendaring (taking into account the flexibility that some users exhibit in terms of starting and ending time tolerated for the required services), (3) the routing of these flows, (4) the decision of which nodes will serve such connections as well as (5) the amount of processing and storage capacity reserved on the chosen nodes that serve such connections, with the objective of maximizing the operator’s profit.

To our knowledge, our work is the first one that considers all these five aspects together. Other works focus, instead, on specific aspects. For instance, in [1], the authors study a task offloading model considering constraints on task queue lengths to minimize the users’ power consumption, while the work in [2] jointly considers task assignment, computing and transmission resources allocation to minimize system latency in a multi-layer MEC context. The authors in [3] study traffic processing and routing policies for service chains in distributed computing networks to maximize network throughput. These works, however, do not consider the resource scheduling problem. The work in [4] studies bandwidth calendaring to allocate network resources and schedule deadline-constrained data transfers, while [5] studies the problem of scheduling and routing deadline-constrained flows in data center networks to minimize the energy consumption. However, the allocation of computing resources is not considered in these works. In [6], the authors study the problem of dispatching and scheduling jobs in edge-cloud system to minimize the job response time; [7] studies online deadline-aware task dispatching and scheduling in edge computing to maximize the number of completed tasks. Finally, the work in [8] proposes a two-time-scale strategy for resource allocation by performing service placement (per frame) and request scheduling (per slot) to reduce the operation cost and system instability. These works, though, do not explicitly consider the routing problem that arises. In our previous works, we first focus in [9] exclusively on minimizing the latency of traffic in a hierarchical network, keeping the network and computation capacity fixed. Then, in [10], we address the joint network planning, slicing and edge computing problem, aimed at minimizing both the total latency and operation cost for arbitrary network topologies. However, in these works we do not consider the requests admission as well as the scheduling of the computation.
storage and bandwidth resources.

To jointly optimize allocation of multiple resources and scheduling, we first formulate an exact optimization model, which turns out to be both nonlinear (due to latency constraints that we model accurately) and integer, and then we provide a reformulation\(^1\) that transforms our original problem into an equivalent Mixed Integer Quadratically Constrained Problem (MIQCP), which can be solved by available commercial solvers. Subsequently, an effective heuristic, named Sequential Fixing and Scheduling (SFS), is proposed. We compare our proposed model and heuristic to a greedy approach, which provides a benchmark for our solutions. Numerical results demonstrate that the proposed heuristic performs close to the optimum in all the considered network scenarios, with a very short computing time.

The paper is organized as follows: Section II illustrates the problem formulation and the proposed exact optimization model. Section III presents the heuristic. A numerical analysis and comparison of the proposed model and heuristics is performed and discussed in Section IV. Finally, section V concludes the paper.

II. PROBLEM FORMULATION

A. System Overview

We consider an edge cloud network represented by an undirected graph \(G(V, E)\), where each node \(v \in V\) represents an edge computing node having \(D_v\) and \(S_v\) as computation and storage capacity, respectively. The two parameters \(\theta_v\) and \(\phi_v\) denote, respectively, the cost of computation and storage capacity of node \(v\). Each edge \(e \in E\) corresponds to a network link characterized by its bandwidth \(B_e\) and its cost per unit of flow \(\psi_e\). Let \(K\) denote the set of requests, with different types, offered to the network. We regard each type of request as an aggregated communication-computation demand, e.g. web, video, game traffic etc., which has to be accommodated in the network and requires some amount of bandwidth, storage and computation resources. We assume that the calendar (i.e., the arriving time and duration) of the requests for the upcoming period is known. This can be achieved assuming that customers have announced their requirements in advance, or that some history-based prediction tools [11] are used.

We discretize the time horizon into a set \(\mathcal{T}\) of equal duration time-slots, where the slot length is \(\tau\). Each request \(k \in K\) is defined as a tuple \((s^k, \alpha^k, \beta^k, d^k, \lambda^k)\). The parameter \(s^k\) is the source node of request \(k\); \(\alpha^k, \beta^k\) and \(d^k\) define the arrival

\[\mathbf{A}\]

time, the latest ending time (deadline) and the duration of request \(k\), respectively. Finally, we consider a Poisson process for each request \(k\) with an average packet arrival rate \(\lambda^k\). The arrival and ending times coincide, respectively, with the arrival of the first packet and the departure of the last packet of request \(k\).

A request \(k\) could be processed immediately (for delay-sensitive tasks) after its arrival, or scheduled for later (for delay-tolerant tasks). Also, it could be entirely processed on the local edge computing node or split into multiple fractions and processed on other nodes. In any case, it must be completed before the deadline \(\beta^k\). Figure 1 shows the arrival time \(\alpha\), deadline \(\beta\) and duration \(d\) of requests 1 and 2. Also, it highlights that request 1 is scheduled to be served from time \(\xi^1\), delayed (shifted) with respect to \(\alpha\) but still compatible with \(\beta\). The ending time for the request will then depend on \(\xi^1\), \(d\), computing latency, and link latency along the routing path if (some fraction of) the request is offloaded to the neighbor edge computing nodes.

Given a calendar of requests \(\mathcal{T}\) over a time horizon, the proposed optimization approach must: a) schedule the starting time of each request, b) decide where to compute the requests, and c) route some fractions of requests when it is necessary to process them on other edge computing nodes, in order to maximize the profit of the provider.

B. Life Cycle of A Request

A given request \(k\) arriving at an edge node \(v\) at time \(\alpha^k\) could be: i. rejected, ii. processed immediately – this is needed if it is a delay sensitive task – or iii. shifted to a future epoch, if it is delay tolerant. To model the fact the delayed (shifted) starting time \(\xi^k\) can vary in the time frame \([\alpha^k, \beta^k - d^k]\), we express \(\xi^k\) as: \(\xi^k = \sum_{t=\alpha^k}^{\beta^k-d^k} t \cdot z^{kt}\), and we have:

\[\sum_{t=\alpha^k}^{\beta^k-d^k} z^{kt} \leq 1, \forall k.\]  

(1)

Essentially, \(z^{kt}\) is a binary variable that can be 1 at most in one point of time which corresponds to \(\xi^k\) for request \(k\). When \(z^{kt} = 0\) for all possible time slots, this implies that the request is not admitted and, therefore, not scheduled. Note that by changing the inequality constraint (1) to an equality, the edge cloud will be forced to serve all the incoming requests, which may be unfeasible in some cases.

A request can be either processed locally in a computing node or split and offloaded to other edge computing nodes. In the latter case, the processing latency, the storage provisioning constraints and the link latency along a routing path should be taken into account by the calendaring scheme. Considering a node \(v\) that is allocated to process a fraction \(\eta^kv\in[0,1]\) of request \(k\), the ending time at \(v\), denoted by \(\tau^kv\), can be expressed as: \(\tau^kv = \xi^k + d^k + \left[\frac{T^L}{\tau}\right] + \left[\frac{T^P}{\tau}\right]\), where \(T^L\) and \(T^P\) are respectively the link latency and processing latency. Note that both \(\xi^k\) and \(\eta^kv\) are integer values in the time slot set and \(\tau\) is the time-slot duration. The ending time of each request

\[\text{Fig. 1: Example of time scheduling of a request.}\]

depends on the last finished piece, which must be completed before the deadline. Such constraint is expressed as:

$$\max_{v \in V} \{ \xi_k^v \} \leq \beta_k, \ \forall k.$$  \hspace{1cm} (2)

In the following, we will express the request routing and the two latency components (link and processing latency) in detail.

C. Network Routing

We assume that a request can be split into multiple pieces only at its source node. Each piece can then be offloaded to another edge computing node independently of the other pieces, but it cannot be further split (we say that each piece is unsplittable). Each link $e \in \mathcal{E}$ may carry different request pieces, $q^e$ (remind that $q^e$ is the fraction of request $q$ to be processed at node $v$). Then, the total flow of request $k$ on link $e$, $f^k_e$, can be expressed as the sum of all pieces of $k$ that pass through such link: $f^k_e = \sum_{v \in V} \sum_{e \in \mathcal{E}} q^e k v$, where $\mathcal{R}^k \in \mathcal{E}$ denotes the routing path (set of traversed links) for the partial request $q^k$ from source node $s$ to node $v$. The traffic flow conservation constraint is enforced by:

$$\sum_{e \in \Phi^-_v} f^k_e - \sum_{e \in \Phi^+_v} f^k_e = \begin{cases} q^k - 1, & \text{if } v = s^k, \\ q^k, & \text{otherwise,} \end{cases} \forall k, \forall v, \hspace{1cm} (3)$$

where $\Phi^-_v$ and $\Phi^+_v$ are, respectively, the set of incoming and outgoing links of node $v$. The fulfillment of this constraint guarantees continuity and acyclicity for the routing path.

D. Link Latency

Let $T^k_L$ denote the link latency for routing request $k$ to node $v$. Each request is routed in a multi-path way, i.e., different pieces of the request may be dispatched to different nodes via different paths. The transmission time of the requests on each link is described by an $M|\mathcal{M}|1$ model; hence, $\forall k, \forall v$, $T^k_L$ is defined as:

$$T^k_L = \begin{cases} \sum_{e \in \mathcal{R}^k} \frac{1}{p^k_e B_e q^k q^e} \chi_e, & \text{if } q^k > 0 \& v \neq s^k, \\ 0, & \text{otherwise,} \end{cases} \hspace{1cm} (4)$$

where $p^k_e$ is the fraction of bandwidth capacity sliced for the request of $q^k$ flowing to node $v$ via link $e$. The link latency is accounted for only if a piece of request $k$ is processed at node $v$ (i.e., $q^k > 0$ and $v \neq s^k$). The following constraint ensures that the flow of request $k$ on each link of the routing path does not exceed the allocated capacity:

$$\begin{cases} q^k \lambda t < p^k_e B_e, & \text{if } e \in \mathcal{R}^k, \\ 0, & \text{otherwise,} \end{cases} \hspace{1cm} (5)$$

where $p^k_e$ is the fraction of link $e$'s bandwidth allocated for a piece of request $q^k$ at time slot $t$. Note that we assume that the reserved bandwidth for each request over its life period does not change in order to provide consistent service guarantee. The superscript $t$ in $p^k_{e,t}$ is used to indicate the life status of the flow. The relation between $p^k_{e,t}$ and $p^k_e$ is given by $p^k_{e,t} = \delta^k_{e,t} p^k_e$, where $\delta^k_{e,t}$ is a binary variable which is equal to 1 if $\xi^k_t \leq t < \xi^k_t + d^k + \lceil \frac{q^k}{\lambda t} \rceil$, and 0 otherwise.

E. Processing Latency and Storage Provisioning

When a request cannot be entirely processed locally, we assume that such request can be segmented and processed on different edge computing nodes. Hence, each node can slice its computation capacity to serve several requests coming from different source nodes. Notice that a request $k$ also requires a fixed amount of storage resource $m^k$ on a node $v$ if $k$ is to be processed on this node later. Thus, only if both computation and storage resources on a node are sufficient, a request could be processed on that node. Let variable $\nu^k_v$ denote the fraction of computation capacity $D_v$ sliced for the piece of request $k$. The processing of user requests is also described by an $M|\mathcal{M}|1$ model. Let $T^k_p$ denote the processing latency of edge computing node $v$ for request $k$. Then, based on the computational capacity $\nu^k_v D_v$ with an amount $q^k \lambda t$ to be served, $\forall k, \forall v, T^k_p$ is expressed as:

$$T^k_p = \begin{cases} \frac{1}{\nu^k_v D_v - \eta^k q^k \lambda t}, & \text{if } q^k > 0, \\ 0, & \text{otherwise,} \end{cases} \hspace{1cm} (7)$$

where $\nu^k_v$ is the fraction of node $v$’s computation capacity sliced to request $k$, and $\eta^k$ is the processing density [12] of request $k$ measured in “cycles/bit”. In the above equation, when request $k$ is not processed on node $v$, the latency is set to 0 and, at the same time, no computation resource should be allocated to request $k$. The corresponding constraint is:

$$\begin{cases} \eta^k q^k \lambda t \nu^k_v < r^k D_v, & \text{if } q^k > 0, \\ r^k = 0, & \text{otherwise.} \end{cases} \hspace{1cm} (8)$$

$q^k$ and $r^k$ also have to fulfill the consistency constraints:

$$\sum_{v \in V} q^k = \sum_{t=0}^\infty \sum_{k \in K} \nu^k_v \frac{\nu^k_v D_v}{\eta^k q^k \lambda t}, \forall k. \hspace{1cm} (9)$$

Remind that the right hand of equation (9) represents whether a request $k$ is admitted in the system or not. If a request $k$ is rejected by the admission controller, the right hand expression is equal to 0 and $q^k = 0$ is enforced.

The storage constraint can be expressed as follows:

$$\sum_{k \in K} m^k \leq S_v, \forall v. \hspace{1cm} (10)$$

Different requests $k \in K$ may share an edge computing node at a time slot. Thus, the reservation constraint of a node computation capacity at any time slot is implemented by:

$$\sum_{k \in K} r^k_{k,t} \leq 1, \forall v, \forall t, \hspace{1cm} (11)$$

where $r^k_{k,t}$ is the fraction of node $v$’s computation capacity allocated for request $k$ at time slot $t$. We assume that the reserved computation power for each request over its life period will not change due to both the computation scaling
overhead and task reconfiguration overhead. The superscript \( t \) in \( r^{vkt} \) allows us to keep track of the life status of the request. The relation between \( r^{vkt} \) and \( r^{vk} \) is given by \( r^{vkt} = \rho^{vkt,kv}r^{vk} \), where \( \rho^{vkt,kv} \) is a binary variable which is equal to 1 if \( \xi^{k}t + \left[ \frac{T^{k}}{t} \right] \leq t < \xi^{k}o \), and 0, otherwise.

\[ F_{k'} \] is determined by the function check_overlap(\( \cdot \)) (line 6 of Algorithm 1). This function takes as input \( k, k' \) and the partial solution \( S^* \) computed up to the current point, returns \( F_{k'} \) and proceeds as follows: i) it initializes two local variables \( \alpha \) and \( \beta \) with the arrival time \( \alpha^{k} \) and deadline \( \beta^{k} \) of request \( k' \), respectively; ii) it verifies if \( k' \) is admitted; if yes, it updates, respectively, \( \alpha \) and \( \beta \) with the exact starting time \( \xi^{k}o \) and ending time \( \max_{v \in V} \xi^{k}o \) of \( k' \) according to the solution \( S^* \); iii) it computes the (partial) overlapping between \( k \) and \( k' \) as: \( \text{overlap} = \beta^{k} - \alpha^{k} \), if \( \alpha^{k} > \beta^{k} \); \( \text{overlap} = \beta - \alpha^{k} \), otherwise (a negative value of overlap means no overlapping); iv) Finally, it calculates and returns the maximum relative overlap value \( F_{k'} \) between \( k \) and \( k' \), which is expressed as: \( \min(\frac{\max(\text{overlap}, 0)}{\beta^{k} - \alpha^{k}}) \).

Based on \( \{ F_{k'} | k' \in K \backslash \{ k \} \} \), for each edge node \( v \in V \), we select the maximum \( F_{k} \) for all \( k' \) being processed at \( v \), and we identify this overlap value with \( C_{v} \) (line 7). Next, we compute the ordered set \( Q^{k} \) which contains sets \( V_{i} \) of best candidate edge nodes to process request \( k \). In doing so, we consider \( C_{v} \) and limit to \( L_{v} \) the computation resource of each node in \( V_{i}, \forall v \in Q^{k} \) (line 8 of Algorithm 1; details in Algorithm 2). If we successfully find some candidates (\( Q^{k} \neq \emptyset \)), we further update the residual bandwidth \( B'_{e} \) for all links \( e \) and create a weighted graph \( G' \) with the reciprocal bandwidth \( B^{-1}_{e} \). Then (lines 12-16), we select the first \( V_{i} \in Q^{k} \) that permits to find a profitable solution \( (O \geq 0) \) according to

### Algorithm 1 Sequential fixing and scheduling

1. Initialize \( z^{kt} = 0, \forall k, t \) and profit \( O = 0 \) for \( P1 \);
2. Set \( \hat{z}^{vk} = 0, \hat{r}^{vk} = 0, \hat{p}^{vk} = 0 \) (in \( S^* \)), \( \forall v, e, \gamma \);
3. Sort \( K \) in descending order as \( K_{\alpha} : \frac{\sum_{k \in K} \alpha^{k} \mu^{k}}{\sum_{k \in K} \mu^{k}} \), \( k \in K \);
4. for \( k \in K_{\alpha} \) do
5. Reset admission \( z^{kt} = 1, \forall t \in [\alpha^{k}, \beta^{k} - d^{k}] ;
6. \( F_{k'} \leftarrow \text{checkoverlap}(k', S^{*}) \), \( \forall k' \in K \backslash \{ k \} \);
7. \( C_{v} \leftarrow \max_{k' \in K \backslash \{ k \}} \{ \hat{z}^{k'v} F_{k'} \} \), \( \forall v \in V_{i} \);
8. \( Q^{k}_{i}, L_{v} \leftarrow \text{findcandidates}(k, G', S^{*}, C_{v}) ;
9. if \( Q^{k}_{i} \neq \emptyset \) then
10. \( B'_{e} \leftarrow B_{e}(1 - \sum_{(k',v) \in E_{v}} F_{k'}) \), \( \forall v \in E \);
11. Create graph \( G' \) weighted by \( B'_{e} ;
12. for \( v \in Q^{k}_{i} \ do
13. \text{Set } \hat{z}^{k'v} = 1, \hat{r}^{k'v} \leq L_{v} \, \forall v \in V_{i} ;
14. \text{Fix route } (\hat{z}^{k'v}) \text{ using Dijkstra} ;
15. \text{Optimize } P1 \text{ to get profit } O \text{ and solution } S_{i} ;
16. if \( O \geq 0 \) then break; \( \triangleright P1 \) is feasible
17. if \( O \geq 0 \) \& \( Q^{k}_{i} \neq \emptyset \) then
18. Update \( O* \leftarrow O, S* \leftarrow S_{i} ;
19. Admit \( k \) and allocate resources based on \( S^{*} ;
20. Set \( P1 \) ’s lower bound \( LB = O^{*} ;
21. else \text{Reject } k \text{ (set } z^{kt} = 0, \forall k, t \);
Algorithm 2 Find candidates

Input: \( k, \mathcal{G}, \mathcal{S}^* \) (solution), \( C_v, \alpha, \beta \) (conflict);
Output: \( \mathcal{Q}^k \) (candidates), \( L_v, \mathcal{V}_v \) (limit);

1: \( L_v \leftarrow 1 - \sum_{k' \in \mathcal{K}} C_{v} \leq w_{c} \) \( \forall v \in \mathcal{V} \); \( b_{c} = 0.6 \)
2: \( V^* = \{ v \in \mathcal{V} | C_v \leq w_c \parallel L_v \geq w_{d} \} ; \) \( b_{d} = 0.25 \)
3: \( \mathcal{V}_v \leftarrow (-\text{hop}(\mathcal{G}, s^k, v), v \notin \{ s^k | k' \in \mathcal{K} \}, L_v), \forall v \in V^* ; \)
4: Sort \( \mathcal{V}^* \) in descending order by \( \mathcal{V}_v \);
5: \( \mathcal{V}_1 = \varnothing, D_0 = 0 ; \)
6: for \( v \in \mathcal{V}^* \) do
7: \( \text{if } \lambda^k \geq w_{d}D_2 \) then \( \mathcal{V}_1 \leftarrow \mathcal{V}_1 \cup \{ v \} ; \) \( b_{d} = 0.9 \)
8: \( D_2 \leftarrow D_2 + D_v, L_v ; \)
9: \( \mathcal{Q}^k = (\mathcal{V}_1) \cup \{ (v, v \in \mathcal{V}^* - \mathcal{V}_1) ; \)

in the 2nd place and the deadline \( \beta^k \) in the 3rd (last) place. For each request \( k \), we try to guarantee sufficient computation power by using its closest neighbor nodes with a condition \( \frac{\lambda^k}{D_0} \leq w_{\text{greedy}}, w_{\text{greedy}} = 0.6 \) is a threshold appropriately chosen based on our experiments.

### IV. Numerical Results

We evaluate the performance of the proposed model, the SFS and Greedy heuristics in terms of the operator’s profit, expressed as in \( \mathcal{P}_0 \), the serving rate (the fraction of admitted requests) and the computing time to get the solution. We first present the experimental setup and then we discuss our results.

#### Experimental Setup:

All numerical results presented in this section have been obtained on a server equipped with an Intel(R) Xeon(R) E5-2640 v4 CPU @ 2.40GHz and 126 GB of RAM and with an open-source framework, SCIP (Solving Constraint Integer Programs). The parameters of SCIP used in our experiments are set to their default values. The results illustrated in Figure 2 are obtained by averaging over 50 instances, with 97% narrow confidence intervals.

The network topologies used in our experiments are generated based on Erdős-Rényi random graph by specifying the numbers of nodes and edges. Due to space constraints, we present and discuss in this section the results obtained for a representative topology (denoted as \( 30N5E3R \)), composed of 30 nodes and 50 edges with 30 requests, as well as those for a small topology (denoted as \( 5N3E3R \)) consisting of 5 nodes, 5 edges, and 3 incoming requests. The \( 5N3E3R \) topology allows us to compare the SFS heuristic to the optimal solution (Figure 2(d)). The full set of results is available online.

We uniformly extract, at random, a source node as well as the arrival/ending times and duration, and the revenue gained by the operator in serving each request, in range \([100, 300]\). We further generate random request rates according to a Gaussian distribution \( N(\lambda^k, \sigma^2) \), where \( \lambda^k \) is randomly selected in range from 30 to 60 Gb/s and \( \sigma = 0.5 \). For the sake of simplicity, we assume that all links have the same bandwidth (\( B_c = 30 \) Gb/s) and nodes have the same computation capacity (\( D_v = 30 \) Giga cycles/s) and storage capacity (\( S_c = 40 \) GB). The costs of using one unit of these three resources, \( \psi_e, \theta_e \), and \( \varphi_e \), are all set to 0.01. Finally, we set the processing density \( \eta^k = 1 \) and the storage requirement \( m^k = 10 \) for all requests. Note that our proposed model and heuristics are general, and can be applied to optimize resource allocation in all network scenarios with any parameters setting.

#### Effect of the request rate and revenue

\( (\lambda^k, \mu^k) \): Figures 2(a) and 2(b) illustrate the variations of profit and serving rate versus the request rate and revenue in the \( 30N5E3R \) topology. Values of \( \lambda^k \) and \( \mu^k, k \in \mathcal{K} \) are both scaled from 0.5 to 2.0 with respect to their initial values. This implicitly indicates that serving each request provides a revenue proportional to its arrival rate. As \( (\lambda^k, \mu^k) \) increase, the profits for all approaches increase; the network operator, in fact, is able to select and admit the requests which can cover the system cost and provide higher profit at the same time. When the request rate is low, all connections can be served; when it increases,
specifically after the point around 1.2, the serving rate of SFS decreases since the system can accommodate less requests, which become more demanding, hence costlier in terms of required resources, as the scale factor (thus $\lambda^k$) increases. Finally, SFS exhibits better performance compared to Greedy with gaps up to 18% for profit and 20% for serving rate.

**Effect of the computation capacity $D_v$:** Figure 2(c) shows the variations of the profit against the edge node computation capacity $D_v$, scaled with respect to its initial value from 0.5 to 1.5, in the $30N50E30R$ topology. When $D_v$ increases, the profit (and serving rate, not shown for space reasons) increases and converge to a specific value for all approaches. Note that, for SFS, the profit increases from 700 up to 980, while for Greedy, from about 630 to 950. As for the serving rate it increases for SFS from 0.64 up to 1, while for Greedy, from about 0.61 to 0.98. These trends reflect the strong effect of the available computation capacity on the profit and serving rate. Additionally, SFS performs better than Greedy with clear gaps: up to 13% for profit and 14% for serving rate. As expected, with large available node computation capacity, the performance gap between SFS and Greedy decreases since the utilization of enhanced algorithms is less critical to perform a good resource allocation, when resources are abundant.

**Optimum and Computing time:** The exact model $\mathcal{P}0$ could be solved in a reasonable time only in the small topology ($5N5E3R$), and Figure 2(d) plots the profit versus the request rate $\lambda^k$ keeping the revenue $\mu^k$ fixed. The decreasing trend of the profit for all the three approaches, when increasing $\lambda^k$, is due to the fact that more and more resources are needed and hence the cost incurred by the operator continues to increase while the revenue is fixed, and therefore the profit decreases. SFS exhibits indeed very good performance since its curve and that of the optimal solution are completely overlapping, while the Greedy approach shows lower performance.

The optimal solution has an average computing time of 146 s, while SFS took just 5 s, and Greedy 4 s. Furthermore, in larger scenarios (up to $30N50E30R$) SFS exhibited a computing time always inferior to 1096 s, thus confirming its efficiency in computing very good solutions in a short time. The Greedy approach needs less computation time, on average 822 s, to obtain the solution, at the cost of higher performance gaps with the proposed model and SFS heuristic.

**V. Conclusion**

In this paper we formulated and solved the resource calendaring problem in mobile networks equipped with Mobile Edge Computing (MEC) capabilities. Specifically, we proposed both an exact optimization model as well as an effective heuristic able to obtain near-optimal solution in all the considered, real-size network scenarios.

The decisions we optimized include admission control for the connections offered to the network, their calendaring (scheduling) and bandwidth constrained routing, as well as the determination of which nodes provide the required computation and storage capacity. Calendaring, in particular, permits to exploit the intrinsic flexibility in the services demanded by different users, whose starting time can be shifted without penalizing the utility perceived by the user while, at the same time, permitting a better resource utilization in the network.

**ACKNOWLEDGMENT**

This research was supported by the H2020-MSCA-ITN-2016 SPOTLIGHT under grant agreement number 722788.

**REFERENCES**


