Hindawi
Wireless Communications and Mobile Computing
Volume 2019, Article ID 1975312, 18 pages
https://doi.org/10.1155/2019/1975312

Research Article

Enhancing Offloading Systems with Smart Decisions, Adaptive Monitoring, and Mobility Support

Paulo A. L. Rego,[1] **Fernando A. M. Trinta**,[1] **Masum Z. Hasan**,[2] **and Jose N. de Souza**[1]

[1]*Group of Computer Networks, Software Engineering and Systems (GREat), Federal University of Ceará (UFC), Fortaleza, CE, Brazil*
[2]*Verizon, San Jose, CA, USA*

Correspondence should be addressed to Paulo A. L. Rego; pauloalr@ufc.br

Received 17 December 2018; Revised 16 March 2019; Accepted 24 March 2019; Published 21 April 2019

Academic Editor: Mohammad Shojafar

Mobile cloud computing is an approach for mobile devices with processing and storage limitations to take advantage of remote resources that assist in performing computationally intensive or data-intensive tasks. The migration of tasks or data is commonly referred to as offloading, and its proper use can bring benefits such as performance improvement or reduced power consumption on mobile devices. In this paper, we face three challenges for any offloading solution: the decision of when and where to perform offloading, the decision of which metrics must be monitored by the offloading system, and the support for user's mobility in a hybrid environment composed of cloudlets and public cloud instances. We introduce novel approaches based on machine learning and software-defined networking techniques for handling these challenges. In addition, we present details of our offloading system and the experiments conducted to assess the proposed approaches.

1. Introduction

The use of mobile devices in the daily activities of a large part of the world population is a common reality. Mobile applications (or apps) allow users to access data and services quickly and easily, virtually anywhere. Over the last decades, mobile apps have evolved significantly, with 3D graphics support, growing integration with different kinds of sensors, and remote services. In consequence, some devices with limited processing and storage capabilities face the challenge of running applications with some aforementioned features.

One strategy to address this issue is the integration between mobile devices and cloud resources, through a paradigm called mobile cloud computing (MCC) [1]. According to [2], MCC aims to provide a range of services, equivalent to the cloud, adapted to the capacity of resource-constrained devices, besides performing improvements of telecommunications infrastructure to improve the service provisioning. Along last years, different research groups have been studying this theme, including white-papers, software architecture, case studies, and research challenges for MCC. Most of them focus on the migration of data and tasks from mobile devices to remote servers with better processing and storage capacity. This migration is known in the literature as offloading.

In general, when proposing an offloading solution, the most important issue to be addressed is how to decide when it is worthwhile to migrate data or tasks from a mobile device to a remote execution environment (REE), which might be another mobile device or a cloudlet/cloud. This decision may take into account the time needed to transfer and execute the task remotely and the energy cost of the offloading operation, as well as several software, hardware, and network-related metrics. Diverse solutions have been proposed to decide when to offload [3–10], but most of them depend on constant profiling of metrics, which imposes the use of local resources that should be avoided. Besides that, the decision-making processes may also use computationally costly tasks, which should preferably be executed outside the mobile device.

Another important aspect in offloading systems is the support for users' mobility. Mobility is one of the most important characteristics of a pervasive computing environment and is defined as the ability of mobile nodes to seamlessly continue their work regardless of their displacements [11]. Many obstacles hinder the goals of seamless connectivity and consistency in mobile applications services [12]. As a result,

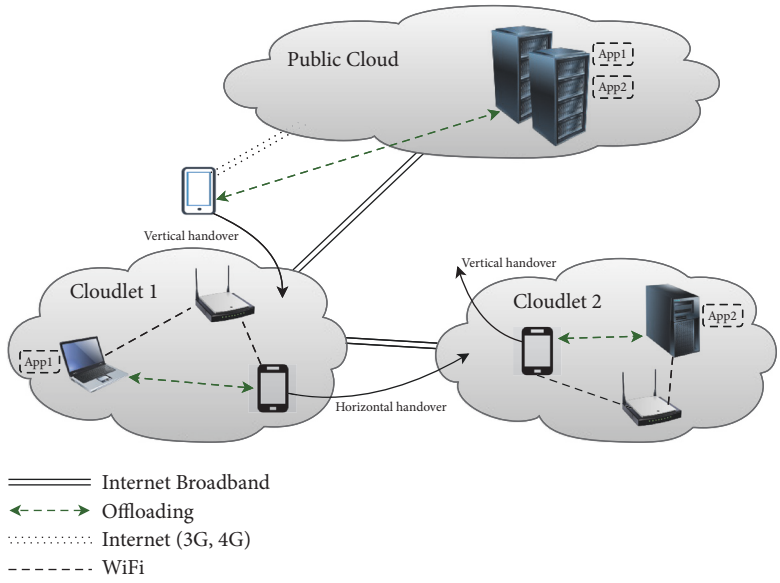


Figure 1: Supported scenario: hybrid environment composed of cloudlets and public cloud instances.

providing seamless connectivity and uninterrupted access to those services is a serious research issue for MCC, mainly because the execution of a mobile application may be highly context-dependent, which means that several metrics, such as availability of network and local/remote resources, impact at runtime [13].

In previous works [14, 15], we introduced a method-based multiplatform offloading system developed in our research group and a decision tree-based approach for assisting the offloading decision on MCC systems. Herein, we face some of the challenges of MCC and related computing paradigms (e.g., Fog Computing and Fog of Everything [16]) such as handling adaptive energy-efficient reconfiguration, reducing computing-plus-communication delay and service latency, and supporting devices mobility. To do that, we extend our offloading framework in three ways: (i) to adopt a lightweight offloading decision module, where the high intensive operations related to decision-making are executed outside the mobile device, (ii) to leverage an adaptive monitoring system to reduce the burden of metrics monitoring, and (iii) to provide seamless connectivity to mobile applications in a hybrid environment composed of public cloud and cloudlets.

Our enhanced framework leverages machine learning and software-defined networking techniques to handle offloading decision and to provide mobility support. Several experiments were conducted to evaluate the three aspects of the proposed solution.

The rest of the paper is structured as follows. First, Section 2 describes the proposed solution and Section 3 presents details and results of the experiments performed to evaluate the work. Section 4 discusses related work. Finally, Section 5 presents conclusion and future work.

2. Enhanced Offloading System

Existing offloading frameworks have motivated us to develop an offloading solution [17] in which the decisions of when and where to offload are not performed exclusively on mobile devices. Also, the lack of mobility support when a hybrid scenario is considered and the overhead caused by periodic context information gathering have also motivated this work. In the face of such problems, herein we introduce novel solutions to handle offloading decisions, perform adaptive monitoring, and support users' mobility.

We present Figure 1 to clarify the scenario supported by this work. As illustrated, our solution considers mobile devices and two types of remote servers (cloudlets and public cloud instances), where one or more mobile devices can use the same remote server. We assume that mobile devices and cloudlets are connected to the same Wi-Fi access point (i.e., one-hop away) or to a group of access points (when WLAN controllers are used), while mobile devices can connect to public cloud instances via Internet using either cellular network (e.g., 3G or 4G) or Wi-Fi hotspots.

Regarding users' mobility, the solution supports (*i*) horizontal handover when a user moves from one cloudlet to another and (*ii*) vertical handover, when the user moves from a cloudlet to public cloud or from public cloud to a cloudlet.

The proposed offloading system was built upon three solutions: the MpOS framework [3, 14], the OpenStack platform, and seamless cloud (SLC) API [18]. MpOS is an offloading framework for Android and Windows Phone, where mobile applications are composed of a set of methods, which can be marked by developers using the `@Remotable` Java annotation. As presented in Listing 1, the methods `applyEffect1` and `applyEffect2` are marked, in lines (2) and (4), as candidates to be offloaded.

OpenStack is an open source cloud computing platform that we use to run remote servers (cloudlets and public cloud instances) and SLC is an abstraction to facilitate secure and seamless interconnection of distributed clouds. One can use the SLC API to perform CRUD operations (i.e., create, read, update, and delete) on cloud resources programmatically, which means resources under control of multiple cloud

```
(1) public interface Effect {
(2)    @Remotable
(3)    public byte[ ] applyEffect1(byte source[ ]);
(4)    @Remotable
(5)    public byte[ ] applyEffect2(byte source[ ]);
(6)  }
(7) public class ImageEffect implements Filter {
(8)    public byte[ ] applyEffect1(byte source[ ]) { … }
(9)    public byte[ ] applyEffect2(byte source[ ]) { … }
(10) }
```

Listing 1: Example of the @Remotable markup.

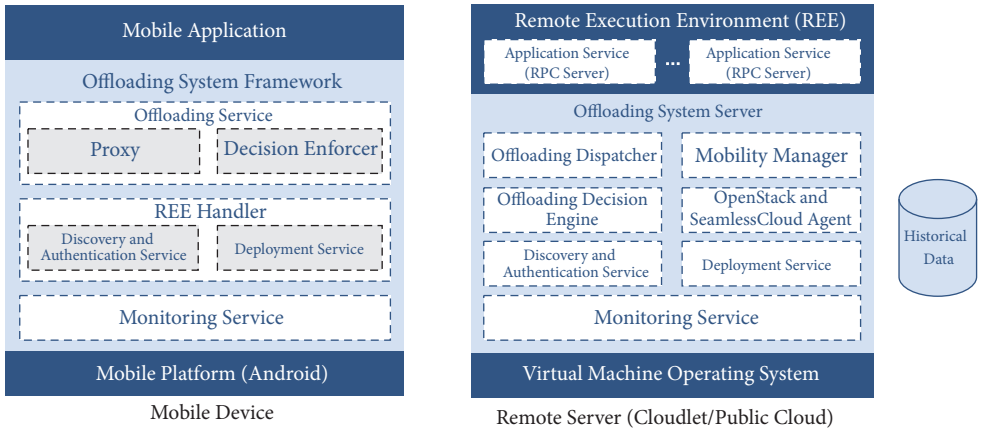


Figure 2: Overview of the new architecture components.

controllers can be seen as one seamless cloud instance, although resources are widely distributed on multiple clouds and geographical locations.

As in the MpOS framework, mobile devices are clients and access services executed on remote servers. Figure 2 presents an overview of the proposed offloading system architecture. In the following subsections we present the most important components and details of the offloading decision approach, the adaptive monitoring service, and the scheme for providing mobility support.

2.1. Handling Offloading. The *Offloading Service* is responsible for intercepting all methods that are marked with the @Remotable Java Annotations. The service uses the *Decision Enforcer* to make the offloading decision based on a two-class decision tree that is created on the remote server. The *Decision Enforcer* uses the monitoring information provided by the *Monitoring Service* to parse the decision tree and decide at runtime where the methods must be executed (e.g., locally on mobile device or on remote server). When a method is offloaded, the *Offloading Dispatcher* acting as a reverse proxy receives the offloading request and distributes it to a proper Application Services, even when such service is running in a different REE (in a different geographical location).

Then, the *Application Service* component receives the method's arguments, executes the method, and sends the result back to the mobile device, which can continue the application execution flow. The *Historical Data* is a database that stores details of all executed methods, such as application and method names, the size of arguments and result, and upload/download rate during the offloading, beyond other metrics.

The role of the *Offloading Decision Engine* is to use the history offloading and profiling information (monitoring data) to create an offloading decision tree for each application and user. The Offloading Decision Engine is also responsible for asynchronously sending the decision tree to the mobile device when a different tree is created (e.g., when the offloading decision is affected by changes in network condition or remote server workload).

The *Monitoring Service* collects information regarding several software, hardware, and network metrics, which are listed in Table 1. Such metrics are important to decide whether a method must be offloaded or not. However, continuous monitoring can profoundly impact the consumption of mobile devices resources. To handle this problem, we introduce an adaptive monitoring service, in which only the metrics actually "relevant" to the offloading decision are continuously monitored.

2.1.1. System Model. The computation offloading technique does not always increase performance [3–5]. *Kumar* et al.

Table 1: Metrics monitored by the monitoring service.

	Metrics
Software	heap size, local execution time, remote execution time, queue time
Hardware	available memory, processor type and number of cores
Network	connection type, RTT, jitter, upload and download rate, wireless RSS

Table 2: Model notation.

Symbol	Description
$E_{local,m}$	Time to execute the method m on mobile device (seconds)
SE_m	Time to serialize/encrypt input data of method m (seconds)
DD_m	Time to deserialize/decrypt output data of method m (seconds)
U_m	Time to transfer the method m (upload) (seconds)
D_m	Time to transfer the method m (download) (seconds)
$E_{rs,m}$	Time to execute the method m on remote server rs (seconds)∗
Q_{rs}	Queue waiting time on remote server rs (seconds)∗
I_m	Size of the input data of the method m (bytes)
O_m	Size of the output data of the method m (bytes)
T_{up}	Upload rate of the network in use (bytes/second)
T_{do}	Download rate of the network in use (bytes/seconds)

∗Note: rs might be *cloudlet* or *cloud*.

[19] present an analytical model to answer the question of when offloading improves the performance of mobile devices. The model decides when to offload by comparing the time to execute the computation on the mobile device with the time required to execute the computation on the remote server plus the time to transfer the data to/from such server.

In [15], we have extended *Kumar* et al. by considering (*i*) methods as the offloading units and (*ii*) other aspects related to the execution of the method outside the mobile device, such as encryption/decryption time and queue waiting time on remote server. Herein, we discuss the model in more detail (Table 2 presents the notation for easy reference).

A denotes the mobile application and $M_A = \{m_k, (1 \leq k)\}$ the set of methods of the application A that can be offloaded. Let $E_{local,mk}$ be the time to execute the method m_k on the mobile device and $E_{rs,mk}$ the time to execute the method m_k on the remote server.

If a method is offloaded to a remote server, we have to consider the time to upload the input data I_{mk} of method m_k to the server as

$$U_{mk} = \frac{I_{mk}}{T_{up}}, \tag{1}$$

and the time to download the output data O_{mk} of method m_k from the server as

$$D_{mk} = \frac{O_{mk}}{T_{do}}. \tag{2}$$

Besides that, SE_{mk} denotes the time to serialize/encrypt the input data of the method m_k, and DD_{mk} denotes the time to deserialize/decrypt the output data of the method m_k, while Q_{rs} denotes the queueing time on the remote server (i.e., delay before beginning the computation), which is variable and depends on the remote server workload. Hence, by extending the inequality of *Kumar* et al., offloading improves performance when (3) is satisfied (as illustrated in Figure 3(a)).

$$E_{local,mk} > SE_{\mathrm{mk}} + U_{mk} + Q_{rs} + E_{rs,mk} + D_{mk} + DD_{mk} \tag{3}$$

To decide where to offload, we consider two scenarios:

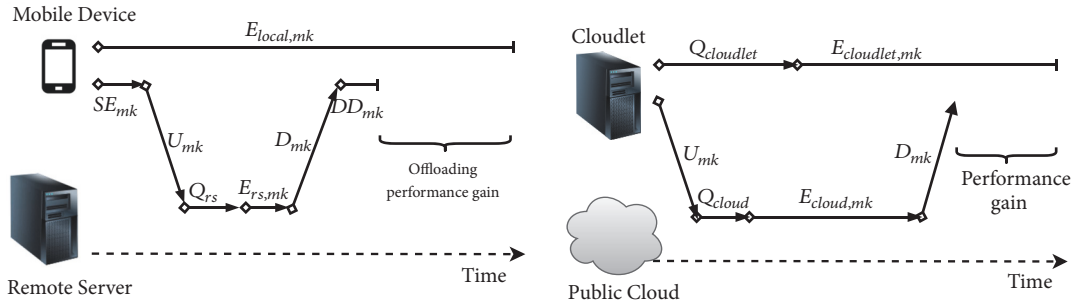
(i) The mobile device using 3G/4G: since there is no cloudlet, the only option is to perform offloading to the public cloud

(ii) The mobile device using Wi-Fi: when cloudlet and public cloud are available, the mobile device sends its offloading requests to the cloudlet, which has to decide where to execute the method, locally on the cloudlet or the public cloud (the same idea of (3))

$$E_{cloudlet,mk} + Q_{cloudlet} > U_{mk} + Q_{cloud} + E_{cloud,mk} + D_{mk} \tag{4}$$

If (4) is satisfied, the method must be executed on the public cloud (as illustrated in Figure 3(b)). Otherwise, it must be executed on the cloudlet. Unlike (3), we consider the waiting time on both remote execution environments, and we disregard encryption and decryption times because the connection between REEs must be secure. Therefore, depending on the queue waiting time Q_{rs} of each REE and the network condition between cloudlet and public cloud, the system may offload the methods to different REE.

Mathematical models similar to ours have been used for simulation in the literature [20], where a method can be represented as a number of instructions to be executed, and the computational power of mobile devices and remote servers



(a) Equation (3): When to offload?

(b) Equation (4): Where to offload?

Figure 3: Parameters that comprise the decisions of when and where to offload.

(1) $Average_I \longleftarrow$ Average I_m for all method $m \in M_A$
(2) $Average_O \longleftarrow$ Average O_m for all method $m \in M_A$
(3) **if** $Q_{cloudlet} > Average_I/T_{up} + Q_{cloud} + Average_O/T_{do}$ **then**
(4) **return** "public cloud"
(5) **else**
(6) **return** "cloudlet"
(7) **end if**

Algorithm 1: Procedure for selecting the main REE for the application A of user U.

can be represented based on MIPS (millions of instructions per second). In contrast to simulated environments, calculating $E_{local,m}$ and $E_{rs,m}$ in real systems is challenging. That is why our solution relies on online profiling and historical data to estimate such values.

Among other metrics, the *Monitoring Service* stores in the *Historical Data* database the context of the mobile device (e.g., T_{up}, T_{do}, $E_{local,m}$, $E_{rs,m}$, and Q_{rs}) when a method is executed and information about the method itself (e.g., I_m and O_m).

In [15], we presented all details regarding the experiment performed to validate the model and show how well our model represents the offloading process previously described.

2.1.2. Decision Tree Creation Process. The first step to creating a decision tree is to define the main REE (cloudlet or public cloud). Such remote server selection is a step prior to deciding when to perform computation offloading. Nevertheless, in this paper, we consider that there is only one remote server available or the server was randomly/previously selected. In fact, when multiple remote servers are considered, one can use 0-1 integer linear programming to model and solve the problem.

Once the main REE is defined, we use the heuristic presented in Algorithm 1. The idea is to simplify (4) by considering that $ET_{cloudlet} = ET_{cloud}$. In fact, our experiments have shown that public cloud instances are usually less or as powerful as cloudlets; therefore the waiting time and time spent transferring the data (which includes RTT) are the most relevant metrics for deciding where to offload. Therefore, we consider the average size of input and output data of all offloaded methods of the same application.

Once the remote server is known, the Offloading Decision Engine uses inequation (3) and the metrics estimated values to classify each instance of the historical data between two classes: *local* and *remote*. Such labeled data is used as the training set for creating the offloading decision tree that will assist the Decision Enforcer in the offloading decision. Since a decision tree learns from the given data set, the number of instances in the training data set may impact the decision tree generalization and prediction accuracy. Therefore, before creating the decision tree, our algorithm analyzes the historical data and considers different network conditions to create new instances for the training data set. In Algorithm 2, we present the pseudocode of the algorithm developed to handle the data and to create the decision tree.

The `CollectData` procedure (line (1)) gets the data from a database. `ClassifyInstance` (lines (4), (9), and (13)) classifies an instance as *local* or *remote*, based on inequality (3). In line (5), we duplicate the instance to use different network conditions. Thus, when the instance is classified as *local*, we check whether the classification would change when improving upload and download rates (lines (7) and (8)). Otherwise, when the instance is classified as *remote*, we check whether the classification would be the same, even when the network condition gets worse (lines (11) and (12)). Depending on the impact of the network condition on classification, we add both or only one instance to the training data set (lines (15) to (19)). Then, the `CreateDecisionTree` procedure (line (21)) creates a C4.5 decision tree [21]. In the current implementation, we use the J48 algorithm of the Java Weka library. (Weka: http://www.cs.waikato.ac.nz/ml/weka.)

It is important to highlight that a decision tree created for a specific pair of mobile device and remote server may

```
(1)  DataSet ⟵ CollectData(U, A)
(2)  TrainingSet ⟵ ∅
(3)  for all instance I ∈ DataSet do
(4)      I.class ⟵ ClassifyInstance(I)
(5)      N ⟵ I
(6)      if I.class = local then
(7)          N.upload ⟵ 2 × N.upload
(8)          N.download ⟵ 2 × N.download
(9)          N.class ⟵ ClassifyInstance(N)
(10)     else
(11)         N.upload ⟵ N.upload / 2
(12)         N.download ⟵ N.download / 2
(13)         N.class ⟵ ClassifyInstance(N)
(14)     end if
(15)     if I.class = N.class then
(16)         TrainingSet ⟵ N
(17)     else
(18)         TrainingSet ⟵ I, N
(19)     end if
(20) end for
(21) DecisionTree_{U,A} ⟵ CreateDecisionTree(TrainingSet)
```

Algorithm 2: Procedure for creating a decision tree of an application A and a user U.

not be suitable for a different mobile device or remote server, since any change in $E_{local,m}$ or $E_{rs,m}$ can affect the result of inequality (3) and therefore the decision tree created.

2.2. Handling Adaptive Monitoring. The algorithm used for building a C4.5 decision tree uses a top-down greedy search through the space of possible trees [21]. An important phase of the algorithm is the tree building, in which the decision model is built by making locally optimal decisions about which attribute to use for partitioning the data. Then, the tree grows in a recursive fashion by partitioning the training set into successively purer subsets.

The central choice in decision tree algorithms is selecting which attribute is most useful for classifying the data. There are many measures that can be used to determine the best attribute on which to split the data. The C4.5 algorithm uses a statistical property derived from information theory, called information gain, that measures how well a given attribute separates the training instances according to their target classification.

The information gain, $Gain(A, S)$, of an attribute A, relative to a data set S, is

$$Gain\,(A, S) = Ent\,(S) - \sum_{v \in A} \left(\frac{|S_v|}{|S|} Ent\,(S_v) \right) \quad (5)$$

where v represents any possible values of attribute A, S_v is the subset of S for which attribute A has value v (i.e., $S_v = \{s \in S \mid A(s) = v\}$), and $Ent(S)$ represents the entropy of the data set S. Entropy of any two-class (e.g., local and remote) data set D is defined as

$$Ent\,(D) = -\left(P_{D,L}\right) \log_2 \left(P_{D,L}\right) - \left(P_{D,R}\right) \log_2 \left(P_{D,R}\right) \quad (6)$$

where $P_{D,L}$ is the proportion of D belonging to class *Local* and $P_{D,R}$ is the proportion of D belonging to class *Remote*.

Our solution uses the concepts of entropy and information gain to identify the most relevant metrics for the offloading decision, as the decision tree creation algorithm. In doing so, we can dynamically adapt the *Monitoring Service* to monitor only such metrics, instead of monitoring all of them (like MpOS does). For instance, considering the decision tree depicted in Figure 4, the system only needs to monitor upload rate and RTT. Hence, the mobile device does not waste resources monitoring download rate and other metrics, at least until a new decision tree is created. The experiments presented in Section 3 show how this innovative scheme can conserve the battery life of handsets.

Figure 4 presents an example of an offloading decision tree for a dummy application. In this example, if the upload rate is equal to 200 Kbps and RTT is equal to 100 ms, the method `Bar` must be executed locally. Moreover, in all cases the method `Foo` must be executed on the remote server.

2.3. Handling Mobility. In order to support mobile devices mobility, our system needs to interconnect geographically distributed remote servers, therefore, allowing all of them to reach each other. The *OpenStack and Seamless Cloud Agent* is responsible for managing the OpenStack platform and the seamless cloud controller. The component is the interface between our solution and the underlying cloud and network platform.

By managing the SLC controller, our solution can seamlessly and programmatically interconnect multiple cloudlets and public cloud instances by creating a secure overlay network using IPSec over GRE tunnels. SLC uses Cisco's One

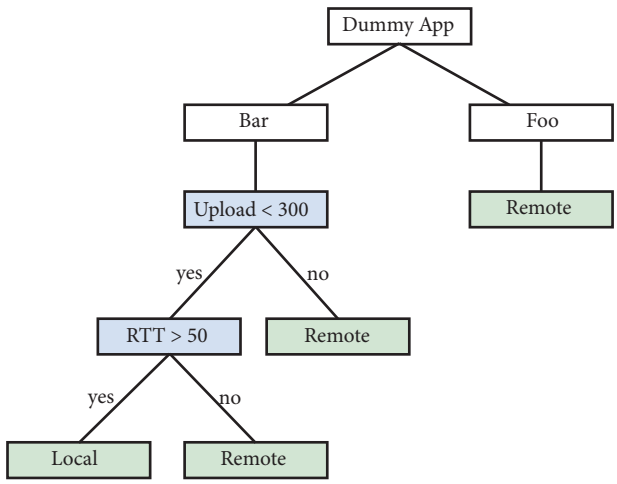


Figure 4: Example of offloading decision tree.

Platform Kit (onePK) or Network Configuration Protocol (NETCONF) to configure real or virtual equipment. In our current implementation, we use the SLC REST API to interact with the SLC controller, which leverages the NETCONF protocol to configure virtual routers (Cisco CSR 1000V) and interconnect different sites by creating a seamless network, which enables our system to migrate applications and offloaded computations between distributed remote servers in order to follow users displacements.

The *REE Handler* is the component responsible for managing the endpoint of the REE being used by the mobile application. When the mobile device leaves or enters a cloudlet coverage area, the endpoint is updated. According to the type of handover, the new endpoint points to a different cloudlet or public cloud.

The *Mobility Manager* is the component that detects when a mobile device enters or leaves a cloudlet coverage area. In these situations, the component triggers the migration of offloaded methods, as well as the entire application, when the Application Service related to the mobile application is not yet deployed and running on the destination cloud.

The *Mobility Manager* uses the Cisco wireless location services (Cisco MSE (http://www.cisco.com/c/en/us/products/wireless/mobility-services-engine/index.html.) and Cisco CMX (https://developer.cisco.com/site/cmx-mobility-services/.)) to monitor the location of mobile devices. In a nutshell, Cisco MSE collects discrete time, location, and MAC address of every device detected within the coverage area of network access points, even when mobile devices are not connected to the WLAN. Once a significant amount of data is collected, Cisco CMX assembles observations into a set of time-ordered points per device and processes data with movement analytics algorithms. Hence, we leverage the REST API exposed by Cisco MSE to track users' location.

Figure 5 presents a view of the components in a geographically distributed scenario. In the illustrated scenario, the SLC controller is executed in the public cloud and is used to configure all virtual routers. Once the seamless cloud network is configured, the Mobility Manager can use Cisco MSE to handle vertical and horizontal handovers.

3. Evaluation

This section presents the experiments performed to evaluate the proposed offloading solution. The experiments were planned to assess (i) the performance of mobile applications when using the proposed offloading solution under different scenarios, (ii) the impact of using decision trees in the decision module, (iii) the impact of using the adaptive monitoring service in the energy consumption of mobile devices, and (iv) the user's mobility support in scenarios composed of multiple cloudlets and public cloud.

3.1. Experiment 1: Using Decision Trees in the Decision Module. The objective of this experiment is to evaluate the impact of leveraging decision trees in the decision module of the proposed solution. The experiment aims to calculate the time needed to create decision trees in remote servers when varying the amount of historical data and number of clients and also the time required to parse decision trees on mobile devices when changing the tree height.

We consider a scenario composed of two remote servers and two mobile devices, which are described in Table 3.

3.1.1. Experiment 1.1: The Impact of the Decision Tree Creation. In order to assess the overhead of creating decision trees on remote servers, we developed a Java program that measures the time required to create trees under different conditions. We executed the experiment on two remote servers (a cloudlet and a public cloud instance), considering a variable number of clients (1, 2, 5, and 10) and historical records (10, 100, 1000, 10000, and 100000). The experiment was repeated 30 times for each client, to all possible combinations of remote server and amount of historical records.

Figure 6 presents the results of the experiment, the mean time for creating decision trees with 95% confidence interval. As we can see, the process of creating decision trees depends on the amount of historical records used. When the training data set is composed of 1000 or fewer historical records, the time needed to create decision trees is less than 25 ms, even when ten concurrent clients are considered. The time required for creating decision trees exceeds 1 second only when the largest training data set is used (100000 historical records), reaching approximately 2.5 seconds when decision trees for ten concurrent clients are created on the cloudlet remote server. The results also show that, in most cases, the time for creating decision trees on the cloud remote server is shorter than on the cloudlet.

This experiment indicates that the decision tree creation process is fast and lightweight. Even when ten concurrent clients are considered, remote servers with standard computing resources performed well with large data sets.

3.1.2. Experiment 1.2: The Impact of the Decision Tree Parsing. In order to assess the overhead of handling decision trees on mobile devices, we developed an Android application that measures the time required to deserialize and parse decision

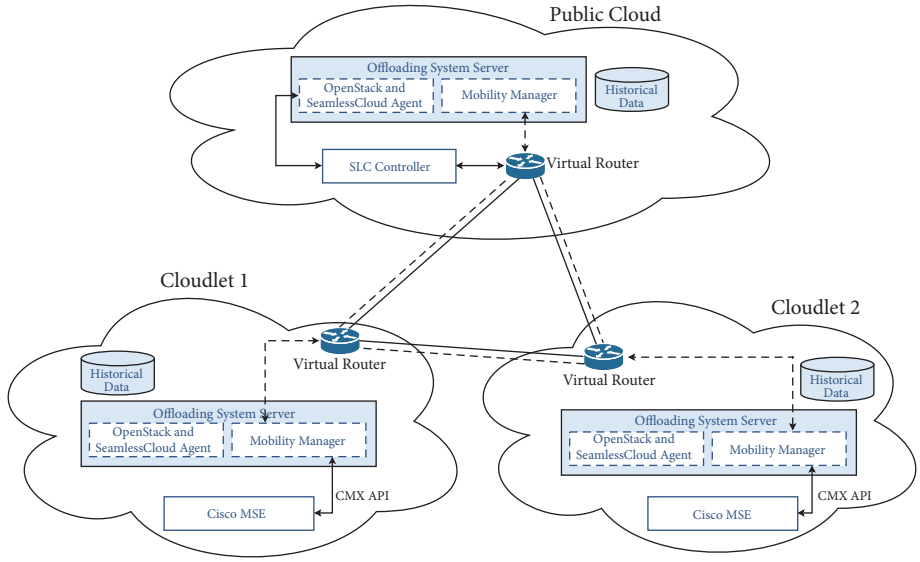


Figure 5: Overview of the components related to the user mobility support.

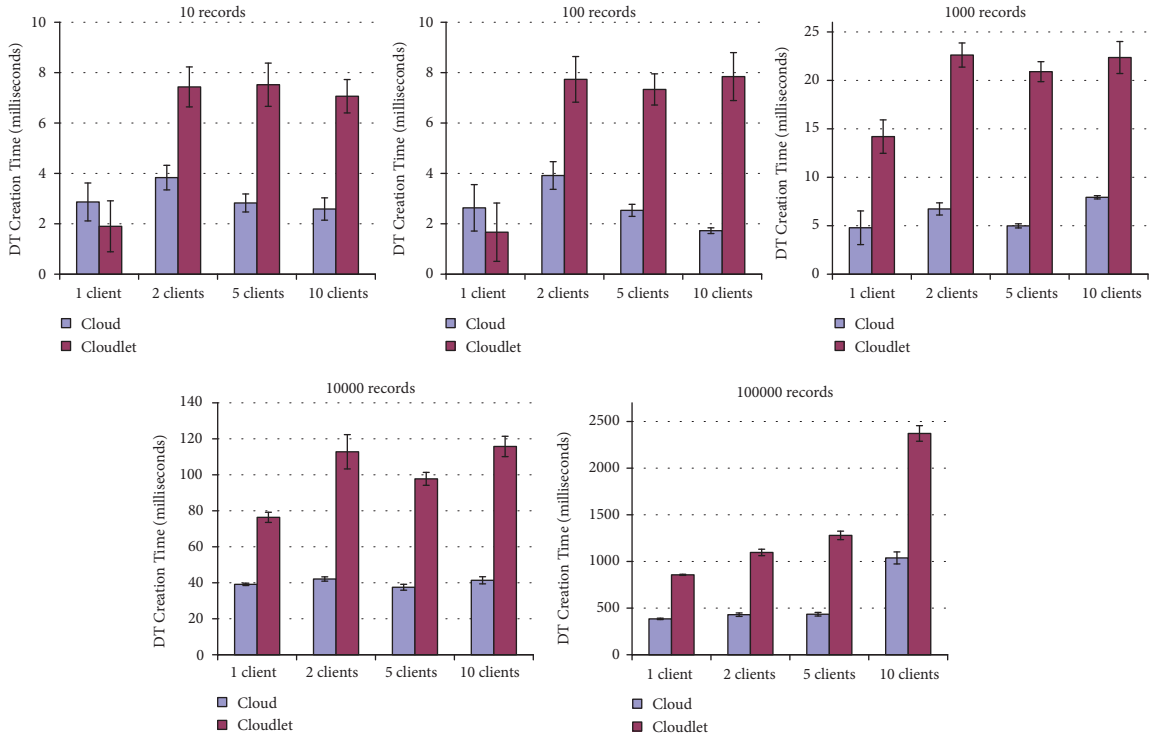


Figure 6: Decision trees' creation time (Experiment 1.1 results).

Table 3: Devices' configuration.

Mobile Device	Configuration
Handset A	Android 5.1, 4G capable, 3 GB RAM, and processor Qualcomm Snapdragon 808 (1.8 GHz 6-core).
Handset B	Android 4.1.2, 768 MB RAM, and processor Qualcomm MSM8225 Snapdragon (1 GHz dual-core).
Cloudlet	VM instance running on Laptop connected to a 802.11b/g network. Ubuntu Server 14.04, 2 VCPU, 4 GB RAM.
Public Cloud	VM instance running on Cisco Intercloud Services (CIS), Amsterdam datacenter. Ubuntu Server 14.04, general purpose medium instance (1 VCPU, 4 GB RAM).

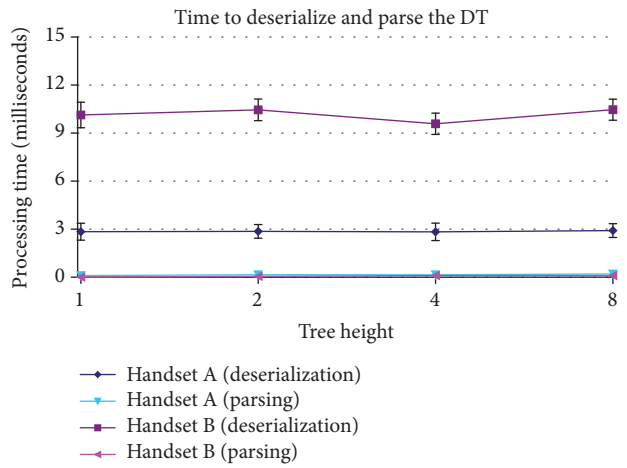


Figure 7: Time needed to deserialize and parse decision trees (Experiment 1.2 result).

trees with different heights. We executed the experiment on two mobile devices (handset A and handset B), considering different tree heights (1, 2, 4, and 8). The experiment was repeated 100 times for each combination of mobile device and tree height.

It is important to highlight that the deserialization process is executed only when the mobile device receives a decision tree, which is asynchronously sent from the remote server, while the parsing process is executed whenever a "remotable" method is called.

Figure 7 presents the mean time for deserializing and parsing decision trees with 95% confidence interval. As we can see, regardless of the tree height, deserialization and parsing times slightly vary. Handset A takes approximately 3 ms for deserializing decision trees, while handset B takes about 10 ms. Now, regarding the parsing time, both handsets take in average less than 1 ms to parse decision trees, even for the ones with the height equal to 8.

These results are important to show how lightweight is to handle decision trees on mobile devices. The parsing time is particularly relevant because the decision tree parsing process is executed whenever a "remotable" method is called. Thus, we show that the idea of using decision trees to assist the offloading decision is not overloading the system.

3.2. Experiment 2: Applications Performance When Offloading. The objective of this experiment is to assess the performance of two applications when using the proposed offloading solution under different scenarios. The experiment aims to calculate the execution time of methods when they are executed locally on mobile devices and when they are executed on remote servers (i.e., when offloading).

As in Experiment 1, we used the mobile devices handset A and handset B and the remote servers cloudlet and public cloud, described in Table 3. And in addition to the performance evaluation, we used the Monsoon power monitor (Monsoon power monitor website: https://www.msoon.com/LabEquipment/PowerMonitor.) to measure the energy consumption of smartphones during the experiment.

Several types of applications have been used in the literature to evaluate offloading solutions. In this regard, [22] performed a systematic mapping to catalog the most commonly used applications in mobile cloud computing research papers. The authors identified that image processing and mathematical tools are the categories of applications most used, which is consistent with what we have identified in our literature review. Therefore, we rely on such findings and use the applications BenchImage and MatrixOperations. The former is an image processing application that allows users to apply filters to pictures with different resolutions (8 MP, 4 MP, 2 MP, 1 MP, and 0.3 MP). The application provides the filters Sharpen, Cartoonizer, and Red Tone, which have different computation requirements and therefore different execution times [3]. The latter is an application to perform common operations with matrices, such as addition and multiplication of matrices.

Since the applications are different and have distinct configuration options, we present details of the experiments in separate subsections.

3.2.1. Experiment 2.1: BenchImage. We executed the BenchImage application on handset A and handset B. The methods Maptone and Cartoonizer were executed 30 times locally on smartphones and also offloaded to the cloudlet and public cloud remote servers with different picture resolutions (1 MP, 2 MP, 4 MP, and 8 MP). Handset A used both Wi-Fi and 4G LTE to perform offloading to the public cloud, while handset B used only Wi-Fi because the smartphone does not support 4G.

Tables 4 and 5 present the mean execution time of the methods Maptone and Cartoonizer with 95% confidence interval for handsets A and B, respectively. As we can see in both tables, the best results are achieved when performing offloading to the cloudlet. In such a case, handset A reduces the execution time of the method Maptone approximately 2.5 times, while the method Cartoonizer is executed approximately 3 times faster when offloaded to the cloudlet (when the 8 MP picture is considered). When analyzing the results for handset B, we can see that the methods Maptone and Cartoonizer are, respectively, executed approximately 5 and 11 times faster when offloaded to the cloudlet (when the 8 MP picture is considered). The difference in results is due to the fact that handset B is less powerful than handset A.

Since handset A is a powerful device, we can see in Table 4 that offloading to the cloud is not always worth it. In other words, it is better to execute the method locally depending on the picture resolution and the connection being used (e.g., when executing the method Maptone on cloud, via 4G, with a 2 MP picture; or when executing the method Cartoonizer on cloud, via Wi-Fi, with a 1 MP picture). On the other hand, Table 5 shows that the performance of handset B is so poor that performing offloading to the cloud using either 4G or Wi-Fi always speeds up the method execution. Therefore, this experiment reinforces the importance of the offloading decision, which has to consider the context of mobile devices among other variables, as discussed in Section 2.1.

Table 4: Results of the BenchImage experiment for handset A.

Method	Where	Execution time for different resolutions (seconds)			
		1 MP	2 MP	4 MP	8 MP
Maptone	Locally	1.24 ± 0.01	1.94 ± 0.006	3.78 ± 0.009	10.74 ± 0.06
	Cloudlet	0.61 ± 0.01	1.04 ± 0.01	1.65 ± 0.01	4.15 ± 0.05
	Cloud (4G)	4.43 ± 0.07	6.33 ± 0.05	7.56 ± 0.23	14.48 ± 0.28
	Cloud (Wi-Fi)	2.19 ± 0.04	3.15 ± 0.07	3.85 ± 0.12	7.45 ± 0.29
Cartoonizer	Locally	2.41 ± 0.009	4.35 ± 0.009	8.3 ± 0.01	19.79 ± 0.05
	Cloudlet	0.9 ± 0.02	1.54 ± 0.009	2.85 ± 0.02	6.61 ± 0.1
	Cloud (4G)	5.36 ± 0.04	8.06 ± 0.14	11.34 ± 0.52	18.58 ± 0.33
	Cloud (Wi-Fi)	2.57 ± 0.03	3.64 ± 0.06	5.62 ± 0.19	10.3 ± 0.39

Table 5: Results of the BenchImage experiment for handset B.

Method	Where	Execution time for different resolutions (seconds)			
		1 MP	2 MP	4 MP	8 MP
Maptone	Locally	5.21 ± 0.02	9.09 ± 0.05	16.82 ± 0.01	34.98 ± 0.03
	Cloudlet	0.84 ± 0.04	2.2 ± 0.14	2.69 ± 0.1	6.88 ± 0.18
	Cloud (Wi-Fi)	3.36 ± 0.16	5.63 ± 0.4	7.46 ± 0.41	24.78 ± 1.28
Cartoonizer	Locally	15.74 ± 0.01	29.05 ± 0.01	57.85 ± 0.2	118.24 ± 0.5
	Cloudlet	1.43 ± 0.2	2.01 ± 0.08	3.5 ± 0.17	10.66 ± 0.9
	Cloud (Wi-Fi)	4.57 ± 0.23	6.19 ± 0.22	10.75 ± 0.47	27.94 ± 1.29

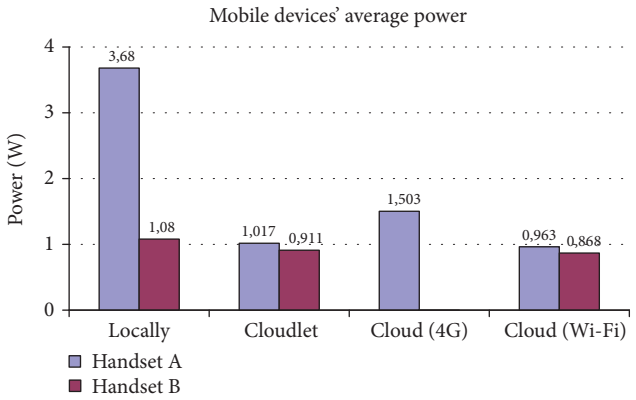


Figure 8: Mobile devices' average power during the BenchImage experiment.

We used the power monitor to measure the energy consumption of smartphones during the experiment. Figure 8 shows the average power of handsets A and B when executing the methods on different locations. As we can see, for both mobile devices, the average energy consumed when a method is executed locally is greater than when a method is offloaded. Moreover, the average power consumed when handset A uses 4G is greater than when it uses Wi-Fi.

Using the average values presented in the figure, we can do a fair estimation of how much energy the mobile devices consume when executing a method, by multiplying the average power by the method's execution time. For instance, handset A consumes approximately 16J (3.68W × 4.35s) when executing the method Cartoonizer locally for a 2 MP picture, while it consumes approximately 12.11J (1.503W × 8.06s) when executing the same method with the same picture on the cloud using 4G.

3.2.2. Experiment 2.2: MatrixOperations. We executed the MatrixOperations application on handset A and handset B. The methods Add and Multiply were executed locally on smartphones and offloaded to the cloudlet and the public cloud remote servers. As in Experiment 2.1, handset A used both Wi-Fi and 4G LTE to perform offloading to the public cloud, while handset B used only Wi-Fi.

We considered different dimensions for the matrices (200×200, 400×400, 600×600, 800×800, and 1000×1000) during the experiment, and each method was executed locally as well as offloaded to remote servers, being repeated 30 times for each combination of mobile device and dimensions of the matrices.

Tables 6 and 7 present the mean execution time of the methods Add and Multiply with 95% confidence interval for handsets A and B, respectively. As we can see in both tables, the method Add is faster when executed locally because performing the addition of two matrices is a quite simple operation. On the other hand, when the method Multiply is considered, the best results are achieved in most cases when performing offloading to the cloudlet. In fact, handset B reduces the execution time of the method Multiply approximately 9 times when the larger matrix dimension is considered (1000x1000).

Handset A also performs well when offloading to the cloudlet, achieving a speedup of approximately 4 times when the larger matrix dimension is considered. Nevertheless, it is important to highlight two cases: (1) it is faster to execute the multiplication locally when considering 200x200 matrices and (2) sometimes it is faster to perform offloading to the

Table 6: Result of the experiment: execution time for different cases on handset A.

Method	Where	Execution time for matrices with dimension $n \times n$ (seconds)				
		$n = 200$	$n = 400$	$n = 600$	$n = 800$	$n = 1000$
Add	Locally	0.008 ± 0.001	0.01 ± 0.001	0.02 ± 0.001	0.04 ± 0.001	0.07 ± 0.002
	Cloudlet	0.44 ± 0.02	1.40 ± 0.05	3.3 ± 0.1	5.39 ± 0.16	8.72 ± 0.25
	Cloud (4G)	3.97 ± 0.10	8.06 ± 0.25	10.28 ± 0.25	15.02 ± 0.41	20.22 ± 0.55
	Cloud (Wi-Fi)	2.57 ± 0.06	5.15 ± 0.1	8.27 ± 0.59	9.83 ± 0.24	12.99 ± 0.55
Multiply	Locally	0.31 ± 0.003	4.69 ± 0.01	18.19 ± 0.28	45.96 ± 0.44	96.38 ± 1.07
	Cloudlet	0.46 ± 0.01	1.69 ± 0.1	6.12 ± 0.22	13.07 ± 0.64	24.54 ± 0.29
	Cloud (4G)	3.68 ± 0.09	7.96 ± 0.18	11.24 ± 0.39	17.52 ± 0.46	23.86 ± 0.99
	Cloud (Wi-Fi)	2.54 ± 0.06	5.24 ± 0.08	7.85 ± 0.27	12.27 ± 0.29	16.79 ± 0.71

Table 7: Result of the experiment: execution time for different cases on handset B.

Method	Where	Execution time for matrices with dimension $n \times n$ (seconds)				
		$n = 200$	$n = 400$	$n = 600$	$n = 800$	$n = 1000$
Add	Locally	0.02 ± 0.002	0.07 ± 0.001	0.12 ± 0.001	0.15 ± 0.001	0.24 ± 0.003
	Cloudlet	0.89 ± 0.08	1.95 ± 0.06	3.86 ± 0.23	6.76 ± 0.21	11.07 ± 0.67
	Cloud (Wi-Fi)	3.53 ± 0.12	10.56 ± 0.63	19.62 ± 1.16	32.95 ± 2.2	52.38 ± 2.86
Multiply	Locally	0.99 ± 0.003	9.67 ± 0.003	38.51 ± 0.14	100.40 ± 0.14	218.95 ± 0.18
	Cloudlet	0.72 ± 0.04	2.13 ± 0.06	6.59 ± 0.26	12.64 ± 0.12	24.14 ± 0.57
	Cloud (Wi-Fi)	3.67 ± 0.16	10.26 ± 0.56	17.63 ± 1.19	35.99 ± 1.97	48.72 ± 1.97

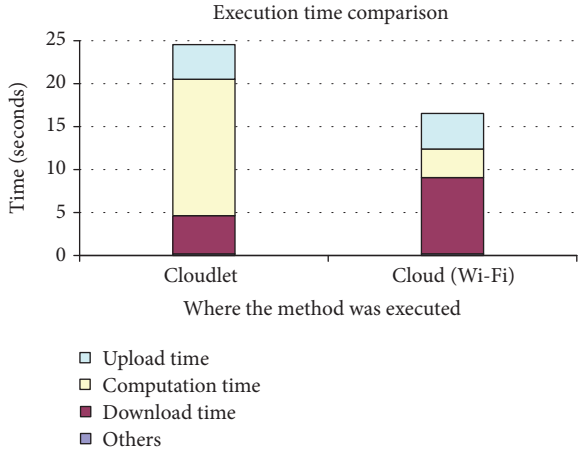


Figure 9: Execution time comparison of two observations of handset A.

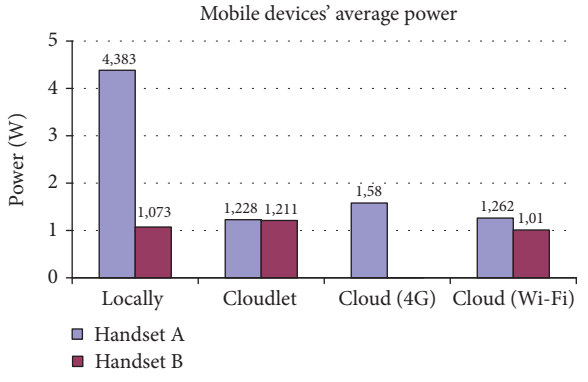
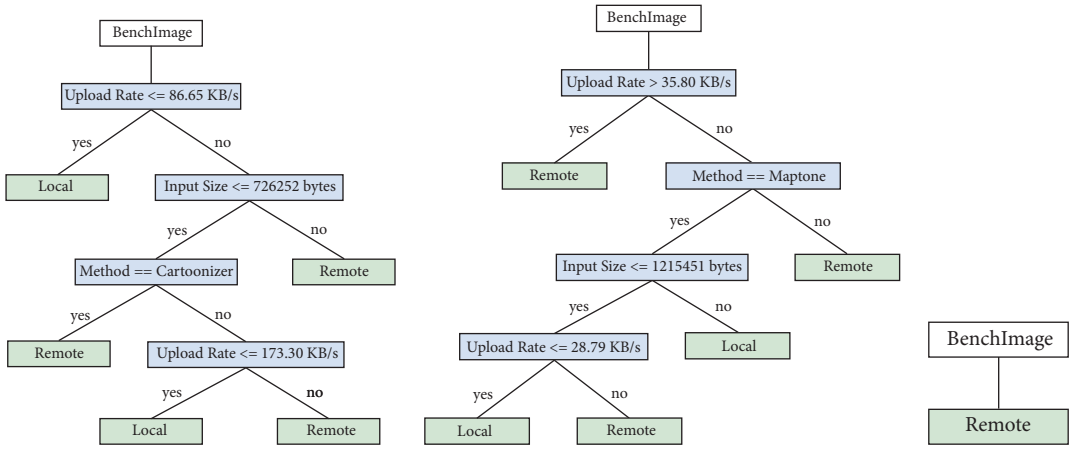


Figure 10: Mobile devices' average power during the MatrixOperations experiment.

cloud using Wi-Fi than offloading to the cloudlet (e.g., when the larger matrix dimension is considered). Figure 9 helps to explain the latter case, by showing the main variables that compose the total offloading time for two observations collected during the experiment. As we can see, the cloud outperforms the cloudlet when comparing the computation time of the matrix multiplication. Thus, despite the short download and upload times when offloading to the cloudlet, it is better to perform offloading to the cloud. These results reinforce how important the context of mobile devices is to the offloading decision.

We also used the power monitor to measure the energy consumption of smartphones during this experiment. As in the last experiment, to simplify and expedite the measurement, we monitored the average power of mobile devices during the MatrixOperations execution in benchmark mode (i.e., during the 30 executions of the methods Add and Multiply for each matrix dimension). Thus, Figure 10 shows the average power of handsets A and B when executing the methods on different locations. As we can see, for handset A, the average energy consumed when a method is executed locally is greater than when it is offloaded. On the other hand, for handset B, the average energy consumed is greater when a method is executed on cloudlet.

Using the average values presented in the figure, we can do a fair estimation of how much energy the mobile devices consume when executing a method, by multiplying the average power by the method's execution time. For instance, handset A consumes approximately 20.5J (4.38W $*$ 4.69s) when executing the method Multiply locally for 400x400



(a) Handset A - cloud (b) Handset B - cloud (c) Handsets A and B - cloudlet

Figure 11: Decision trees created for handsets A and B when connected to the cloud or the cloudlet.

matrices, while it consumes approximately 12.5J (1.58W ∗ 7.96s) when executing the same method for the same matrix dimension on the cloud using 4G.

3.3. Experiment 3: Adaptive Monitoring Service Evaluation. The objective of this experiment is to assess the impact of using the adaptive monitoring service in the energy consumption of mobile devices. The experiment aims to compare the energy consumed by mobile devices when they are using or not the adaptive monitoring service.

To perform the experiment, we launched the BenchImage application on handset A and handset B (described in Table 3) with different configurations for the adaptive monitoring service (enabled and disabled), and we considered that the application could connect to the remote servers cloudlet or public cloud. Then we used the Monsoon power monitor to measure the energy consumption of the mobile devices for 100 seconds while they were running the BenchImage application.

This experiment was performed after Experiment 2, so the remote servers created decision trees based on the offloading historical records, and the trees were asynchronously sent to mobile devices. It is also important to highlight that no method was executed on mobile devices during the measurement to avoid affecting the results. Thus, in all experiments, the power values were taken with the screen on, 50% display brightness, Bluetooth disabled, minimal background application activity, and no foreground activity.

Figure 11 presents the offloading decision trees created for the BenchImage application when handsets A and B are connected to the cloudlet and public cloud remote servers. As we can see, when both mobile devices are using the cloudlet, regardless of the handset context, all methods must be offloaded (Figure 11(c)) , which is a quite straightforward offloading decision. On the other hand, when mobile devices are using the cloud, the created offloading decision trees are more complex, depending on the metrics: method name, upload rate, and method's input size.

When the adaptive monitoring service is disabled, the offloading framework monitors several metrics, such as upload rate, download rate, latency, and wireless RSS. In contrast, when the adaptive monitoring service is enabled, the decision tree is used to define which metrics must be monitored, since the idea is to monitor only the metrics relevant to the offloading decision. For this experiment, the Monitoring Service is configured to measure each metric every 30 seconds.

Figure 12 compares the power consumption variation of handsets A and B during the experiment for each scenario (adaptive monitoring disabled and adaptive monitoring enabled with the mobile device connected to the cloudlet and public cloud remote servers). As we can see, when the adaptive monitoring service is disabled, both mobile devices consume more energy. In turn, the best case scenario is when the adaptive monitoring is enabled and mobile devices are connected to the cloudlet because, according to the decision trees presented in Figure 11(c), no metric needs to be monitored.

Figure 13 presents the total energy consumed by mobile devices during the experiment. When the adaptive monitoring service is disabled, handset A consumes approximately 2 times more energy than when the adaptive monitoring service is enabled, while handset B consumes approximately 80% more energy. In turn, when the adaptive monitoring service is enabled, the difference in energy consumption reaches 10% when comparing mobile devices connected to the cloud and cloudlet. In such a case, the difference is caused by the overhead of monitoring the metric upload rate, which is a metric relevant to the offloading decision when mobile devices are connected to the cloud.

As a result, since the offloading framework only needs to monitor the relevant metrics for the offloading decision, we can reduce the overhead of monitoring the entire system and consequently reduce the energy consumption of mobile devices.

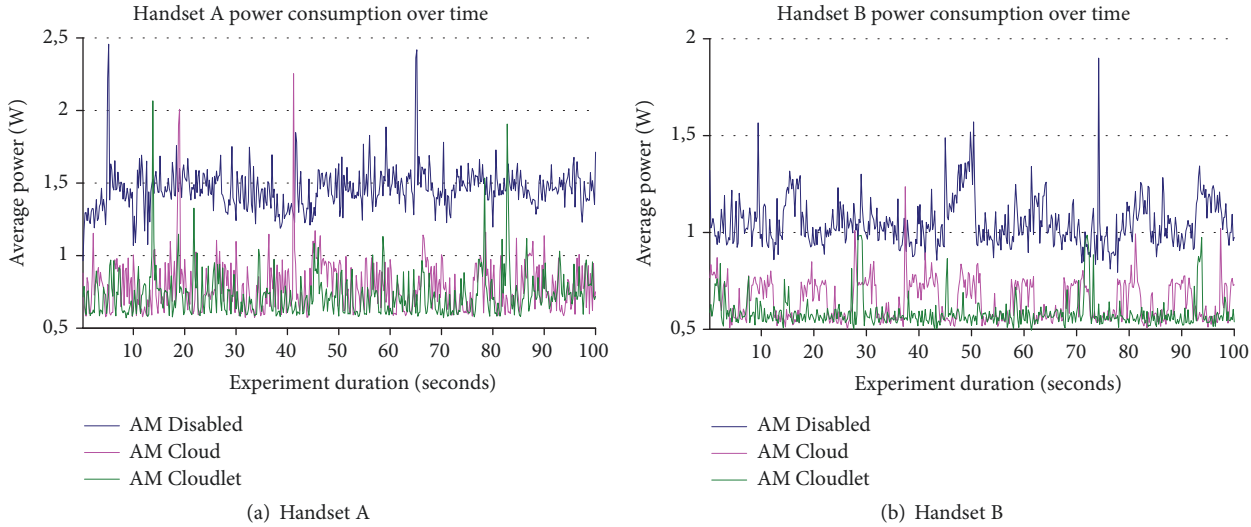


Figure 12: Mobile devices' power consumption during the experiment. AM Disabled: adaptive monitoring disabled. AM cloudlet: handset connected to the cloudlet and adaptive monitoring enabled. AM cloud: handset connected to the cloud and adaptive monitoring enabled.

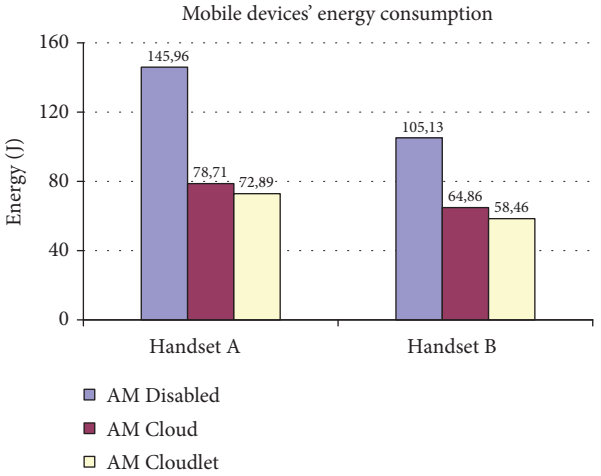


Figure 13: Result of the experiment: mobile devices' energy consumption.

3.4. Experiment 4: Mobility Support. The objective of this experiment is to evaluate the proposed solution for supporting users' mobility in hybrid scenarios composed of multiple cloudlets and public cloud. We used an application called CameraOffloading, which allows a user to take a picture and apply filters called Effect 1, Effect 2, and Effect 3 to the picture taken.

This experiment was performed in a testbed configured in the Cisco campus at San Jose, California, United States. This testbed leverages the Cisco wireless network and is composed of two cloudlets and a public cloud instance running in a Cisco datacenter at Amsterdam, Netherlands. The configuration of the mobile device and remote servers used is presented in Table 8.

The filters were applied in 2.4 MP pictures taken during a walk across the Cisco campus. Figure 14 shows the map of the campus, detailing the path taken during the walk and the locations where the pictures were taken. Depending on the location, handset C used Wi-Fi or 3G to perform offloading, respectively, to cloudlets and public cloud.

Figure 15 presents a graph with the total execution time of the methods executed during the experiment. Naturally, the execution time when we perform offloading to the public cloud is longer than when we use cloudlets. But as expected, the framework handled user's displacements, and the execution flow of the mobile application was not affected despite the changes in the REE endpoint, which were transparent to the user.

4. Related Work

In the last few years, several studies have been developed in response to the challenges offered by MCC [3–10], and with them different approaches for performing computation offloading of mobile applications were proposed. MpOS [3], MAUI [4], ThinkAir [5], MobiCOP [8], Foreseer [9], and CSOS [10] use methods as offloading units, while CloneCloud [6] and EMCO [7] migrate threads to a remote server. Since our solution extends MpOS, we also work at method granularity.

Regarding the offloading decision, MobiCOP, ThinkAir, and MpOS execute all operations related to the decision on the mobile device. On the other hand, CloneCloud, MAUI, EMCO, and CSOS are some of the few works that execute the complex operations of the decision component outside the mobile device. Since the overall idea is to offload data and computation from mobile devices to remote servers, it seems reasonable to avoid storing profiling information and traces of offloaded tasks on mobile devices. Besides that, we can leverage a cloud/cloudlet to store such data as well as to perform the compute-intensive tasks related to the offloading decision.

Figure 14: Map of the Cisco campus testbed that was used in the vertical handover experiment. The path starts in the green user symbol (in building SJC-15) and ends in the red user symbol (in building SJC-13). The yellow circles indicate the coverage area of cloudlets.

Table 8: Configuration of the equipment used in the vertical handover experiment.

Equipment	Description
Cloudlet SJC-15	VM instance running on Laptop connected to a 802.11b/g/n network. Ubuntu Server 14.04, 4 VCPUs, 4 GB RAM.
Cloudlet SJC-13	VM instance running on an Intel MiniPC connected to a wired network. Ubuntu Server 14.04, 2 VCPUs, 2 GB RAM.
Public Cloud	VM instance running on Cisco Intercloud Services (CIS), Amsterdam datacenter. Ubuntu Server 14.04, general purpose medium instance (1 VCPU, 4 GB RAM).
Handset C	Android 4.1.0, 1 GB RAM, and processor ARM Cortex A9 (1 GHz dual-core)

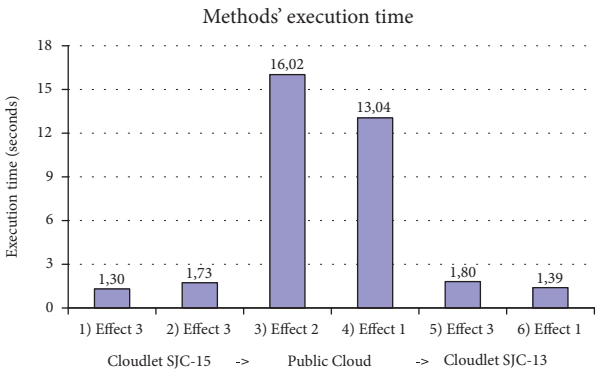


Figure 15: Result of the experiment: execution time of methods with vertical handover.

Some solutions, such as CloneCloud and CSOS, rely on an offline phase in order to train the decision algorithm. In CloneCloud, during the development phase, developers must collect traces of the application being executed under different network and hardware conditions to create a database of possible ways to partition the application. Once the traces are collected, CloneCloud solves a standard integer linear problem to decide which precomputed partition must be offloaded.

In CSOS, the developer must run offline experiments to generate a data set of executions that will be used to train classifiers models, which must be included in the application in order to enable the offloading system to decide at runtime when to offload. The offline training process is a burden for the developer and generates a model that is heavily dependent on the offline experiment context (e.g., handsets and remote servers used, network condition). In our solution, remote servers create a unique decision tree for each mobile device based on its profiling data.

MAUI and EMCO are more similar to our proposal. MAUI solves a 0-1 integer linear programming problem on the remote server to decide where each method must be executed and periodically updates the mobile device partition information. Since MAUI's objective is to save energy, the solution depends on mechanisms to identify the energy consumption of each method. On the other hand, EMCO proposes the use of a fuzzy logic system to aggregate the profiling metrics and use historical data for building on the remote server an inference system that can be used by the mobile device to classify where the threads must be executed.

Foreseer [9] is one of the few works that investigates the overhead caused by frequently measuring metrics such as upload and download rate of a mobile device. To reduce the data transmission and battery consumption caused by profiling operations, the authors exploit crowdsourcing to collect the users trajectories and bandwidth in their locations and learn the probabilistic model of network bandwidth conditioned on the location. Foreseer then dynamically decides when to offload based on the network status.

As shown by related works, historical data can be used to improve the offloading decision, especially by using classification mechanisms. In this regard, our solution innovates by leveraging decision trees to instrument the offloading decision of when each method must be offloaded. In addition, to the best of our knowledge, the idea of using an adaptive monitoring scheme to monitor only the metrics that are relevant to the offloading decision is unique and saves energy, prolonging the battery life of mobile devices.

Regarding the mobility support, few solutions have addressed the offloading continuity when users are moving in a hybrid environment composed of cloudlets and public cloud instances. The authors of [23] propose a solution that considers users' mobility when deciding where to perform offloading. The decision module is executed on mobile devices, which receive periodical information regarding all available servers and communication latency, and defines where to offload based on received information. In order to handle server transitions, the origin server, where the user was previously connected, sends results of existing computations to the destination server, which forwards results to the mobile device. The solution only considers mobility between public clouds, not considering hybrid environments.

In [24], the authors propose a system architecture composed of three remote execution environments (cloud, cloudlet, and other mobile devices). The solution provides a multicriteria offloading decision algorithm to select where to offload tasks and a handover strategy to move offloaded tasks between REEs, aiming to reduce energy consumption. Mobile devices execute offloading decisions and the solution assumes that all resources can connect each other. Besides that, the solution considers that applications are already running on all resources. This is one of the few works that considers user's mobility between cloudlets and public cloud.

The authors of [20] also use public cloud and cloudlets as REEs. They propose an architecture based on ThinkAir and an offloading algorithm that aims to decide whether to perform offloading to clones running in a public cloud or a cloudlet. The offloading decision takes into consideration the energy consumption for offloaded methods and network context while satisfying certain response time constraints. Besides, the solution considers users' disconnection and handles users' mobility by migrating data between public clouds and cloudlets. Nevertheless, it is not clear how the proposed mechanism will be implemented in practice since authors implemented the solution using the simulator CloudSim [25], which is not properly designed to simulate network events.

MOSys [26] supports user's mobility while compute-intensive methods are offloaded to a cloudlet/cloud. The solution is based on MpOS and leverages OpenFlow rules to seamlessly switch ongoing sessions when a mobile device moves between access points. Besides, the authors use data caching to improve offloading performance. The solution only considers mobility between access points in a cloudlet-based scenario, not considering hybrid environments composed of multiple cloudlets and public cloud, where users can offload to any of the remote execution environments using Wi-Fi or mobile networks (e.g., 3G or 4G).

Current computational offloading techniques are subject to several challenges inherent to wireless networks and mobile devices mobility, such as network disruption, latency, and packet loss. In the face of such challenges, our offloading framework leverages network programmability to interconnect geographically distributed cloudlets and public cloud instances in order to handle users mobility. Table 9 summarizes the comparison to the aforementioned related works. We highlight six features to differentiate the works: offloading granularity, offline training dependency, decision module location, type of profiling used, mobility support, and scenario supported.

5. Conclusion and Future Work

This paper presented novel approaches for handling the offloading decision, performing adaptive monitoring, and supporting users' mobility on mobile cloud computing systems. We introduced a mathematical model and algorithms that leverage profiling information and historical data to create decision trees for assisting the offloading decision. Different from related works, all compute-intensive operations related to the creation of a decision tree are performed on remote servers, while mobile devices only have to parse the tree to make a decision. The results of the experiments show that the decision tree creation process on remote servers is fast and lightweight, as well as the process of parsing decision trees on mobile devices.

Also, we develop a solution to reduce the burden of monitoring the metrics related to the offloading decision by using statistical concepts of information gain and entropy. The experiments showed that, in some cases, it is possible to reduce the energy consumption of a mobile device up to 50% when the adaptive monitoring service is enabled.

Finally, we implemented a scheme to handle users' mobility in a scenario composed of multiple cloudlets and public cloud instances, where users can perform horizontal and vertical handovers. The experiments show that the proposed solution supports the most variate scenarios of user's mobility and can perform offloading to different remote servers, transparently to the user, while maintaining the correct execution flow of the mobile application without interfering with the user's decisions and displacements.

As the next steps of our research, we intend to investigate the scalability of remote servers for mobile cloud computing applications and the use of crowdsensing techniques to improve the offloading decision.

Table 9: Comparison to the related works.

Offloading Solution	Offloading Granularity	Offline Training Dependency	Decision Module Location	Type of Monitoring (Profiling)	Mobility Support	Supported Mobility Scenario
MpOS [3]	Method	No	Device	All metrics	No	-
MAUI [4]	Method	No	Remote Server and Device	All metrics	No	-
ThinkAir [5]	Method	No	Device	All metrics	No	-
CloneCloud [6]	Thread	Yes	Device	All metrics	No	-
EMCO [7]	Thread	No	Remote Server and Device	All metrics	No	-
CSOS [10]	Method	Yes	Remote Server and Device	All metrics	No	-
MobiCOP [8]	Method	No	Device	All metrics	No	-
Foreseer [9]	Method	No	Device	All metrics (Crowdsourcing)	No	-
[23]	Method	No	Device	All metrics	Yes	Public Cloud
[24]	Method	No	Device	All metrics	Yes	Cloudlets and Public Cloud
[20]	Method	No	Device	All metrics	Yes	Cloudlets and Public Cloud
MOSys	Method	No	Device	All metrics	Yes	Cloudlets
Our Work	Method	No	Remote Server and Device	Relevant metrics (Adaptive)	Yes	Cloudlets and Public Cloud

Data Availability

Applications and experimental data are available from the corresponding author (Paulo Rego) upon request.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This research is partially supported by the Brazilian National Council for Scientific and Technological Development (CNPq) under grant numbers 477223/2012-5 and 201986/2014-0 and by the São Paulo Research Foundation (FAPESP) under grant number 2015/24144-7. We would like to acknowledge Cisco Systems for the support provided during the first author's internship.

References

[1] L. Li, X. Zhang, K. Liu, F. Jiang, and J. Peng, "An energy-aware task offloading mechanism in multiuser mobile-edge cloud computing," *Mobile Information Systems*, vol. 2018, Article ID 7646705, 12 pages, 2018.

[2] H. T. Dinh, C. Lee, D. Niyato, and P. Wang, "A survey of mobile cloud computing: Architecture, applications, and approaches," *Wireless Communications and Mobile Computing*, vol. 13, no. 18, pp. 1587–1611, 2013.

[3] P. B. Costa, P. A. Rego, L. S. Rocha, F. A. Trinta, and J. N. de Souza, "MpOS: a multiplatform offloading system," in *Proceedings of the 30th Annual ACM Symposium on Applied Computing (SAC '15)*, pp. 577–584, ACM, Salamanca, Spain, April 2015.

[4] E. Cuervoy, A. Balasubramanian, D.-K. Cho et al., "MAUI: making smartphones last longer with code offload," in *Proceedings of the 8th Annual International Conference on Mobile Systems, Applications and Services (MobiSys '10)*, pp. 49–62, ACM, New York, NY, USA, June 2010.

[5] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, "Thinkair: dynamic resource allocation and parallel execution in the cloud for mobile code offloading," in *Proceedings of the IEEE INFOCOM*, pp. 945–953, March 2012.

[6] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "CloneCloud: elastic execution between mobile device and cloud," in *Proceedings of the 6th ACM EuroSys Conference on Computer Systems (EuroSys '11)*, pp. 301–314, ACM, New York, NY, USA, April 2011.

[7] H. R. Flores Macario and S. Srirama, "Adaptive code offloading for mobile cloud applications: Exploiting fuzzy sets and evidence-based learning," in *Proceeding of the 4th ACM Workshop on Mobile Cloud Computing and Services (MCS '13)*, pp. 9–16, ACM, New York, NY, USA, June 2013.

[8] J. I. Benedetto, G. Valenzuela, P. Sanabria, A. Neyem, J. Navón, and C. Poellabauer, "MobiCOP: a scalable and reliable mobile code offloading solution," *Wireless Communications and Mobile Computing*, vol. 2018, Article ID 8715294, 18 pages, 2018.

[9] L. Yang, J. Cao, S. Tang, D. Han, and N. Suri, "Run time application repartitioning in dynamic mobile cloud environments," *IEEE Transactions on Cloud Computing*, vol. 4, no. 3, pp. 336–348, 2016.

[10] W. Junior, E. Oliveira, A. Santos, and K. Dias, "A context-sensitive offloading system using machine-learning classification algorithms for mobile cloud environment," *Future Generation Computer Systems*, vol. 90, pp. 503–520, 2019.

[11] N. Fernando, S. W. Loke, and W. Rahayu, "Mobile cloud computing: a survey," *Future Generation Computer Systems*, vol. 29, no. 1, pp. 84–106, 2013.

[12] M. Shiraz, A. Gani, R. H. Khokhar, and R. Buyya, "A review on distributed application processing frameworks in smart mobile devices for mobile cloud computing," *IEEE Communications Surveys & Tutorials*, vol. 15, no. 3, pp. 1294–1313, 2013.

[13] S. Abolfazli, Z. Sanaei, and A. Gani, "Mobile cloud computing: A review on smartphone augmentation approaches," *CoRR*, 2012, https://arxiv.org/abs/1205.0451.

[14] P. A. L. Rego, P. B. Costa, E. F. Coutinho, L. S. Rocha, F. A. M. Trinta, and J. N. D. Souza, "Performing computation offloading on multiple platforms," *Computer Communications*, vol. 105, pp. 1–13, 2017.

[15] P. A. L. Rego, E. Cheong, E. F. Coutinho, F. A. M. Trinta, M. Z. Hasany, and J. N. D. Souza, "Decision tree-based approaches for handling offloading decisions and performing adaptive monitoring in MCC systems," in *Proceedings of the 5th IEEE International Conference on Mobile Cloud Computing, Services and Engineering (MobileCloud '17)*, pp. 74–81, April 2017.

[16] E. Baccarelli, P. G. V. Naranjo, M. Scarpiniti, M. Shojafar, and J. H. Abawajy, "Fog of everything: energy-efficient networked computing architectures, research challenges, and a case study," *IEEE Access*, vol. 5, pp. 9882–9910, 2017.

[17] P. A. L. Rego, *Applying smart decisions, adaptive monitoring and mobility support for enhancing offloading systems [Ph.D. Thesis]*, Federal University of Ceará, 2016.

[18] M. Z. Hasan, M. Morrow, L. Tucker, S. L. D. Gudreddi, and S. Figueira, "Seamless cloud abstraction, model and interfaces," in *Proceedings of the ITU - Fully Networked Human? - Innovations for Future Networks and Services (K '11)*, pp. 1–8, December 2011.

[19] K. Kumar, J. Liu, Y.-H. Lu, and B. Bhargava, "A survey of computation offloading for mobile systems," *Mobile Networks and Applications*, vol. 18, no. 1, pp. 129–140, 2013.

[20] C. M. Sarathchandra Magurawalage, K. Yang, L. Hu, and J. Zhang, "Energy-efficient and network-aware offloading algorithm for mobile cloud computing," *Computer Networks*, vol. 74, pp. 22–33, 2014.

[21] J. R. Quinlan, *C4.5: Programs for Machine Learning*, Morgan Kaufmann Publishers Inc., San Mateo, Calif, USA, 1993.

[22] F. A. Silva, G. Zaicaner, E. Quesado, M. Dornelas, B. Silva, and P. Maciel, "Benchmark applications used in mobile cloud computing research: a systematic mapping study," *The Journal of Supercomputing*, vol. 72, no. 4, pp. 1431–1452, 2016.

[23] S. Tasnim, M. A. R. Chowdhury, K. Ahmed, N. Pissinou, and S. S. Iyengar, "Location aware code offloading on mobile cloud with QoS constraint," in *Proceedings of the IEEE 11th Consumer Communications and Networking Conference (CCNC '14)*, pp. 74–79, January 2014.

[24] A. Ravi and S. K. Peddoju, "Handoff Strategy for Improving Energy Efficiency and Cloud Service Availability for Mobile Devices," *Wireless Personal Communications*, vol. 81, no. 1, pp. 101–132, 2015.

[25] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. de Rose, and R. Buyya, "CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Software: Practice and Experience*, vol. 41, no. 1, pp. 23–50, 2011.

[26] W. Junior, A. França, K. Dias, and J. N. de Souza, "Supporting mobility-aware computational offloading in mobile cloud environment," *Journal of Network and Computer Applications*, vol. 94, pp. 93–108, 2017.