Efficient Placement of Multi-Component Applications in Edge Computing Systems

Tayebeh Bahreini
Dept. of Computer Science
Wayne State University
Detroit, Michigan 48202
tayebeh.bahreini@wayne.edu

Daniel Grosu
Dept. of Computer Science
Wayne State University
Detroit, Michigan 48202
dgrosu@wayne.edu

ABSTRACT

Mobile Edge Computing (MEC) is a new paradigm which has been introduced to solve the inefficiencies of mobile cloud computing technologies. The key idea behind MEC is to enhance the capabilities of mobile devices by forwarding the computation of applications to the edge of the network instead of to a cloud data-center. One of the main challenges in MEC is determining an efficient placement of the components of a mobile application on the edge servers that minimizes the cost incurred when running the application. In this paper, we address the problem of multi-component application placement in edge computing by designing an efficient heuristic on-line algorithm that solves it. We also present a Mixed Integer Linear Programming formulation of the multi-component application placement problem that takes into account the dynamic nature of users' location and the network capabilities. We perform extensive experiments to evaluate the performance of the proposed algorithm. Experimental results indicate that the proposed algorithm has very small execution time and obtains near optimal solutions.

CCS CONCEPTS

•Networks → Cloud computing; •Computer systems organization → Cloud computing;

KEYWORDS

edge computing, services, component placement algorithm

ACM Reference format:
Tayebeh Bahreini and Daniel Grosu. 2017. Efficient Placement of Multi-Component Applications in Edge Computing Systems. In *Proceedings of The Second ACM/IEEE Symposium on Edge Computing, San Jose, CA USA, October 2017 (SEC'17),* 11 pages.
DOI: 10.475/123_4

1 INTRODUCTION

Mobile devices have become the primary computing platforms for many users, mainly because of their availability and convenience. Many of the popular mobile applications require heavy processing while mobile devices have limited computational resources and storage capacity. Furthermore, some mobile applications such as video streaming, speech recognition, and navigation consume a large amount of energy and reduce the battery life of mobile devices. These limitations of mobile devices can be addressed by running applications remotely on a static infrastructure that does not suffer from these limitations. Mobile Cloud Computing (MCC) has been introduced to allow mobile applications to perform their computation on cloud servers. MCC is a centralized paradigm for mobile applications in which data computation and storage are moved from mobile devices to resource-rich servers located in clouds. In fact, MCC is an extension of cloud computing which considers the mobility of users and the ad-hoc structure of mobile devices [8].

Rudenko et al. [13] and Flinn et al. [9] are among the first researchers who suggested running mobile applications on cloud servers in order to extend the battery life of mobile devices. Rudenko et al. [13] evaluated the effectiveness of offloading techniques by conducting several experiments. Their results indicate that remote execution of mobile applications can reduce the energy consumption significantly. However in MCC, the centralized data-centers that are used by cloud services are usually far from end-users, therefore, the communication between mobile devices and data-centers involves many network hops and results in high latencies. Thus, MCC is not efficient for some applications that need a quick response time or transmit large amounts of data [14].

In recent years, several paradigms such as Cloudlet [15], Fog Computing [4], Follow Me Cloud [16], and Mobile Edge Computing (MEC) [12] have been proposed to solve the inefficiencies of mobile cloud computing by sending a portion of data/computation to the edge of the network instead of sending it to the cloud data-centers.

Cloudlet, an architecture proposed by Satyanarayanan et al. [15] has the goal of bringing the cloud closer to the user. A Cloudlet, also known as mobile micro-cloud, is a cluster of multi-core computers with high internal connectivity that is available for nearby mobile devices. Mobile Edge Computing [12] was also introduced to enable processing data at the edge of the network, where an edge can be any computing resources of the network. In contrast to Cloudlets, edge nodes are widely deployed and available to all mobile users, not just to some specific ones. Since edge nodes can be co-located with base stations, they have access to additional information such as position and mobility of users, while Cloudlets are mostly stationary computers with fast and stable internet access, offering computing, bandwidth, and storage resources to nearby mobile users; and users access them via a local area network such as Wi-Fi [3].

One of the challenging issues in MEC is that the users change their locations dynamically and the current assignment of components to the edge servers might not be the best in terms of the costs involved. Thus, in order to minimize the costs of executing the application, some of the application's components might need to be offloaded to different edge or core cloud nodes. The problem of assigning components of an application to the edge/core nodes is called the *multi-component application placement problem*.

Application placement is an important problem in MEC and is associated with deciding on which resources the components of an application should be executed. In this problem, each application is represented as a graph in which nodes are components of the application and edges between nodes indicate the communication between them. Physical resources can also be represented by a graph in which nodes represent computing resources (servers) and edges between nodes represent communication links between them. Thus, the application placement problem can be viewed as the problem of mapping application graphs onto resource graphs.

In the placement problem, the components of a users' application can be run either on the core cloud, or on the edge of the network. In MEC, the resource availability, network conditions, and users' locations are changing dynamically. Hence, the components of the application may need to migrate from one edge/core server to another over time. Therefore, a good solution for placing applications should consider both the network conditions and the mobility of users.

1.1 Our Contributions

In this paper, we formulate the offline version of the multi-component application placement problem as a Mixed Integer Linear Program (MILP). The solution obtained by solving the offline version is used as the lower bound on the performance of the proposed online algorithm. Our formulation of the problem departs from the existing work since it does not impose any restrictions on the topology of the graphs characterizing both the applications and the physical resources. Considering this general setting, we design a heuristic algorithm that solves the online version of the multi-component application placement problem. Our goal is to obtain an algorithm based on simple algorithmic techniques such as matching and local search, and to avoid the use of complex approaches such as those based on Markov Decision Processes. The main idea in the design of our algorithm is to determine the best matching between components of the application and the edge/core servers without considering the communication requirements among the components, and then consider the communication requirements among the components and use a local search procedure to improve the solution. The proposed algorithm has low complexity and adds a negligible overhead to the execution of the applications. We perform extensive experiments to characterize the performance of the proposed algorithm using synthetically generated traces. We compare the performance of our algorithm against the optimal solution obtained by solving the offline version of the problem.

1.2 Organization

The rest of the paper is organized as follows. In Section 2, we review the related work on the placement problem in cloud and edge computing. In Section 3, we introduce the multi-component application placement problem and present its MILP formulation. In Section 4, we present the proposed heuristic algorithm. In Section 5, we present and analyze the experimental results. In Section 6, we conclude the paper and suggest possible directions for future research.

2 RELATED WORK

Recently, several approaches for solving variants of the application/service placement problem in data-centers have been proposed. These approaches are not directly applicable to MEC because they do not take into account the changes in execution costs due to the mobility of users when making placement decisions. Chowdhury et al. [6] proposed an LP-based approximation algorithm for minimizing the total cost with some considerations on load balancing. The approximation ratio of their algorithm is $O(N)$, where N is the number of nodes in the data-center. In their formulation, they assumed that servers in the data-center do not support resource sharing among different application nodes. Another LP-based approximation algorithm for solving the placement problem of tree applications with the goal of minimizing the total network congestion was developed by Bansal et al. [2]. Their algorithm has an approximation ratio of $\gamma = O(D^2 \log(ND))$ with $N^{O(D)}$ time-complexity, where D is the number of levels of nodes in the application graph. The authors also provided an online algorithm to minimize the maximum load balance on the physical nodes and links with competitive ratio of $O(\gamma \log(N))$. Dutta et al. [7] proposed an offline LP-based approximation algorithm for solving the placement problem with the objective to minimize the congestion on the physical links. They assumed that the resource graph has a tree structure. The algorithm places application nodes on the leaves of the tree. However, the settings and objectives of the placement problems considered by these authors are different from those we consider in this paper and the algorithms do not directly apply to the multi-component application placement problem in MEC.

The application placement problem in MEC has to consider several issues that were not present in the data-center settings. After the initial application placement, mobile users may move to a different location. In addition to this, the resource availability of servers may change over time. An efficient component placement algorithm must be adaptive to this dynamic setting and must change the location of components over time to minimize the cost, if necessary. Many of the dynamic application placement approaches formulated the problem as a sequential decision making problem in the framework of Markov Decision Processes (MDPs). Ksentini et al. [10] modeled the application/service migration problem considering the user's mobility in the Follow Me Cloud paradigm using MDPs [16]. In their formulation, they considered one dimensional mobility patterns. They implemented the value iteration algorithm in MATLAB to find the optimal application migration policy. Urgaonkar et al. [17] modeled the application placement problem as an MDP. To reduce the state space, they converted the problem into two independent MDP problems with separate state spaces and designed an online algorithm for the new problem that is provably cost-optimal. Wang et al. [18] presented a novel online algorithm for the application placement problem in the context of MEC. They modeled the problem as an MDP. Then, they reduced the state space by deriving a new MDP model in which states are defined only based on the distance between user and servers. The authors provided an online algorithm for the distance-based MDP and showed that the distance-based MDP is a good approximation to the original problem. An online approximation algorithm for the placement problem was also developed by Wang et al. [19] in which both the application and the resource graphs are assumed to have a tree topology. Their algorithm finds the best placement for nodes as well as links between nodes such that the maximum weighted cost on each physical node and link of the system is minimized.

Most of the dynamic application migration approaches described above differ from the approach we propose in this paper in several aspects. They either considered a restricted topology for the application and resource graphs or considered different objectives. The above dynamic approaches are based on complex techniques and algorithms while ours is based on standard algorithmic techniques such as matching and local-search. Because our approach employs low complexity algorithms it is very suitable for implementation in real MEC systems.

3 MULTI-COMPONENT APPLICATION PLACEMENT PROBLEM

In this section, we formulate the *multi-component application placement problem (MCAPP)*. We consider a time slotted system where the location of users may change from one time slot to another. The time slots are denoted by t, $t = 1, \dots, T$, where T is the maximum time required to complete the execution of the application in the system. The locations of users do not change during one time slot. The location of a user is specified by its coordinates in a two-dimensional grid of cells. A user can change its location between two time slots, that is, it can move into any of the neighboring cells or stay in the same cell.

We consider an edge system composed of a set $\mathcal{S} = \{S_1, S_2, \dots, S_m\}$ of m servers. Servers can be edge or core cloud servers having different computational powers, and therefore, different costs. The cost of running one unit of load on server S_i in time slot t is denoted by c_{it}. Servers can be located in any cell of the two-dimensional grid and their positions are fixed.

Each service/application C of a user consists of a set of components $C = \{C_1, C_2, \dots, C_n\}$. The processing requirement of component C_j in time slot t is denoted by p_{jt}. This represents the amount of component C_j's load that needs to be processed in time slot t. We do not impose any restrictions on the communication between the components, any component can communicate with any other component of the application (i.e., the graph modeling the application is not restricted). We also assume that a server can communicate with any other server, incurring different costs for different servers. Here the *objective* is to find a mapping between components and servers, such that the total placement cost is minimized. The total placement cost is composed of four types of costs:

(i) γ_{ijt}: *the cost of running component C_j on server S_i in time slot t.* This cost is defined as the product of the cost of processing a unit load at S_i and the amount of load that needs to be processed:

$$\gamma_{ijt} = c_{it} \cdot p_{jt} \tag{1}$$

(ii) $\rho_{ii'jt}$: *the cost of relocating component C_j from server S_i to server $S_{i'}$ in time slot t.* In MEC, the locations of users may change during the execution of their applications. Also, the workload of the edge servers and other conditions of the network may vary from time to time. Therefore, it may be required to change the location where the components are running. The relocation cost is defined as follows:

$$\rho_{ii'jt} = l_{ii't} \cdot q_{jt} \cdot r_t \tag{2}$$

where $l_{ii'}$ is the distance between servers S_i and $S_{i'}$ in time slot t, q_{jt} is the size of component C_j in time slot t that would migrate, and r_t is the cost of transferring one unit of data over one unit of distance in time slot t. Here the distance between servers is the Manhattan distance, that is, if server S_i is located in cell (x, y) and server $S_{i'}$ is located in cell (x', y') in time slot t, then the distance between the two servers is given by $l_{ii't} = |x - x'| + |y - y'|$.

(iii) δ_{ijt}: *the communication cost between component C_j (assigned to server S_i) and the user in time slot t.* In each time slot, data communication between components and the user may be required. This cost is defined as follows:

$$\delta_{ijt} = d_{it} \cdot h_{jt} \cdot r_t \tag{3}$$

where h_{jt} is the size of data that must be transferred between component C_j and the user in time slot t, and d_{it} is the distance between server S_i (that runs component C_j) and the user, in time slot t. The distance between the server and user is the Manhattan distance as defined in (ii) above.

(iv) $\tau_{ii'jj't}$: *the communication cost between components C_j and $C_{j'}$ that are located on servers S_i and $S_{i'}$, respectively, in time slot t.* Suppose that component C_j is located on server S_i and component $C_{j'}$ is located on server $S_{i'}$. The communication cost between components is defined as follows:

$$\tau_{ii'jj't} = l_{ii't} \cdot g_{jj't} \cdot r_t \tag{4}$$

where $g_{jj't}$ is the size of data that must be transferred between component C_j and component $C_{j'}$ in time slot t.

In Table 1, we present the notation that is used throughout the paper.

We consider the *offline* version of MCAPP in which all the parameters specified above are assumed to be known in advance. This assumption will be removed in the next section, where we consider the *online* version of the problem and provide a heuristic algorithm to solve it. The objective of MCAPP is to minimize the total cost of executing the application. Considering this objective and the offline setting, we formulate the MCAPP problem as a Mixed Integer Linear Program (MILP) as follows:

MCAPP-MILP:

$$\min \quad \sum_{i=1}^{m}\sum_{j=1}^{n}\sum_{t=0}^{T}(\gamma_{ijt}+\delta_{ijt})\cdot x_{ijt} + \sum_{i=1}^{m}\sum_{i'=1}^{m}\sum_{j=1}^{n}\sum_{t=1}^{T}\rho_{ii'jt}\cdot y_{ii'jt} + \sum_{i=1}^{m}\sum_{i'=1}^{m}\sum_{j=1}^{n}\sum_{j'=1}^{n}\sum_{t=0}^{T}\tau_{ii'jj't}\cdot z_{ii'jj't} \tag{5}$$

subject to:

$$\sum_{j=1}^{n} x_{ijt} \le 1 \quad \forall i,t \tag{6}$$

$$\sum_{i=1}^{m} x_{ijt} = 1 \quad \forall j,t \tag{7}$$

$$x_{ijt} + x_{i'j't} - 1 \le z_{ii'jj't} \quad \forall i,i',j,j',t \tag{8}$$

$$x_{i'j(t-1)} + x_{ijt} - 1 \le y_{ii'jt} \quad \forall i,i',j,\forall t>0 \tag{9}$$

$$x_{ijt} \in \{0,1\} \quad \forall i,j,t \tag{10}$$

$$y_{ii'jt} \in \{0,1\} \quad \forall i,i',j,t \tag{11}$$

$$z_{ii'jj't} \in \{0,1\} \quad \forall i,j,i',j',t \tag{12}$$

The decision variables x_{ijt} are defined as follows: $x_{ijt} = 1$, if component C_j is assigned to server S_i in time slot t; and 0 otherwise. We define a binary variable $y_{ii'jt}$ which is 1, if both $x_{i'j(t-1)}$ and x_{ijt} are 1; and 0 otherwise. This means that for component C_j in time slot t, if relocation from server $S_{i'}$ to server S_i happens, then $y_{ii'jt} = 1$. We also define another binary variable $z_{ii'jj't}$ which is 1, if both variables x_{ijt} and $x_{i'j't}$ are 1; and 0 otherwise. The objective function is the sum of the four types of costs that we defined above. Constraints (6) ensure that in each time slot each component is assigned to exactly one server. Constraints (7) guarantee that in each time slot, each server is used by at most one component. Constraints (8) and (9) define $z_{ii'jj't}$ and $y_{ii'jt}$, respectively. Constraints (10)-(12) represent the integrality requirements for the decision variables. The optimal solution obtained by solving the MCAPP-MILP will be used in the experimental results section as a lower bound for the solution obtained by the proposed online heuristic algorithm for solving MCAPP.

Table 1: Notation

m	Number of servers
n	Number of components
T	The time required to complete the execution of application C in the system
γ_{ijt}	Cost of running component C_j on server S_i in time slot t
c_{it}	Cost of processing one unit of load on server S_i
p_{jt}	Processing requirement of component C_j in time slot t
$\rho_{ii'jt}$	Cost of relocating component C_j from server S_i to server $S_{i'}$ in time slot t
$l_{ii't}$	Distance between servers S_i and $S_{i'}$ in time slot t
q_{jt}	Size of component C_j in time slot t
r_t	Cost of transferring one unit of data over a unit of distance in time slot t
δ_{ijt}	Communication cost between component C_j (assigned to server S_i) and the user in time slot t
d_{it}	Distance between server S_i and the user in time slot t
h_{jt}	Size of data that needs to be transferred between component C_j and the user in time slot t
$\tau_{ii'jj't}$	Communication cost between components C_j and $C_{j'}$ that are located on servers S_i and $S_{i'}$, respectively, in time slot t
$g_{jj't}$	Size of data that needs to be transferred between components C_j and $C_{j'}$ in time slot t

4 AN ONLINE ALGORITHM FOR THE MULTI-COMPONENT APPLICATION PLACEMENT PROBLEM

In this section, we consider the *online* version of MCAPP and design a heuristic algorithm to solve it. In the online version of MCAPP the values of the cost parameters introduced in the previous section may change every time slot and are not known *a priori*. We assume that at the end of every time slot the values of the parameters for the next slot are known and the proposed algorithm determines the allocation for the next time slot based on those new values.

The main design idea of the algorithm is to first ignore the communication costs between the components (i.e., $\tau_{ii'jj't} = 0$) and determine the minimum cost assignment of the components to servers using the Hungarian algorithm [11]. The Hungarian algorithm is a polynomial time algorithm that solves the assignment problem optimally. The algorithm has as input the weights Γ_{ij} of the edges of the bipartite graph in which one partition is composed of vertices corresponding to the servers, and the other composed of vertices corresponding to the application components. The algorithm finds a perfect matching that gives the minimum cost assignment of components to servers. Once the assignment is determined, the algorithm takes into account the communication costs between the components, $\tau_{ii'jj't}$ and performs a local search procedure that obtains the final solution to MCAPP.

As was mentioned above, when there is no inter-component communication, in each time slot the problem can be viewed as a matching problem. In the first time slot ($t = 0$), the problem

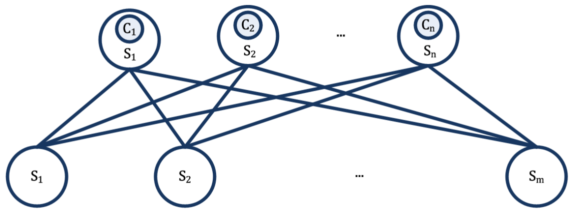


Figure 1: Matching components to servers (in time slots $t > 0$).

is to match components to servers, where each assignment of component C_j to server S_i has a specific cost given by:

$$\Gamma_{ij} = \delta_{ij0} + \gamma_{ij0} \tag{13}$$

The relocation cost is not considered in the first time slot, since $\rho_{ikjt} = 0$. In the next time slots ($t > 0$), the problem is to reassign components to servers taking users' location dynamics and other mentioned factors into account. In other words, the algorithm must decide whether a component stays on the current server or migrates to another one (see Figure 1). The cost of assigning component C_j to server S_i must include the relocation cost and thus, it is given by:

$$\Gamma_{ij} = \gamma_{ijt} + \delta_{ijt} + \rho_{ikjt} \tag{14}$$

where k is the location of component j in time slot $t-1$.

The proposed online algorithm, called MCAPP-IM (where IM stands for Iterative Matching), is given in Algorithm 1. The algorithm is executed in each time slot t, until the application completes its execution. The input to the algorithm in time slot t consists of the four cost parameters, γ_{ijt}, ρ_{ikjt}, δ_{ijt}, and $\tau_{ii'jj't}$, and a vector $\boldsymbol{y}$ specifying the current assignment of components to servers. Vector $\boldsymbol{y}$ indicates the location of each component (i.e., $y_j = i$, if component C_j is assigned to server S_i). To make it easy to describe the algorithms we use the following notation: $\boldsymbol{\gamma}_t$ is the array of costs of executing components on servers; $\boldsymbol{\delta}_t$ is the array of communication costs between user and components; $\boldsymbol{\rho}_t$ is the array of relocation costs; and $\boldsymbol{\tau}_t$ is the array of inter-component communication costs. The values of these parameters are determined during time slot $t-1$ and are used as input to the algorithm in slot t. The variable *cost* is the cost of running the application under the current assignment.

First, the algorithm computes the cost matrix $\boldsymbol{\Gamma}$ (lines 2-6) and then, in line 7, it determines the optimal assignment of the components to servers by calling the function HUNGARIAN($\boldsymbol{\Gamma}$). This function implements a variant of the Hungarian method algorithm and takes as input the cost $\boldsymbol{\Gamma}$ and returns the assignment as the vector $\boldsymbol{y}$. Since the Hungarian algorithm is well known we will not describe it here, but we refer the reader to Kuhn [11]. The entry Γ_{ij}, $i = 1, \ldots, m$; $j = 1, \ldots, n$, of the cost matrix, represents the cost of assigning component C_j to server S_i without considering the cost of communication between the components. Since this cost is not considered, the Hungarian algorithms is able to determine the optimal assignment of components to servers. This optimal assignment is not a solution for the MCAPP problem, it is an optimal assignment for the MCAPP with zero costs for the communication between

Algorithm 1 MCAPP-IM Algorithm

```
    {Executed every time slot $t$}
Input: $\boldsymbol{\gamma}_t$: costs of running components on servers
Input: $\boldsymbol{\delta}_t$: costs of comm. between user & components
Input: $\boldsymbol{\rho}_t$: relocation costs
Input: $\boldsymbol{\tau}_t$: inter-component communication costs
Input: $\boldsymbol{y}$: current assignment of components to servers
 1: $cost \leftarrow 0$
 2: for $i = 1, \ldots, m$ do
 3:     for $j = 1, \ldots, n$ do
 4:         $\Gamma_{ij} \leftarrow \gamma_{ijt} + \delta_{ijt} + \rho_{iy_jjt}$
 5:     end for
 6: end for
 7: $\boldsymbol{y} \leftarrow$ HUNGARIAN($\boldsymbol{\Gamma}$)
 8: for $j = 1, \ldots, n$ do
 9:     $cost \leftarrow cost + \Gamma_{y_jj}$
10: end for
11: $(\boldsymbol{y}, cost) \leftarrow$ L-SEARCH($\boldsymbol{y}, \boldsymbol{\tau}_t, \boldsymbol{\Gamma}, cost$)
Output: $(\boldsymbol{y}, cost)$
```

components (i.e., $\tau_{ii'jj't} = 0$). In lines 8-10, the algorithm computes the total cost of the new assignment determined by the Hungarian algorithm. Next, in line 11, the algorithm calls the local search algorithm, L-SEARCH, that takes into account the cost of communication between components and finds a solution for the MCAPP problem. For the first time slot ($t = 0$) the cost of relocation is zero (i.e., $\rho_{ikjt} = 0$) and thus, it does not contribute to the costs determined in lines 2-6.

The local search algorithm, L-SEARCH, is given in Algorithm 2. The input to the algorithm consists of: $\boldsymbol{y}$, the vector specifying the current assignment obtained by the Hungarian algorithm; $\boldsymbol{\tau}_t$, the array of inter-component communication costs; the cost matrix $\boldsymbol{\Gamma}$; and *cost*, the cost of the current assignment. First, the algorithm computes the total inter-component communication cost Λ_j of each component C_j for the current time slot (lines 4-9). Then, in line 10, it determines the index of the *bottleneck component*, denoted by b. The bottleneck component is the component that has the maximum total inter-component communication cost. After that, the algorithm executes a for loop (lines 11-26) in which it tries to find a lower total cost assignment by swapping the component that is currently placed on server S_i with the bottleneck component. This is done by obtaining the index of the component assigned to server S_i denoted by k and then calling the function SWAP-COMPONENTS(b, k). This function swaps the components C_b and C_k that is, assigns the bottleneck component to S_i and component C_k to the server in which C_b resided. If there is no component C_k on server S_i, then C_b is assigned to S_i and the server on which C_b resided is marked as available. The function outputs a new assignment vector $\boldsymbol{y}$. After this, in lines 15-20, the algorithm computes *new_cost*, the total cost of the system under the new assignment. If there is an improvement in the cost, the algorithm updates the total cost, *cost*, otherwise it restores the previous assignment by calling the SWAP-COMPONENTS function (lines 21-25). The algorithm continues this procedure as long as there is an improvement in the solution.

Algorithm 2 L-SEARCH Algorithm

```
Input: $y$: current assignment of components to servers
Input: $\tau_t$: inter-component communication costs
Input: $\Gamma$: cost matrix
Input: $cost$: total cost of current assignment
 1: $prev\_cost \leftarrow cost$
 2: $stop \leftarrow false$
 3: while not $stop$ do
 4:     for $j = 1, \ldots, n$ do
 5:         $\Lambda_j \leftarrow 0$
 6:         for $j' = 1, \ldots, n$ do
 7:             $\Lambda_j \leftarrow \Lambda_j + \tau_{y_j, y_{j'} jj' t}$
 8:         end for
 9:     end for
10:     $b \leftarrow \mathrm{argmax}_j \{\Lambda_j\}$
11:     for $i = 1, \ldots, m$ do
12:         $new\_cost \leftarrow 0$
13:         $k \leftarrow$ index of the component assigned to server $S_i$
14:         $y \leftarrow$ SWAP-COMPONENTS($b$, $k$)
15:         for $j = 1, \ldots, n$ do
16:             $new\_cost \leftarrow new\_cost + \Gamma_{y_j j}$
17:             for $j' = 1, \ldots, n$ do
18:                 $new\_cost \leftarrow new\_cost + \tau_{y_j y_{j'} jj' t}$
19:             end for
20:         end for
21:         if $new\_cost < cost$ then
22:             $cost \leftarrow new\_cost$
23:         else
24:             $y \leftarrow$ SWAP-COMPONENTS($k$, $b$)
25:         end if
26:     end for
27:     if $cost < prev\_cost$ then
28:         $prev\_cost \leftarrow cost$
29:     else
30:         $stop \leftarrow true$
31:     end if
32: end while
Output: $y$
Output: $cost$
```

We now investigate the time complexity of the proposed algorithm. The time complexity of the Hungarian algorithm is $O(\max(m^3, n^3))$. Because each component is not chosen as bottleneck more than once, the time complexity of L-SEARCH is $O(mn^3)$. Therefore, the time complexity of MCAPP-IM is $O(\max(m^3, n^3) + mn^3) = O(m^3 + mn^3)$.

5 EXPERIMENTAL RESULTS

We perform extensive experiments in order to investigate the properties of the proposed algorithm, MCAPP-IM. We compare its performance against that of the optimal solution for the offline version of MCAPP problem, and that of another online algorithm. In the following, we describe the experimental setup and analyze the experimental results.

Table 2: Simulation parameters

	Distribution	
Param.	communication-intensive	computation-intensive
c_{it}	$N(\mu_{it}, 0.2\mu_{it})$, $\mu_{it} \sim U[1, 10]$	$N(\mu_{it}, 0.2\mu_{it})$, $\mu_{it} \sim U[1, 10]$
p_{jt}	$N(\mu_{jt}, 0.2\mu_{jt})$, $\mu_{jt} \sim U[0, 10]$	$N(\mu_{jt}, 0.2\mu_{jt})$, $\mu_{jt} \sim U[1, 10^7]$
q_{jt}	$U[10, 40]$	$U[10, 40]$
h_{jt}	$U[1, 20]$	$U[1, 20]$
$g_{jj't}$	$U[1, 10^7]$	$U[1, 10]$
r_t	$U[0, 1]$	$U[0, 1]$

5.1 Experimental Setup

Because the development of MEC is still in the early stages, there are no MEC workload traces that are publicly available. Therefore, for our experiments, we have to rely on synthetically generated instances for the MCAPP problem. In the following, we describe how we generate the problem instances that drive our simulation experiments and describe the experimental setup.

We consider a time slotted system in which the locations of users in the network may change from one time slot to another, but do not change during one time slot. The users and servers are located within a two-dimensional grid of 150×150 cells. Initially, a user can be in any cell of the grid network and its location is drawn randomly from a uniform distribution over the locations of the grid. In our setting, we assume that the mobility of users is based on the random walk model [5] in a two-dimensional space, which is an approximation of real world mobility traces. In every new time slot, a user can stay in its place or move into any of neighboring cells with equal probability. The servers are located within the same two-dimensional grid network and the coordinates of their positions are drawn from a uniform distribution. The distance between servers and between servers and users, is the Manhattan distance (as defined in Section 3).

We generate several problem instances with different values for the number of components of the application (n), the number of servers in the network (m), and the total running time of the mobile application (T). The number of components for each application ranges from 4 to 180, while the number of servers ranges from 10 to 200. The reason for choosing these ranges is that in practice the number of components of an average application rarely exceeds 20 and most likely is on the lower part of the range considered here.

To generate the cost parameters defined in Section 3 we take into account the type of applications we consider. Since the determinant factors in the performance of any algorithm for solving MCAPP are the computation cost and the inter-component communication cost, we decided to generate the parameters for the instances we consider according to a metric called *communication to computation ratio* (*CCR*). This metric is defined as the ratio of the average communication cost per component and the average execution cost per component.

Table 2 shows the type of distributions used to generate the parameters characterizing the problem instances used in our simulation experiments. We consider different ranges for the distribution for two classes of applications: a typical communication-intensive application and a typical computation-intensive application. All the cost parameters for these two types of applications are the same, except for parameters p_{jt} and $g_{jj't}$. These parameters indicate the computation/communication intensiveness of the application. In Table 2, we denote by $U[x, y]$, the uniform distribution within interval $[x, y]$, and by $N(\mu, v)$, the normal distribution with mean μ and variance v.

We compare the performance of our algorithm MCAPP-IM with that of another algorithm called MATCH and with that of the optimal solution obtained by solving MCAPP-MILP. The MATCH algorithm implements a variant of the Hungarian algorithm [11] and does not take into account the communication among components when making the placement decisions. We compare with this algorithm in order to investigate the improvement in the quality of the solution due to considering the communication time among components in the local search phase of MCAPP-IM.

For each type of instance, we execute MCAPP-IM and MATCH algorithms for ten random instances. The performance of the algorithms is evaluated by computing the *actual competitive ratio* (*CR*) which is defined as the ratio between the value of the solution obtained by an online algorithm (V) and that of the optimal solution for the offline problem (V^*):

$$CR = \frac{V}{V^*} \tag{15}$$

The optimal solution is obtained by solving the MCAPP-MILP program with the CPLEX solver.

The MCAPP-IM and MATCH algorithms are implemented in C++ and the experiments are conducted on an Intel 1.6GHz Core i5 with 8 GB RAM system. For solving MCAPP-MILP we use the CPLEX 12 solver provided by IBM ILOG CPLEX optimization studio for academics initiative [1].

5.2 Analysis of Results

We first investigate the performance of the MCAPP-IM algorithm in terms of actual competitive ratio and execution time on a set of small size instances consisting of $m = 10$ servers and $n = 4$ components. We chose this type of instances in order to be able to solve them optimally using CPLEX and compare the performance of our algorithm with that of the optimal solution. In Table 3, we give the communication to computation ratios, the competitive ratios, and the execution times obtained by MCAPP-IM, MATCH, and CPLEX on those instances. We also considered six values for T, the number of time slots needed to complete the application under consideration. We group the instances according to their CCR, into seven classes. Starting from the top of the table we list the computation-intensive instances and ending at the bottom of the table with the communication-intensive ones. CPLEX was not able to solve some of the communication-intensive instances in feasible time, and thus, we were not able to determine the competitive ratios for MCAPP-IM and MATCH algorithms. In those cases we left the entries in the table empty.

Table 3: Competitive ratios and execution times (MCAPP with $m = 10$ servers, $n = 4$ components)

		Competitive Ratio		Execution time (msec)		
CCR	T	MCAPP-IM	MATCH	MCAPP-IM	MATCH	CPLEX
0.002	1	1	1.00001	0.064	0.021	70.8
0.005	2	1	1	0.085	0.037	115.2
0.002	4	1	1	0.132	0.063	185.7
0.005	8	1	1.00001	0.221	0.120	303.3
0.008	16	1	1	0.389	0.180	575.7
0.023	32	1	1	0.765	0.380	1368.7
0.019	1	1.00015	1.00067	0.095	0.090	75.0
0.041	2	1	1.00001	0.094	0.048	100.3
0.052	4	1.00011	1.000012	0.135	0.054	205.0
0.081	8	1.00005	1.00005	0.230	0.090	348.0
0.085	16	1.00012	1.000013	0.456	0.216	657.4
0.051	32	1.00002	1.00003	0.734	0.379	1369.0
0.13	1	1.01	1.0113	0.036	0.015	65.2
0.16	2	1	1.003	0.103	0.040	181.0
0.12	4	1.001	1.003	0.137	0.050	1365.7
0.14	8	1.001	1.0004	0.352	0.012	1809.6
0.13	16	1.001	1.002	0.307	0.012	5281.3
0.14	32	1.001	1.005	0.140	0.080	5669.3
1.10	1	1.10	1.199	0.057	0.015	280.7
0.80	2	1.08	1.124	0.183	0.110	2182.3
1.02	4	1.07	1.081	0.153	0.061	6763.8
1.21	8	1.15	1.254	0.258	0.125	10295.4
0.70	16	1.03	1.061	0.469	0.183	566251.0
1.41	32	1.17	1.217	0.971	0.430	5745128.0
11.29	1	1.38	1.570	0.053	0.018	417.3
13.94	2	1.73	1.920	0.106	0.033	8840.4
17.80	4	1.72	1.926	0.561	0.155	9101456.0
12.80	8	-	-	0.323	0.119	$>$ 3 hours
12.11	16	-	-	0.588	0.210	$>$ 3 hours
11.81	32	-	-	1.310	0.442	$>$ 3 hours
128.2	1	1.47	2.381	0.087	0.052	549.7
172.8	2	1.41	2.002	0.105	0.054	6216.0
90.1	4	1.65	2.145	0.363	0.070	292797.0
141.6	8	-	-	0.451	0.180	$>$ 3 hours
116.6	16	-	-	0.680	0.230	$>$ 3 hours
138.1	32	-	-	1.132	0.416	$>$ 3 hours
1241	1	1.9	4.712	0.150	0.050	974.9
1028	2	1.7	2.431	0.223	0.070	25814.2
950	4	1.81	2.968	0.347	0.085	76330.1
981	8	-	-	0.466	0.146	$>$ 3 hours
931	16	-	-	0.794	0.235	$>$ 3 hours
1615	32	-	-	1.253	0.445	$>$ 3 hours

We select instances of three types from this table, one computation-intensive instance, one computation-communication balanced instance, and one communication-intensive instance, and perform a detailed analysis of the results. In Figure 2, we plot the total execution times (i.e., the sum of the execution times in all time slots) obtained by MCAPP-IM, MATCH, and CPLEX on those instances for different values of T using a logarithmic scale. The execution time of CPLEX is several orders of magnitude higher than the execution times of both MCAPP-IM and MATCH algorithms for all three types of instances. For the last two types of instances and larger values of T, CPLEX was not able to find the optimal solution in feasible time, and therefore there are no bars in the plots corresponding to these cases. The execution times of our proposed algorithm, MCAPP-IM, are under 1 millisecond in all cases making it very suitable for deployment in real MEC systems. We observe a slight increase of the execution time of MCAPP-IM with the increase in the number of time slots, but we believe that this increase is reasonable and that it will not make our algorithm a significant contributor to the overhead

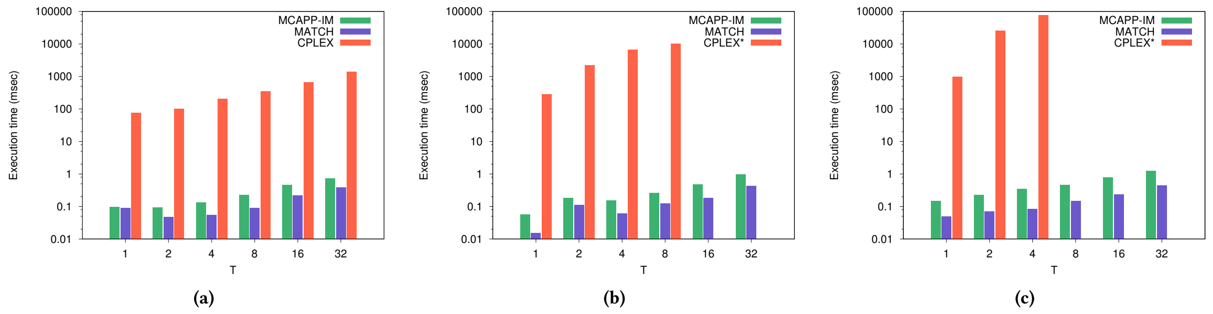


Figure 2: Execution time vs. number of time slots (a) Computation-intensive case; (b) Balanced communication-computation case; (c) Communication-intensive case. (*CPLEX was not able to determine the solution for $T = 16$ and 32 in (b), and for $T = 8$, 16, and 32 in (c) in feasible time, and thus, there are no bars in the plots for those cases)

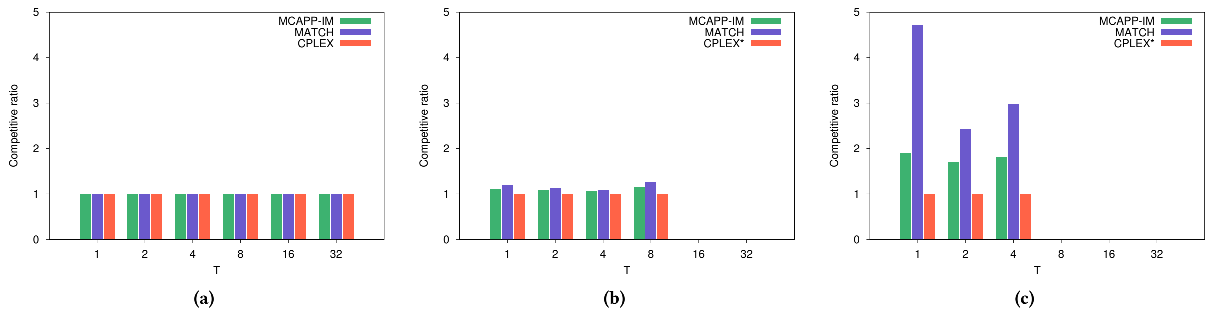


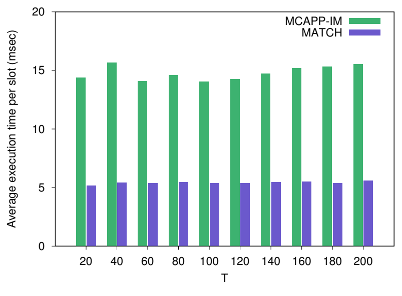
Figure 3: Competitive ratio vs. number of time slots (a) Computation-intensive case; (b) Balanced communication-computation case; (c) Communication-intensive case. (*CPLEX was not able to determine the solution for $T = 16$ and 32 in (b), and for $T = 8$, 16 and 32 in (c) in feasible time, and thus, there are no bars in the plots for those cases)

of placing the application components on edge servers. The MATCH algorithm obtains a slightly lower execution time than MCAPP-IM but as we will show next, this small execution time is obtained at the expense of not being able to provide near optimal solutions to the problem. The reason behind this, is that MATCH does not take into account the communication between components when determining the placement, requiring less time than MCAPP-IM.

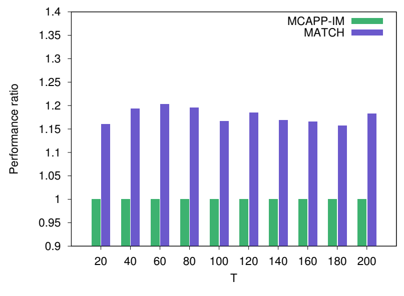
In Figure 3, we plot the actual competitive ratios obtained by MCAPP-IM, MATCH, and CPLEX. Since CPLEX obtains the optimal solution, we plot its competitive ratio as 1. In the case of computation-intensive instances (Figure 3a) the actual competitive ratios obtained by MCAPP-IM and MATCH are very close to 1, thus both algorithms obtain optimal solutions or solutions that are very close to the optimal. If there is almost no communication among the components, MCAPP-IM behaves similarly to MATCH, that is the local search step is not actually able to improve the solution beyond that obtained by matching. In the case of communication-intensive instances (Figure 3c) MCAPP-IM obtains much better competitive ratios than MATCH for all instances. That means that MCAPP-IM is able to obtain solutions that are closer to the optimal solution than those obtained by MATCH. As an example for $T = 4$ the competitive ratio of MCAPP-IM is 1.81 while that of MATCH is 2.968. Another important observation is that the actual competitive ratios obtained by MCAPP-IM are less than 2 for all the communication-intensive instances considered here, that is, they are independent on the number of slots we considered.

We next analyze the effect of the number of time slots, number of servers, and number of components on the performance of communication-intensive large-scale instances of the MCAPP problem. Since CPLEX is not feasible to use for solving such large instances, we will not compare the performance of our algorithm against the performance of the optimal solution obtained by CPLEX. Instead we will compare the performance of our algorithm against that of MATCH. In order to do this we use a new metric called the *performance ratio* defined as the ratio of the total cost obtained by MATCH and the total cost obtained by MCAPP-IM.

First, we investigate the effect of the number of time slots required to complete the application on the performance of MCAPP-IM. We consider large instances with $m = 100$ servers and $n = 50$ components and several values for the number of time slots ranging from 20 to 200. In Figure 4a, we plot the average execution time per time slot obtained by MCAPP-IM and MATCH. This allows us to determine the amount of time spent by the local search phase of MCAPP-IM in each time slot (since MATCH does not perform local search to improve the solution). We can observe that the execution time for the local

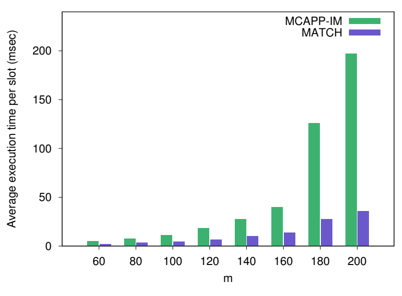


(a)

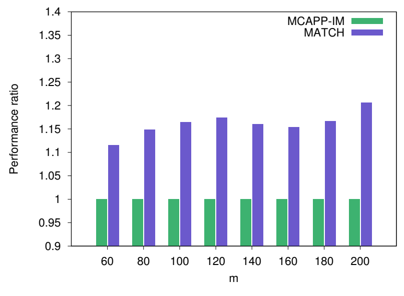


(b)

Figure 4: The effect of the number of time slots. (communication-intensive case, $m = 100$ servers, $n = 50$ components): (a) Average execution time per time slot; (b) Performance ratio.



(a)



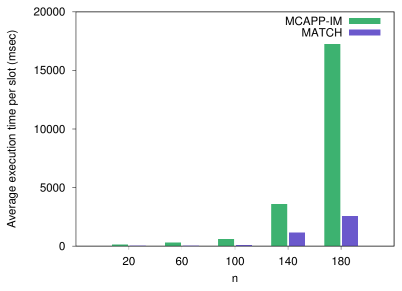
(b)

Figure 5: The effect of the number of servers (communication-intensive case, $n = 40$ components, $T = 20$ time slots): (a) Average execution time per time slot; (b) Performance ratio.

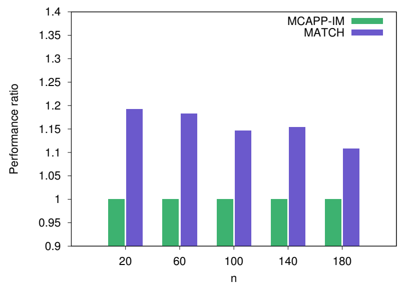
search phase is about 60 percent of the total execution time of MCAPP-IM per time slot. MATCH has a lower execution time per time slot than MCAPP-IM regardless of the total number of time slots. Another observation is that the execution times per time slot of both MCAPP-IM and MATCH are insensitive to the number of time slots. Given that in practice, the time slots are in the order of minutes, the execution times of MCAPP-IM per time slot are within reasonable ranges (13-16 milliseconds) and they do not contribute significantly to the total overhead of component placement in MEC. In Figure 4b, we plot the performance ratio of the two algorithms, MCAPP-IM and MATCH. In order to facilitate the comparison, we plot the performance ratio of MCAPP-IM which is 1, since the ratio is defined relative to the total cost obtained by MCAPP-IM. MCAPP-IM outperforms MATCH by obtaining solutions with smaller total cost. The reduction of the total cost obtained by MCAPP-IM is significant, ranging from 15 to 20 percent. This reduction in cost is consistent across the full range of values for the number of slots considered here.

Next, we investigate the effect of the number of servers on the performance of MCAPP-IM. We consider large instances with a fixed number of components ($n = 40$) and fixed number of slots ($T = 20$), and several values for the number of servers, ranging from 60 to 200. In Figure 5a, we plot the average execution time per time slot obtained by MCAPP-IM and MATCH. As expected, the average execution time per time slot of MCAPP-IM is sensitive to the number of servers in the system. This is because of the cubic growth in terms of m of the running time of the algorithm. For example for $m = 60$, the average execution time is around 5 milliseconds and for $m = 200$, the average execution time is around 197 milliseconds. Similar behavior is observed in the case of MATCH. For example, for $m = 60$, the average execution time is about 2 milliseconds and for $m = 200$, it is about 36 milliseconds. Therefore, both algorithms exhibit similar trends with respect to the execution time per time slot when the number of servers increases. In Figure 5b, we plot the performance ratio of the two algorithms MCAPP-IM and MATCH. MCAPP-IM obtains better solutions with total cost 12 to 20 percent lower than MATCH for all the number of servers. Also, the performance ratio of MATCH increases as the number of servers increases, which means that MCAPP-IM algorithm obtains better solutions for instances with a large number of servers.

Finally, we investigate the effect of the number of components on the performance of MCAPP-IM. We consider large instances with a fixed number of servers ($m = 200$) and a fixed number of time slots ($T = 20$) and several values for the number of components, ranging from 20 to 180. In Figure 6a, we plot the average execution time per time slot obtained by MCAPP-IM and MATCH. The average execution time of both



(a)



(b)

Figure 6: The effect of the number of components (communication-intensive case, $m = 200$ servers, $T = 20$ time slots): (a) Average execution time per time slot; (b) Performance ratio.

algorithms increases with the number of components. For example for $n = 20$, the average average execution time of MCAPP-IM is about 145 milliseconds and for $n = 180$, it is about 17 seconds. These instances with large number of components are not expected to be encountered in practice, but we still consider them here to investigate the scalability of the algorithms. The running time of MATCH is much lower but the algorithm obtains worst solutions, that is, solutions with higher total cost. In Figure 6b, we plot the performance ratio of the two algorithms, MCAPP-IM and MATCH. Again, MCAPP-IM outperforms MATCH for all the number of components, but the performance ratio decreases with the increase in the number of components. In fact, when the number of components is much less than the number of servers, the local search procedure employed by MCAPP-IM explores more alternative placements to find the one that improves the solution.

The experimental results show that MCAPP-IM obtains solutions that are very close to the optimal and requires very low execution time per slot for reasonably large instances. For the average size instances, the ones we expect to encounter in practice, the proposed algorithm performs very well with respect to both the quality of the solutions and the execution time per time slot.

6 CONCLUSION

In this paper, we addressed the problem of placement of multi-component applications in MEC. First, we formulated the offline version of the problem as a Mixed Integer Linear Program (MILP) and then developed a heuristic algorithm for solving the online version of the problem. The algorithm is based on an iterative matching process followed by a locals search phase in which the solution quality is improved. We performed extensive experiments to investigate the performance of the proposed algorithm. The results of these experiments indicated that the proposed algorithm obtains very good performance and requires very low execution time.

For future work, we plan to extend our algorithm to handle more general cases of the placement problem in which several applications of a user need to offload their components to edge servers. Another avenue for future research is to develop placement algorithms that take into account both the users' and providers' incentives when making placement decisions.

REFERENCES

[1] 2009. IBM ILOG CPLEX V12.1 User's Manual. (2009). ftp://ftp.software.ibm.com/software/websphere/ilog/docs/

[2] Nikhil Bansal, Kang-Won Lee, Viswanath Nagarajan, and Murtaza Zafer. 2011. Minimum congestion mapping in a cloud. In *Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*. ACM, 267–276.

[3] Michael Till Beck, Martin Werner, Sebastian Feld, and S Schimper. 2014. Mobile edge computing: A taxonomy. In *Proceedings of the Sixth International Conference on Advances in Future Internet*. 48 – 54.

[4] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. 2012. Fog computing and its role in the internet of things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*. ACM, 13–16.

[5] Tracy Camp, Jeff Boleng, and Vanessa Davies. 2002. A survey of mobility models for ad hoc network research. *Wireless communications and mobile computing* 2, 5 (2002), 483–502.

[6] Mosharaf Chowdhury, Muntasir Raihan Rahman, and Raouf Boutaba. 2012. Vineyard: Virtual network embedding algorithms with coordinated node and link mapping. *IEEE/ACM Transactions on Networking (TON)* 20, 1 (2012), 206–219.

[7] Debojyoti Dutta, Michael Kapralov, Ian Post, and Rajendra Shinde. 2012. Embedding paths into trees: VM placement to minimize congestion. In *European Symposium on Algorithms*. Springer, 431–442.

[8] Xiaopeng Fan, Jiannong Cao, and Haixia Mao. 2011. A survey of mobile cloud computing. *zTE Communications* 9, 1 (2011), 4–8.

[9] Jason Flinn and M. Satyanarayanan. 1999. Energy-aware Adaptation for Mobile Applications. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles (SOSP '99)*. 48–63.

[10] Adlen Ksentini, Tarik Taleb, and Min Chen. 2014. A Markov decision process-based service migration procedure for follow me cloud. In *2014 IEEE International Conference on Communications (ICC)*. IEEE, 1350–1354.

[11] H. W. Kuhn. 1955. The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly* 2, 1-2 (1955), 83–97.

[12] Milan Patel, B Naughton, C Chan, N Sprecher, S Abeta, A Neal, and others. 2014. Mobile-edge computing introductory technical white paper. *White Paper, Mobile-edge Computing (MEC) industry initiative* (2014).

[13] Alexey Rudenko, Peter Reiher, Gerald J Popek, and Geoffrey H Kuenning. 1998. Saving portable computer battery power through remote process execution. *ACM SIGMOBILE Mobile Computing and Communications Review* 2, 1 (1998), 19–26.

[14] Mahadev Satyanarayanan. 2015. A brief history of cloud offload: A personal journey from odyssey through cyber foraging to cloudlets. *GetMobile: Mobile Computing and Communications* 18, 4 (2015), 19–23.

[15] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. 2009. The case for vm-based cloudlets in mobile computing. *IEEE Pervasive Computing* 8, 4 (2009), 14–23.

[16] Tarik Taleb and Adlen Ksentini. 2013. Follow me cloud: interworking federated clouds and distributed mobile networks. *IEEE Network* 27, 5 (2013), 12–19.

[17] Rahul Urgaonkar, Shiqiang Wang, Ting He, Murtaza Zafer, Kevin Chan, and Kin K. Leung. 2015. Dynamic service migration and workload scheduling in edge-clouds. *Performance Evaluation* 91 (2015), 205 – 228.

[18] Shiqiang Wang, Rahul Urgaonkar, Murtaza Zafer, Ting He, Kevin Chan, and Kin K Leung. 2015. Dynamic service migration in mobile edge-clouds. In *IFIP Networking Conference (IFIP Networking), 2015*. IEEE, 1–9.

[19] Shiqiang Wang, Murtaza Zafer, and Kin K Leung. 2017. Online Placement of Multi-Component Applications in Edge Computing Environments. *IEEE Access* 5 (2017), 2514–2533.