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 $\mathcal{G}(\mathcal{V},\mathcal{E})$, where each node $v \in \mathcal{V}$ represents an edge *computing node* having D_v and S_v as *computation* and *storage capacity*, respectively. The two parameters θ_v and ϕ_v denote, respectively, the *cost* of computation and storage capacity of node v. Each *edge* $e \in \mathcal{E}$ corresponds to a network link characterized by its *bandwidth* B_e and its *cost per unit of flow* ψ_e. Let $\mathcal{K}$ 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 τ. Each request $k \in \mathcal{K}$ 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; α^k, β^k and d^k define the *arrival 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 λ^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.

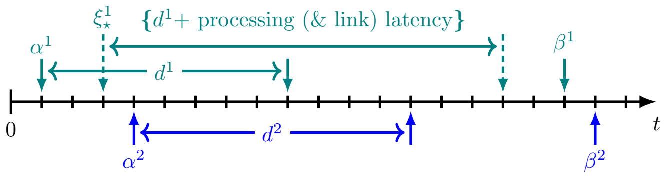


Fig. 1: Example of time scheduling of a request.

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 β^k. Figure 1 shows the arrival time α, deadline β and duration d of requests 1 and 2. Also, it highlights that request 1 is scheduled to be served from time $\xi^1_\star$, delayed (shifted) with respect to α^1 but still compatible with β^1. The ending time for the request will then depend on $\xi^1_\star$, d^1, 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 α^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 $\xi^k_\star$ can vary in the time frame $[\alpha^k, \beta^k - d^k]$, we express $\xi^k_\star$ as: $\xi^k_\star = \sum_{t=\alpha^k}^{\beta^k-d^k} t \cdot z^{kt}$, and we have:

$$\sum\nolimits_{t=\alpha^k}^{\beta^k-d^k} z^{kt} \leqslant 1, \ \forall k. \tag{1}$$

Essentially, z^{kt} is a binary variable that can be 1 at most in one point of time which corresponds to $\xi^k_\star$ 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 $q^{kv} \in [0,1]$ of request k, the ending time at v, denoted by ξ^{kv}_o, can be expressed as: $\xi^{kv}_o = \xi^k_\star + d^k + \left\lceil \frac{T^{kv}_L}{\tau} \right\rceil + \left\lceil \frac{T^{kv}_P}{\tau} \right\rceil$, where T^{kv}_L and T^{kv}_P are respectively the link latency and processing latency. Note that both $\xi^k_\star$ and ξ^{kv}_o are integer values in the *time slot* set and τ is the time-slot duration. The ending time of each request

[1]http://xiang.faculty.polimi.it/files/TechnicalReport.pdf

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

$$\max_{v\in\mathcal{V}}\{\xi_o^{kv}\} \leqslant \beta^k,\ \forall k. \tag{2}$$

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^{kv} (remind that q^{kv} is the fraction of request k to be processed at node v). Then, the total flow of request k on link e, f_e^k, can be expressed as the sum of all pieces of k that pass through such link: $f_e^k = \sum_{v\in\mathcal{V}:\ e\in\mathcal{R}^{kv}} q^{kv}$, where $\mathcal{R}^{kv} \subset \mathcal{E}$ denotes the routing path (set of traversed links) for the partial request q^{kv} from source node s^k to node v. The traffic flow conservation constraint is enforced by:

$$\sum_{e\in\Phi_v^-} f_e^k - \sum_{e\in\Phi_v^+} f_e^k = \begin{cases} q^{kv}-1, & \text{if } v = s^k, \\ q^{kv}, & \text{otherwise,} \end{cases} \quad \forall k, \forall v, \tag{3}$$

where Φ_v^- and Φ_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_L^{kv} 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 $M|M|1$ model; hence, $\forall k, \forall v$, T_L^{kv} is defined as:

$$T_L^{kv} = \begin{cases} \sum\limits_{e\in\mathcal{R}^{kv}} \frac{1}{p_e^{kv}B_e - q^{kv}\lambda^k}, & \text{if } q^{kv} > 0\,\&\, v \neq s^k, \\ 0, & \text{otherwise,} \end{cases} \tag{4}$$

where p_e^{kv} is the fraction of bandwidth capacity sliced for the piece of request (q^{kv}) 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. $q^{kv} > 0$) 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:

$$\begin{cases} q^{kv}\lambda^k < p_e^{kv}B_e, & \text{if } e \in \mathcal{R}^{kv}, \\ p_e^{kv} = 0, & \text{otherwise,} \end{cases} \quad \forall k, \forall v, \forall e. \tag{5}$$

Considering that different requests $k \in \mathcal{K}$ can share the same link at a given time slot, the reservation constraint of a link capacity at any time slot is expressed as:

$$\sum_{k\in\mathcal{K}}\sum_{v\in\mathcal{V}} p_e^{kvt} \leqslant 1,\ \forall e, \forall t, \tag{6}$$

where p_e^{kvt} is the fraction of link e's bandwidth allocated for a piece of request q^{kv} 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_e^{kvt} is used to indicate the life status of the flow. The relation between p_e^{kvt} and p_e^{kv} is given by $p_e^{kvt} = \delta^{kvt}p_e^{kv}$, where δ^{kvt} is a binary variable which is equal to 1 if $\xi_\star^k \leqslant t < \xi_\star^k + d^k + \left\lceil \frac{T_L^{kv}}{\tau} \right\rceil$, 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 r^{kv} denote the fraction of computation capacity D_v sliced for the piece of request $q^{kv}\lambda^k$. The processing of user requests is also described by an $M|M|1$ model. Let T_P^{kv} denote the *processing latency* of edge computing node v for request k. Then, based on the computational capacity $r^{kv}D_v$ with an amount $q^{kv}\lambda^k$ to be served, $\forall k, \forall v$, T_P^{kv} is expressed as:

$$T_P^{kv} = \begin{cases} \frac{1}{r^{kv}D_v - \eta^k q^{kv}\lambda^k}, & \text{if } q^{kv} > 0, \\ 0, & \text{otherwise,} \end{cases} \tag{7}$$

where r^{kv} is the fraction of node v's computation capacity sliced to request k, and η^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:

$$\begin{cases} \eta^k q^{kv}\lambda^k < r^{kv}D_v, & \text{if } q^{kv} > 0, \\ r^{kv} = q^{kv} = 0, & \text{otherwise.} \end{cases} \tag{8}$$

q^{kv} and r^{kv} also have to fulfill the consistency constraints:

$$\sum\nolimits_{v\in\mathcal{V}} q^{kv} = \sum\nolimits_{t=\alpha^k}^{\beta^k-d^k} z^{kt},\ \forall k. \tag{9}$$

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 $q^{kv} = 0$ is enforced.

The storage constraint can be expressed as follows:

$$\sum_{k\in\mathcal{K}:\ q^{kv}>0} m^k \leqslant S_v,\ \forall v. \tag{10}$$

Different requests $k \in \mathcal{K}$ 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:

$$\sum\nolimits_{k\in\mathcal{K}} r^{kvt} \leqslant 1,\ \forall v, \forall t, \tag{11}$$

where r^{kvt} 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^{kvt} allows us to keep track of the life status of the request. The relation between r^{kvt} and r^{kv} is given by $r^{kvt} = \rho^{kvt} r^{kv}$, where ρ^{kvt} is a binary variable which is equal to 1 if $\xi_\star^k + \left\lceil \frac{T_L^{kv}}{\tau} \right\rceil \leqslant t < \xi_o^{kv}$, and 0, otherwise.

F. Optimization Problem

Our goal in the *resource calendaring* problem is to maximize the profit computed as the total revenue obtained from serving the users' requests minus the network operation costs in terms of computation, storage and bandwidth resources, under the constraints (starting and ending times) of requests coming from different nodes:

$$\max \sum_{t\in\mathcal{T}} \sum_{k\in\mathcal{K}} \Big\{ \mu^k z^{kt} - \sum_{v\in\mathcal{V}} \Big\{ r^{kvt} D_v \theta_v + \rho^{kvt} m^k \phi_v + \sum_{e\in\mathcal{E}} p_e^{kvt} B_e \psi_e \Big\} \Big\},$$

$$\text{s.t.} \quad (1) - (11), \qquad (\mathcal{P}0)$$

where μ^k is the revenue gained from serving request k. Problem $\mathcal{P}0$ contains both nonlinear and indicator constraints, therefore, it is a mixed-integer nonlinear programming (MINLP) problem, which is hard to be solved directly [13]. Moreover, we also face the following difficulties: a) routing variables $\mathcal{R}^{kv}$ and request fraction variables q^{kv} are "intertwined": to find the optimal routing, the fraction of request processed at each node v should be known, and at the same time, to solve the optimal resource allocation for a request, the routing path should be known; b) $\mathcal{P}0$ contains indicator functions and constraints, which cannot be directly and easily processed by most solvers. To deal with the above critical issues, we propose an equivalent reformulation of $\mathcal{P}0$, which we call $\mathcal{P}1$ that we can efficiently solve with the Branch and Bound method.

Intuitively, the reformulation in $\mathcal{P}1$ proceeds as follows: (a) we first reformulate the link and processing latency constraints, as well as the node storage constraint (viz., constraints (4), (7) and (10)), then (b) we handle the difficulties related to indicator variables δ^{kvt}, ρ^{kvt} and variables $\mathcal{R}^{kv}$ with the corresponding routing constraints. For space reasons, we do not include the reformulation here. The interested readers can refer to the technical report available online[1].

III. Heuristics

To solve our problem in a reasonable computational time, we propose a heuristic named *Sequential Fixing and Scheduling (SFS)* which realizes a good trade-off between admitting "valuable" connections (the ones that provide high return to the service provider) and the resources they request in terms of transmission rate, storage and computation.

SFS is detailed in Algorithm 1. We first introduce some auxiliary variables of $\mathcal{P}1$, viz., b^{kv}: whether request k is processed on node v, and γ_e^{kv}: whether request piece q^{kv} is routed via link e. The hat notation (like $\hat{b}^{kv}$) represents the values of the corresponding variables in the solution set $\mathcal{S}^\star$. We start by sorting all requests in descending order according to the ratio $\frac{\mu^k}{d^k \lambda^k m^k \eta^k}$; this ranking is designed to give a higher weight to requests that generate more revenue and less cost to the operator. Then, we try to define a schedule where we admit as many requests as possible. For each request k, we check whether its activation period overlaps with the one of other requests k' that are already admitted, and in such case we say there is a *conflict*. The overlap value $F_{k'}$ is determined by the function $check\_overlap(\cdot)$ (line 6 of Algorithm 1). This function takes as input k, k' and the partial solution $\mathcal{S}^\star$ computed up to the current point, returns $F_{k'}$ and proceeds as follows: i) it initializes two local variables α and β 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, α and β with the exact starting time $\hat{\xi}_\star^{k'}$ and ending time $\max_{v\in\mathcal{V}} \hat{\xi}_o^{k'v}$ of k' according to the solution $\mathcal{S}^\star$; iii) it computes the (partial) overlapping between k and k' as: $overlap = \beta^k - \alpha$, if $\alpha > \alpha^k$; $overlap = \beta - \alpha^k$, 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: $\min(\frac{\max(overlap,\, 0)}{\min(\beta^k - \alpha^k,\, \beta - \alpha)},\, 1)$.

Based on $\{F_{k'} | k' \in \mathcal{K}\backslash\{k\}\}$, for each edge node $v \in \mathcal{V}$, 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 $\mathcal{Q}^k$ which contains sets $\mathcal{V}_i$ 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 $\mathcal{V}_i, \forall \mathcal{V}_i \in \mathcal{Q}^k$ (line 8 of Algorithm 1; details in Algorithm 2). If we successfully find some candidates ($\mathcal{Q}^k \neq \varnothing$), we further update the residual bandwidth B'_e for all links e and create a weighted graph $\mathcal{G}'$ with the reciprocal bandwidth $B_e'^{-1}$. Then (lines 12-16), we select the first $\mathcal{V}_i$ in $\mathcal{Q}^k$ that permits to find a profitable solution ($\mathcal{O} \geqslant 0$) according to

Algorithm 1 *Sequential fixing and scheduling*

```
 1: Initialize $z^{kt} = 0, \forall k, t$ and profit $\mathcal{O} = 0$ for $\mathcal{P}1$;
 2: Set $\hat{b}^{kv} = 0, \hat{r}^{kv} = 0, \hat{p}_e^{kv} = 0$ (in $\mathcal{S}^\star$), $\forall k, v, e$;
 3: Sort $\mathcal{K}$ in descending order as $\mathcal{K}_r$ by $\frac{\mu^k}{d^k\lambda^k m^k \eta^k}, k \in \mathcal{K}$;
 4: for $k \in \mathcal{K}_r$ do
 5:     Reset admission $z^{kt} \leqslant 1,\ t \in [\alpha^k, \beta^k - d^k]$;
 6:     $F_{k'} \leftarrow check\_overlap(k, k', \mathcal{S}^\star),\ \forall k' \in \mathcal{K}\backslash\{k\}$;
 7:     $C_v \leftarrow \max_{k'\in\mathcal{K}\backslash\{k\}} \{\hat{b}^{k'v} F_{k'}\},\ \forall v \in \mathcal{V}$;
 8:     $\mathcal{Q}^k, L_v \leftarrow find\_candidates(k, \mathcal{G}, \mathcal{S}^\star, C_v)$;
 9:     if $\mathcal{Q}^k \neq \varnothing$ then
10:         $B'_e \leftarrow B_e(1 - \sum_{(k',v)\in\mathcal{K}\times\mathcal{V}|C_v>0} \hat{p}_e^{k'v}),\ \forall e \in \mathcal{E}$;
11:         Create graph $\mathcal{G}'$ weighted by $B_e'^{-1}$;
12:         for $\mathcal{V}_i \in \mathcal{Q}^k$ do
13:             Set $b^{kv} = 1, r^{kv} \leqslant L_v,\ \forall v \in \mathcal{V}_i$;
14:             Fix route ($\gamma_e^{kv}$) using Dijkstra;
15:             Optimize $\mathcal{P}1$ to get profit $\mathcal{O}$ and solution $\mathcal{S}$;
16:             if $\mathcal{O} \geqslant 0$ then break; ▷ $\mathcal{P}1$ is feasible
17:     if $\mathcal{O} \geqslant \mathcal{O}^\star$ & $\mathcal{Q}^k \neq \varnothing$ then
18:         Update $\mathcal{O}^\star \leftarrow \mathcal{O},\ \mathcal{S}^\star \leftarrow \mathcal{S}$;
19:         Admit $k$ and allocate resources based on $\mathcal{S}^\star$;
20:         Set $\mathcal{P}1$'s lower bound $LB = \mathcal{O}^\star$;
21:     else Reject $k$ (set $z^{kt} = 0, \forall k, t$);
```

Algorithm 2 *Find candidates*

```
    Input: $k$, $\mathcal{G}$, $\mathcal{S}^\star$(solution), $C_v$(conflict);
    Output: $\mathcal{Q}^k$(candidates), $L_v$(limit);
 1: $L_v \leftarrow 1 - \sum_{k' \in \mathcal{K} | C_v > w_c} \hat{r}^{k'v}$, $\forall v \in \mathcal{V}$; $\triangleright$ $w_c = 0.6$
 2: $\mathcal{V}^s = \{v \in \mathcal{V} \mid C_v \leqslant w_c \,||\, L_v \geqslant w_l\}$; $\triangleright$ $w_l = 0.25$
 3: $\mathcal{Y}_v \leftarrow (-hop(\mathcal{G}, s^k, v), v \notin \{s^{k'} | k' \in \mathcal{K}\}, L_v)$, $\forall v \in \mathcal{V}^s$;
 4: Sort $\mathcal{V}^s$ in descending order by $\mathcal{Y}_v$;
 5: $\mathcal{V}_1 = \varnothing$, $D_\Sigma = 0$;
 6: for $v \in \mathcal{V}^s$ do
 7:     if $\lambda^k \geqslant w_d D_\Sigma$ then $\mathcal{V}_1 \leftarrow \mathcal{V}_1 \cup \{v\}$; $\triangleright$ $w_d = 0.9$
 8:     $D_\Sigma \leftarrow D_\Sigma + D_v L_v$;
 9: $\mathcal{Q}^k = (\mathcal{V}_1) \cup (\{v\}, v \in \mathcal{V}^s - \mathcal{V}_1)$;
```

the following criteria: we outsource k to the nodes in $\mathcal{V}_i$ and bound the computation resource by setting b^{kv} and r^{kv}. Based on $\mathcal{G}'$, we route each piece of request q^{kv} using the *Dijkstra* algorithm (lines 10-14). After fixing variables b^{kv}, γ_e^{kv} and the constraints related to z^{kt}, r^{kv} in $\mathcal{P}1$, we start to optimize $\mathcal{P}1$ to get the profit and the solution denoted, respectively, by $\mathcal{O}$ and $\mathcal{S}$. If $\mathcal{P}1$ results infeasible in the current setting ($\mathcal{O} < 0$), we reiterate on the other elements of $\mathcal{Q}^k$. If the result of the new optimization improves, we update the current best profit $\mathcal{O}^\star$ and solution $\mathcal{S}^\star$, we admit request k and allocate resources to it (including time slots, computation, bandwidth, and storage). We also update the lower bound of $\mathcal{P}1$ to $LB = \mathcal{O}^\star$ to accelerate the optimization (line 20). Finally, if the result does not improve or no candidate could be found, we reject k and clear its corresponding variables settings.

In Algorithm 2, we first estimate L_v, the remaining computation capacity of each node v, based on $\hat{r}^{k'v}$ in the solution $\mathcal{S}^\star$ verifying that the conflicts are higher than a given threshold w_c ($C_v > w_c$). Then, we define $\mathcal{V}^s$ as the set of nodes v satisfying $C_v \leqslant w_c \,||\, L_v \geqslant w_l$, where w_l is a threshold on the remaining computation. Hence, $\mathcal{V}^s$ represents the set of nodes that are either in less conflict (for request k) or have enough remaining computation power. For each $v \in \mathcal{V}^s$, we compute three features (denoted by $\mathcal{Y}_v$), i.e., the negative hop distance between s^k and v ($-hop(\mathcal{G}, s^k, v)$), the indicator of whether v is a source node or not ($v \notin \{s^{k'} | k' \in \mathcal{K}\}$) and the estimated left computation capacity L_v (lines 1-3). Based on $\mathcal{Y}_v$, we sort $\mathcal{V}^s$ in descending order to give more priority to a node that is closer to the ingress node for k, better not to be a source, and has more remaining computation capacity with respect to other nodes. Then, we try to add nodes to $\mathcal{V}_1$, in order, until $\lambda^k \geqslant w_d D_\Sigma$, where D_Σ denotes the estimated total computation capacity that can be used and w_d is a threshold controlling the total required computation capacity. Finally, we return the ordered set $\mathcal{Q}^k$ with $\mathcal{V}_1$ at the first place and the left nodes $\mathcal{V}^s - \mathcal{V}_1$ being separately stored as unit sets of backup candidates (lines 5-9). Notice that the values of the thresholds $w_{c/l/d}$ are appropriately chosen based on our experiments.

Greedy approach: We further present an alternative benchmark heuristic, which we call *Greedy*. It first sorts the requests in ascending order by the priority key ($-\mu^k$, α^k, β^k) and then tries to schedule them one by one. The sorting considers the revenue μ^k of a request in the 1st place, the arrival time α^k in the 2nd place and the deadline β^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 $\frac{\lambda^k}{D_\Sigma} \leqslant w_{greedy}$, $w_{greedy} = 0.6$ is a threshold appropriately chosen based on our experiments.

IV. Numerical Results

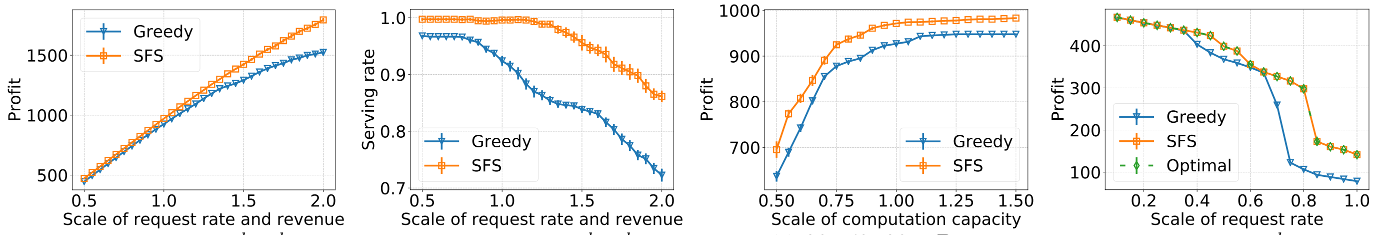
We evaluate the performance of the proposed model, the SFS and Greedy heuristics in terms of the operator's profit, expressed as in $\mathcal{P}0$, the serving rate (the fraction of admitted requests) and the computing time to get the solution. We first present the experimental setup and then we discuss 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 Erdös-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 *30N50E30R*), composed of 30 nodes and 50 edges with 30 requests, as well as those for a small topology (denoted as *5N5E3R*) consisting of 5 nodes, 5 edges, and 3 incoming requests. The *5N5E3R* topology allows us to compare the SFS heuristic to the optimal solution (Figure 2(d)). The full set of results is available online[1].

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 λ^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 ($B_e = 30$ Gb/s) and nodes have the same computation capacity ($D_v = 30$ Giga cycles/s) and storage capacity ($S_v = 40$ GB). The costs of using one unit of these three resources, ψ_e, θ_v, and ϕ_v, 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 (λ^k, μ^k)**:** Figures 2(a) and 2(b) illustrate the variations of profit and serving rate versus the request rate and revenue in the *30N50E30R* topology. Values of λ^k and μ^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 (λ^k, μ^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,



(a) 30N50E30R (λ^k, μ^k) (b) 30N50E30R (λ^k, μ^k) (c) 30N50E30R D_v (d) 5N5E3R λ^k

Fig. 2: Profit and serving rate against scaling parameters (λ^k, μ^k), D_v and λ^k.

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 λ^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) increase 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 λ^k keeping the revenue μ^k fixed. The decreasing trend of the profit for all the three approaches, when increasing λ^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

[1] C.-F. Liu, M. Bennis, M. Debbah, and H. V. Poor, “Dynamic task offloading and resource allocation for ultra-reliable low-latency edge computing,” *IEEE Trans. Commun.*, 2019.

[2] P. Wang, Z. Zheng, B. Di, and L. Song, “Joint task assignment and resource allocation in the heterogeneous multi-layer mobile edge computing networks,” in *IEEE GLOBECOM*, 2019.

[3] J. Zhang, A. Sinha, J. Llorca, A. Tulino, and E. Modiano, “Optimal control of distributed computing networks with mixed-cast traffic flows,” in *IEEE INFOCOM*, 2018, pp. 1880–1888.

[4] M. Dufour, S. Paris, J. Leguay, and M. Draief, “Online bandwidth calendaring: On-the-fly admission, scheduling, and path computation,” in *IEEE ICC*, 2017.

[5] L. Wang, F. Zhang, K. Zheng, A. V. Vasilakos, S. Ren, and Z. Liu, “Energy-efficient flow scheduling and routing with hard deadlines in data center networks,” in *IEEE 34th Int. Conference on Distributed Computing Systems*, 2014, pp. 248–257.

[6] H. Tan, Z. Han, X.-Y. Li, and F. C. Lau, “Online job dispatching and scheduling in edge-clouds,” in *IEEE INFOCOM*, 2017, pp. 1–9.

[7] J. Meng, H. Tan, X.-Y. Li, Z. Han, and B. Li, “Online deadline-aware task dispatching and scheduling in edge computing,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 6, pp. 1270–1286, 2020.

[8] V. Farhadi, F. Mehmeti, T. He, T. La Porta, H. Khamfroush, S. Wang, and K. S. Chan, “Service placement and request scheduling for data-intensive applications in edge clouds,” in *IEEE INFOCOM*, 2019.

[9] B. Xiang, J. Elias, F. Martignon, and E. Di Nitto, “Joint network slicing and mobile edge computing in 5G networks,” in *IEEE ICC*, 2019.

[10] ——, “Joint planning of network slicing and mobile edge computing in 5G networks,” *arXiv preprint arXiv:2005.07301*, 2020.

[11] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, “Achieving high utilization with software-driven wan,” in *ACM SIGCOMM Comp. Comm. Rev.*, vol. 43, 2013, pp. 15–26.

[12] J. Kwak, Y. Kim, J. Lee, and S. Chong, “DREAM: Dynamic resource and task allocation for energy minimization in mobile cloud systems,” *IEEE JSAC*, vol. 33, no. 12, pp. 2510–2523, 2015.

[13] R. Kannan and C. L. Monma, “On the computational complexity of integer programming problems,” in *Optimization and Operations Research*, Springer, 1978, pp. 161–172.