Computer Communications 000 (2016) 1–11

Contents lists available at ScienceDirect

Computer Communications

journal homepage: www.elsevier.com/locate/comcom

Reducing your local footprint with anyrun computing

Alan Ferrari*, Silvia Giordano, Daniele Puccinelli

Institute for Information Systems and Networking, University of Applied Sciences of Southern Switzerland (SUPSI), Switzerland

ARTICLE INFO

Article history:
Received 2 July 2015
Revised 14 December 2015
Accepted 22 January 2016
Available online xxx

Keywords:
Code offloading
Computation offloading
Opportunistic computing
Bayesian network
Software profiling

ABSTRACT

Computational offloading is the standard approach to running computationally intensive tasks on resource-limited smart devices, while reducing the *local footprint*, *i.e.*, the local resource consumption. The natural candidate for computational offloading is the cloud, but recent results point out the hidden costs of cloud reliance in terms of latency and energy. Strategies that rely on local computing power have been proposed that enable fine-grained energy-aware code offloading from a mobile device to a nearby piece of infrastructure. Even state-of-the-art cloud-free solutions are centralized and suffer from a lack of flexibility, because computational offloading is tied to the presence of a specific piece of computing infrastructure. We propose AnyRun Computing (ARC), a system to dynamically select the most adequate piece of local computing infrastructure. With ARC, code can run anywhere and be offloaded not only to nearby dedicated devices, as in existing approaches, but also to peer devices. We present a detailed system description and a thorough evaluation of ARC under a wide variety of conditions. We show that ARC matches the performance of the state-of-the-art solution (MAUI), in reducing the local footprint with stationary network topology conditions and outperforms it by up to one order of magnitude under more realistic topological conditions.

1. Introduction

Though mobile smart devices are becoming increasingly powerful, resource-intensive tasks are still well beyond their reach. Smartphones and tablets are still no match for complex tasks such as pattern matching or face recognition algorithms. Computational offloading is the standard approach to reduce the *local footprint*, *i.e.*, the local resource consumption of smart devices. With computational offloading, a set of programmatic instructions or even an entire program are run onto a remote device. The cloud appears to be the natural place to offload to, as it offers virtually unlimited resources and computing power. Internet connectivity is becoming truly ubiquitous, and at the same time the Internet is becoming increasingly cloud-centric. It is far more cost-effective to outsource resource-intensive tasks to a powerful, dedicated, high-performance computing infrastructure than to run them locally. Nevertheless, in spite of its undeniable benefits, cloud access is still highly inefficient for the offloading device in terms of latency and energy consumption, as shown by recent research that quantifies the local footprint of cloud access [1]. Though large corporations with a stake in cloud computing label it as green, cloud computing is far from green when one accounts for the cost of the information transfer between the cloud and client devices [2]. Cloud computing does save computing energy, but such savings are generally offset by the energy cost of offloading experienced by end devices. This is critical if the end device that interacts with the cloud is a mobile smart device, where battery lifetime and CPU usage are key concern.

In recent years, several approaches to (more or less) cloud-free offloading [1,3,4] have been proposed, and their details are provided in Section 2. Though these cloud-free offloading schemes are extremely valuable, they are not very flexible because they all require dedicated computing resources to run the offloaded code. Because using a remote machine over a WAN results in increased latency and suboptimal energy consumption, as shown in [1], for best results the dedicated computing hardware should be within the same LAN as the devices that need to offload their code. Thus, state-of-the-art cloud-free offloading generally lacks flexibility because dedicated resources need to be known a priori, and the offloading device is bound to share a LAN link with such resources. Using an analogy with the unicast addressing methodology, we refer to this flavor of computational offloading as *unirun computing*.

The goal of any offloading device is to minimize its local footprint, or at least reduce it compared to local execution. For the sake of flexibility, rather than having to offload to a specific high-end resource and being confined to a specific LAN, it would be

* Corresponding author. Tel.: +41 58 666 65 83.
E-mail addresses: alan.ferrari@supsi.ch, alan.ferrari@gmail.com (A. Ferrari), silvia.giordano@supsi.ch (S. Giordano), daniele.puccinelli@supsi.ch (D. Puccinelli).

http://dx.doi.org/10.1016/j.comcom.2016.01.006
0140-3664/

preferable for a smart device to be able to offload to another smart device with enough resources.

To overcome the limitations of state-of-the-art computational offloading strategies and dynamically leverage any suitable device, in this paper we propose (ARC), a system for the reduction of the local footprint of a mobile smart device. The terminology *anyrun computing* is chosen to draw an analogy with anycast addressing.

With ARC, code is offloaded whenever possible not only to dedicated (fixed) devices, but also to peer devices. The ability and flexibility to offload to the best resources available is of paramount importance in a world of heterogeneous devices with varying levels of computing power and resources.

With ARC, any encountered peer can play the role of the offloading device and offloading decisions are made based on Bayesian statistics, whose simplicity makes them particularly suitable to relatively limited resources of mobile smart devices.

In this paper we offer the following research contributions:

- a detailed system description of ARC;
- a thorough evaluation of ARC in a wide variety of conditions on a custom testbed;
- a comprehensive overview of the benefits of anyrun computing compared to unirun computing.

This paradigm drastically differs to standard device-cloud(let) architecture (e.g. the one we found in MAUI [1]) because it leverages any possible devices thus it breaks down the physical link to cloud(let).

As further motivation for ARC, we offer two examples of use cases where the benefits of ARC are made clear.

Home gaming. Though higher-end devices are generally available to smartphone users while at home, smartphones enable gaming experiences that cannot be achieved on desktop machines, laptops, or even tablets thanks to the availability of sensing devices and touch displays. While modern smartphones are equipped with HD screens and GPU to augment the visualization and the 3D rendering quality, their capabilities are much more limited than laptops. As an example, Google's Nexus 5 uses the Adreno 330 graphic board, which is over ten times slower than the Mac Book Pro's HD Graphics 4000 graphics card. The performance gap is an inevitable byproduct of the different form factor and power draw requirements of smartphones. With ARC, GPU computations can be dynamically offloaded to any available higher-end machine; in a home gaming scenario, heavy computational activities (e.g. 3D rendering) can be offloaded to the smartphone's user higher-end devices to drastically improve the gaming experience.

Computation as a service. Just like wireless communication (WiFi) is offered as a service by many businesses (especially franchised chains), communication could also be offered as a service to dynamically augment the computing capabilities of smartphones. Offloading computationally intensive tasks from smartphones to locally available computing infrastructure would contribute to keeping customers on the premises for longer periods of time, contributing to higher sales volumes. With ARC, the offloading would be carried out dynamically to maximize the quality of experience of the user so that desktop- and laptop-grade computation can be accessed on smartphones without the energy and latency penalty of cloud access and with no need for prior knowledge of the high-end computing resources available. Moreover, ARC would also enable the provision of computation as a service by means of the resources of other users, which would be feasible within a specific community of subscribers. Much like users of ridesharing services (such as Uber [5]) can access transportation as a service using resources of other users, users of ARC could achieve something similar for computation as a service. (We view the existence of a specific community of subscribers as a prerequisite for the viability of this model to enable compensation schemes for users lending their own computing resources.)

2. Related work

The ubiquitous and pervasive computing vision is finally becoming a reality as portable devices continue to become more and more widespread and powerful [6,7]. The latest generations of portable smart devices are extremely resource-rich, but they cannot compete with higher-end computing devices when it comes to computationally intensive tasks [8]. Cloud computing is now viewed as a natural solution to overcoming the limitations of mobile devices. Computational offloading (*a.k.a.* code offloading) is a solution to augment the capabilities of mobile systems by migrating computation to more resourceful devices (such as cloud servers) [9]. With the uptake of mobile smart devices, computational offloading is no longer restricted to the cloud, but can also target resource-rich(er) devices.

Given that smart devices generate a huge amount of heterogeneous sensory data that require plenty of processing power, it has been suggested that a mobile phone sensing architecture should rely on the mobile computing cloud [10], so that a smartphone can outsource resource-intensive tasks to a remote high-performance computing system reachable over the Internet. On the one hand, the idea of remote execution [11] and cyber-foraging [12] are first-class citizens in the world of pervasive computing, and the mobile computing cloud appears to be the natural choice [10], because portable smart devices will always be relatively resource-constrained compared to their fixed counterparts. As an example, the CloneCloud system [13] leverages execution migration techniques to clone a smartphone's state to the cloud so that computationally-intensive applications are run on a virtual smartphone clone within the cloud before reintegrating the results from the cloud back into the actual smartphone. On the other hand, the high-performance computing resources that form the computing cloud are typically available at a remote location, and the energy footprint of the data transfer may be significant [14].

To address the inherent resource-poverty of mobile terminals along with the setbacks of relying on distant clouds, the *cloudlet* model [3] has been proposed by Satyanarayanan et al., who empirically show the limitations of WAN-based cloud solutions and propose a novel approach based on accessing high-resource devices located in close proximity. The ThinkAir framework from Kosta et al. [4] proposes a novel computational offloading architecture based on smartphone's virtualization in the cloud. The authors provide method level computational offloading. The offloading strategy is chosen based on a method's energy footprint and device status in terms of resource usage and network connectivity. The authors show that the offloading gain in terms of energy consumption is one order of magnitude greater compared to local execution.

The Mobile Assistance Using Infrastructure (MAUI) system [1] has been recently proposed by Cuervo et al. to enable the fine-grained energy-aware offload of code from a mobile device to a MAUI node, *i.e.*, a nearby piece of infrastructure connected to the mobile device by a high-performance WLAN link. MAUI aims to reduce the energy footprint of mobile devices by delegating code execution to remote devices; it dynamically selects the function to be offloaded depending on the expected transmission costs of the network and provides an easy way for the developers to use the framework in their code. mobile terminals can leverage cloudlets of nearby infrastructure that can be accessed over Wi-Fi. This is certainly a promising strategy, especially given the recent results on the advantages of augmenting 3G with Wi-Fi [15]. It is shown

Please cite this article as: A. Ferrari et al., Reducing your local footprint with anyrun computing, Computer Communications (2016), http://dx.doi.org/10.1016/j.comcom.2016.01.006

in [1] that the cost of 3G for computational offloads is prohibitive compared to Wi-Fi, and that the energy consumption of computational offloads grows almost linearly with the Round Trip Time (RTT): using a nearby server is much more beneficial and energy-efficient than using a distant cloud, which confirms the conclusions in [3].

All the solutions previously shown suffer from centralization. Even if the cloud is decentralized into different servers in different LANs they always need a dedicated infrastructure to support cloud operations. And if small clouds in the neighborhood are better than a distant big cloud, why not further break up the cloud [6]? By adopting opportunistic computing [16,17], pervasive devices can opportunistically tap on each other's resources and access each other's services, or even combine each other's resources. The combination of resources has already been studied in the context of the MobiUS architecture [18] and its better-together paradigm, which focuses on close proximity networking between pairs of devices.

Opportunistic computing represents a radical generalization of both the MobiUS and the cloudlet/MAUI approach: services can be combined across multiple nodes, are offered by any node, and can be offloaded to any node (and not just a special subset of infrastructure nodes). Opportunistic computing effectively offers a *distributed cloud*, *i.e.*, a computing system that harnesses the CPU, memory, energy, and sensing resources of multiple nodes of heterogeneous capabilities that collectively form a cloud that is distributed in both space and time. Each node in the distributed cloud corresponds to a pairwise encounter, and may therefore provide useful local context information, which is something that a distant computing cloud would not be able to give. Opportunistic computing also avoids the centralization of cloud computing, with significant benefits in terms of security and reliability.

Privacy preservation is inherently favored because data are saved in a distributed fashion [19,20] as opposed to a centralized one that would make things easier for an attacker [21] (several server attacks have occurred in recent years; for instance, hackers steeled 15 million T-Mobile customer data[1]). Its distributed nature, however, means that OC has a harder time with device authorization compared to centralized approaches, which may use a central entity to manages all credentials. In OC, eachdevice has to take care of its own authorization, which may result in a cumbersome process especially if the user intervention is required. There exist solutions that address this specific problem; some of them exploit the social relationship among device owners (e.g. [22]) while others exploit user activities in the forms of common gestures to detect the relationship among device owners (e.g. [23]).

3. System description

State-of-the-art offloading schemes rely on *unirun* computing, *i.e.* computational offloading toward a specific high-end resource. In this paper, we propose AnyRun Computing (ARC), a framework for computational offloading where any device that is willing and able to help can be offloaded to.

3.1. AnyRun computing

Rather than assuming the existence of a dedicated piece of higher-end hardware and being rigidly tied to it, ARC may elect to use any piece of hardware, *i.e.*, any other device that is locally available and can be accessed through a LAN and can do the job better than they can. ARC offloads opportunistically so that an offloading device can get help from any other device and not necessarily from the cloud or from a dedicated high-end resource.

The key benefit of ARC is its achievement of higher energy efficiency and reduced latency through the use of nearby resources, as it far more energy-efficient and time-efficient to offload to a resource-rich device in close proximity than to a distant cloud [1]. In this paper, computational offloading is considered advantageous compared to local execution if it reduces the local footprint compared to local execution in the key dimensions of **CPU and RAM usage**, **execution time**, and **energy consumption**. The local footprint of ARC can be extended to other dimensions as well. In this paper, it is assumed that any remote node that offers its resources is willing to run the offloaded code.

The goal of ARC is to dynamically decide whether any kth run ($k \in \mathbb{N}$) of a method m should be offloaded or run locally. To make this decision on resource-constrained devices, we employ a Bayesian approach, which is known to be very efficient and lightweight in terms of resource usage [24,25]. Specifically, we focus on the Naïve Bayes approach [26], which requires relatively little computing power and provides an excellent inference quality compared to more sophisticated approaches.

Note that ARC does not ensure global fairness because it would require a global view that could be maintained only with cloud-based solutions. Our approach draws inspiration from swarm-based algorithms where each device has a limited view (usually limited to its neighbours) and local optima are sought.

In the following subsections we will delve into the ARC architecture as well as the details of the inference model used.

3.2. ARC architecture

The system is built on top of the SCAMPI opportunistic computing framework [17,27] developed within the FP7 EC project SCAMPI (Service Platform for Socially Aware Mobile and Pervasive Computing) [28]. The key issue tackled in SCAMPI is the provision of services on top of the opportunistic computing paradigm; recent work in this space includes [29–31]). SCAMPI offers a set of API to recognize neighboring devices with a set of different technologies (from legacy WIFI to Bluetooth) and allows users to access services on such devices. Services may be run on multiple devices in parallel (multicast access) or on a single device (unicast). This paper extends SCAMPI with an inference layers targeted to computational offloading.

The state machine in Fig. 1 illustrates the behavior of the main components of ARC. The entry point of the ARC framework is the *method stub*, which is a predefined class that allows developers to subscribe to the framework and access it easily. The stub uses the *Inference Engine* to assess the probability that offloading is advantageous compared to local execution in the dimensions of interest (CPU and RAM usage, execution time, and energy consumption). Depending on the response from the Inference Engine, the stub takes the appropriate course of action.

In case local execution is chosen, a dedicated architectural block (the *Local Execution Manager* in Fig. 1) uses Java reflection[2] to call the method. In case remote execution is chosen, a dedicated architectural block (the *Remote Execution Manager* in Fig. 1) serializes the part of the Java heap reachable from the method, sends it to the remote machine through the SCAMPI framework , and awaits a response. A timeout is set based on the local execution time

[1] http://www.pcworld.com/article/2988247/privacy/breach-at-experian-may-have-exposed-data-on-15-million-consumers-linked-to-t-mobile.html .

[2] We use Java as programming language for the implementation. Though the use of Java is not a fundamental requirement, certain features of the Java language (*i.e.*, reflection and serialization) make it particularly suitable for the implementation of computation offloading schemes. A viable alternative would be the use of Microsoft's .NET Framework, employed in [1]. Further details about our Java implementation are provided in Section 4.

Please cite this article as: A. Ferrari et al., Reducing your local footprint with anyrun computing, Computer Communications (2016), http://dx.doi.org/10.1016/j.comcom.2016.01.006

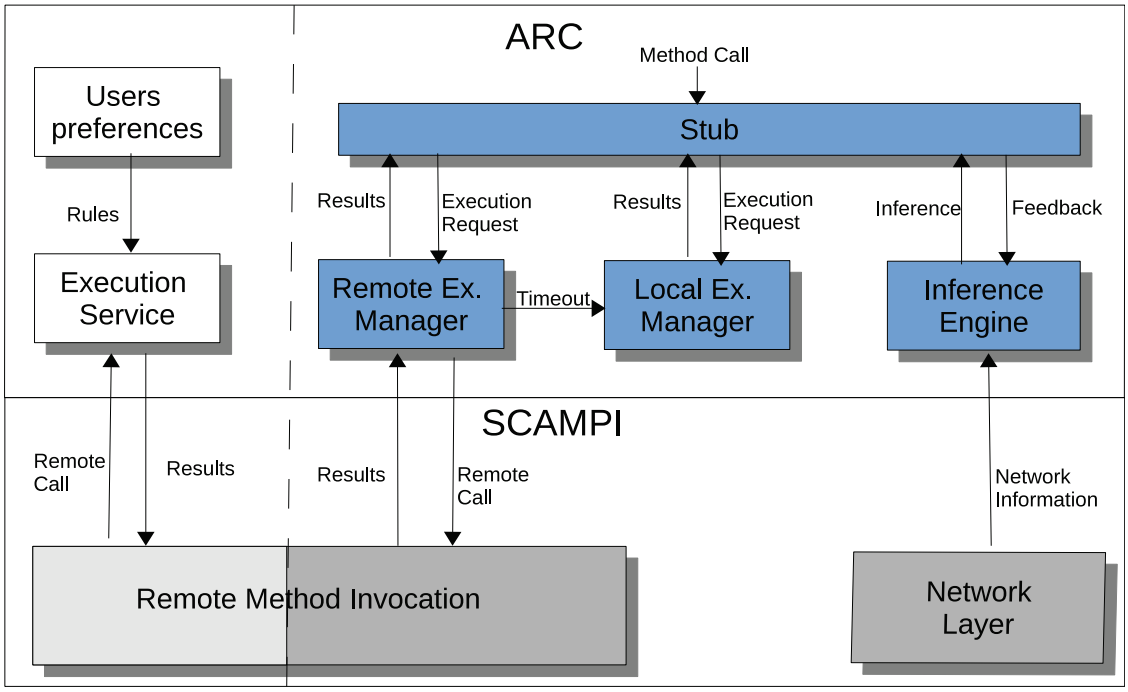

Fig. 1. The components of the ARC architecture and their interaction.

(we assume that each method is run locally at least once). If the remote execution exceeds the timeout, the Remote Execution Manager calls the Local Execution Manager so the method can be run locally. The SCAMPI framework offers a Remote Method Invocation (RMI) service that enables devices to request remote routine execution from each other. In ARC, the Remote Execution Manager uses SCAMPI's RMI jointly with an *Execution Service Manager* on the remote device. The Execution Service Manager is tasked with running the desired method on the remote device and provide the related results, also by way of SCAMPI's RMI.

We now delve into the details of the Inference Engine, which contains most of the intelligence in the system.

The Inference Engine receives a rich set of information about the available remote devices from SCAMPI (for more details, please refer to Section 4) and employs the information to help the stub decide whether offloading is advantageous compared to local execution, *i.e.*, whether it can reduce the local footprint compared to local execution in the dimensions of interest (CPU and RAM usage, execution time, and energy consumption). The output of the Inference Engine is an empirical estimate of the probability that offloading is advantageous in the aforementioned sense.

Let C represent a classification variable that defines mutually exclusive and collectively exhaustive classes, and let c be the value of C. A classifier is a function that assigns a class label to a set of attributes $E = \{E_1..E_{N-1}\}$ (with $N \in \mathbb{N}$). If the attributes E_i are assumed to be independent, then we have a Naïve Bayesian model [26], for which

$$P(E = e|C = c) = \prod_{i=1}^{N} P(E_i = e_i|C = c). \quad (1)$$

In our case, as the goal is to determine whether a method m should be offloaded, our classification variable C may take on either one of two mutually exclusive and collectively exhaustive values: *offload* or *run locally*. We focus on $C = offload$, and $P(C = offload|E = e)$ is to be interpreted as the probability that offloading is advantageous in the dimensions of interest given that the realization e of the set of attributes E has been observed. Since the two values that we assume C may take on are mutually exclusive and collectively exhaustive, $P(C = run\ locally|E) = 1 - P(C = offload|E)$ (Table 1).

Table 1
List of symbols.

Symbol	Description
m	Method marked for offloading.
m_k	Current run of a method.
L_{max}	Maximum size of the most recent executions.
L	Number of most recent runs considered.
Λ_m	Average estimate of the cost of running m.
$\Lambda_{m,k}$	Costs of the kth runs of method m.
E	Mmost up-to-date observation.
λ	Probability Threshold.
T_{max}	Maximum allowed latency.

By definition, the prior probability must be independent from E [25]. We elect to compute the prior probability based on the state of the local node, while we populate E with attributes of the remote node we wish to offload code to. The first step is the computation of the prior probability that offloading is advantageous in the dimensions of interest based on the state of the local node. For each kth run of method m, *either local or remote*, ARC estimates the *local* footprint as a function of the dimensions of interest (for more details, please refer to Section 4).

Let $l < k$ represents the current count of local runs; a moving average of the cost of the L most recent *local* runs is maintained by the offloading block and denoted as Λ_m, with $L = \min(l, L_{max})$. Each method m must be run locally at least once to initialize Λ_m, which serves as our estimate of the average cost of running m locally under the recent conditions at the local node. In principle, it suffices to run m locally just once to initialize Λ_m. It is also possible to initialize Λ_m based on estimates of its CPU, RAM, time, and energy footprint. However, in a realistic scenario with unstable connectivity, it is not always possible to offload m, so it is reasonable to expect that there will be a number of local runs and to leverage them to get an up-to-date estimate of Λ_m. Because the

Please cite this article as: A. Ferrari et al., Reducing your local footprint with anyrun computing, Computer Communications (2016), http://dx.doi.org/10.1016/j.comcom.2016.01.006

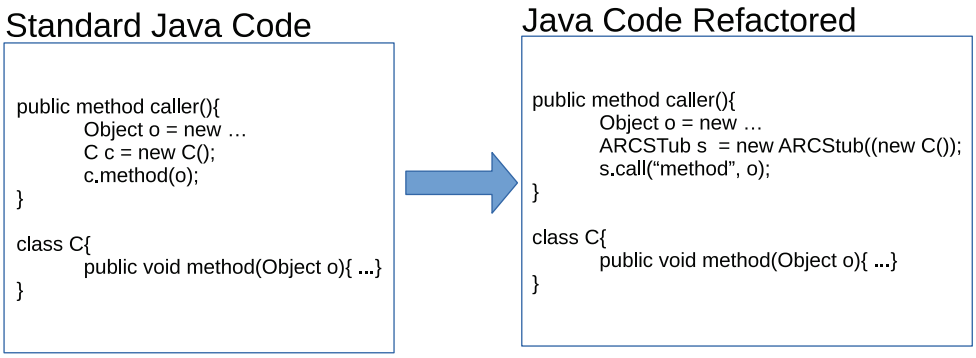

Fig. 2. ARC Java Refactoring where the method stub is included in the application control flow.

prior probability must be independent from the set of selected attributes [25], E is populated with remote node attributes. The set of attributes E is advertised by each remote device in its broadcast control messages (ARC presupposes the existence of a beacon-based neighbor discovery protocol). For the details of the attribute selection in our implementation, please refer to Section 4).

Whenever an offloading decision has to be made, ARC uses the most up-to-date observation of E, say $E_o \triangleq \{e_1 \dots e_N\}$, to compute the Bayesian inference, *i.e.*, the posterior probability that the observed set of attributes can be classified as c:

$$P(C = c|E = E_o) = P(C = c) \prod_{i=1}^{N} \frac{P(E_i = e_i|C = c)}{P(E_i = e_i)} \quad (2)$$

For $c = offload$, Eq. (2) represents the probability that offloading is advantageous under the conditions represented by the attribute observation E_o. To compute this probability, for each realization $e_i \in E_o$ of the random variable $E_i \in E$, we need to compute $P(E_i = e_i|C = c)$ and $P(E_i = e_i)$. $P(E_i = e_i|C = offload)$ represents the probability that the ith attribute within E takes on the value e_i given that offloading is convenient. This probability is estimated based on the history of the runs of a given method m as the fraction of the runs with $\Lambda_{m,k} \leq \Lambda_m$ for which $E_i = e_i$ is observed. Likewise, $P(E_i = e_i)$ is estimated as the fraction of the runs of m for which $E_i = e_i$ is observed.

Having gauged the probability that offloading is advantageous in the dimensions of interest, we set a threshold $\lambda \in [0, 1]$ that captures how conservative we wish to be with the use of local resources. For instance, if $\lambda = 1$, everything will be run locally, while if $\lambda = 0$, every method will be offloaded.

The specific value of the threshold λ must be set by the application whose methods are to be offloaded or run locally. For instance, if the application doesn't have particularly tight timing requirements, it is acceptable for the application to set λ to a lower value; conversely, the tighher the timing constraints of the application, the higher the value it ought to set λ to.

In case it is possible to offload to multiple devices, ARC sends offloading request in parallel to all available devices that are inferred as possible good candidates. The requests are sent to all the good candidates in descending order of probability of success, starting from the one that offers the highest probability of success. We plan to refine this greedy approach in our future work.

We also set a timeout T_{max} that captures the maximum latency value we can settle for.

The execution of method m is offloaded if the probability that offloading is advantageous exceeds or matches λ and is run locally otherwise. If the overall execution latency exceeds T_{max}, ARC defaults to local execution.

ARC continuously collects information about newly encountered devices that advertise their resources to assess whether they could be good offloading partners.

Because we are using heterogeneous devices, standard solutions employed with cloud-based offloading are not applicable. In our solution, we delegate the creation of a security policy to individual users (who may set it up as part of their *User Preferences*). While we do not address security aspects in this paper, a possible way for users to set an individual security policy is to use a set of predefined rules in the form of *key* → *value*. The key could be *block/allow* and the value a device id this allow uses to create a list of allowed/blocked devices. Or the rule could be *allow if* and the value is a boolean expression applied to system values (e.g. CPU, Battery, Memory) to allow offloading only when the resources are enough.

Some examples of rules:

- *block* device123
- *allow if* battery > 0.2

In the first case we block the device that has id *device123* and, in the second case, we allow offloading only if the battery is greater than 20%.

We do not address the use of cryptography and data obfuscation techniques in this paper. Such techniques may be used to increase security in case sensitive information is transmitted, however, their energy cost would be significant and care should be exercised.

4. Implementation details

Our ARC implementation is written in Java and works in both the Dalvik virtual machine [32] on Android-based[3] devices and the Oracle's HotSpot virtual machine [33] used in desktop environments. Portability to other Java-compatible virtual machine may be possible depending on the implementation of the Java memory model. We use the standard Java Object serialization to exchange class instances at runtime and Java reflection to dynamically choose the appropriate method at runtime. Object mobility is allowed inside the previously cited virtual machines because they share the same object representation in the memory space.

In ARC, similarly to MAUI [1], if a developer wishes to offload a portion of her application, she simply implement a ARC class stub that will call the offloadable code dynamically through the framework.

In ARC we implemented a Java *refactoring* technique (that could be easily automatized in the majority of Java IDE, e.g. Eclipse [34]) to allow developers to easily include the system in their application. Fig. 2 shows the standard refactoring procedure in ARC:

[3] Due to changes in the virtual machine Android version should be 4.0 (Ice Cream Sandwich) or newer. The most recent version we have tested is 5.1 (Lollipop).

- At first it replaces the initialization of the classes we want to offload with the ARCSTub class (let *C* be the class we want to use in offloading).
- It initializes the *ARCStub* class with *C* as its constructor argument.
- It replaces each method call in *C* with the call to the *ARCStub.call* method and it passes the method name and the parameters as arguments. If the user doesn't want a given method to be offloaded, the *"callLocal"* function can be called on *ARCStub* to force local execution.

We are currently implementing and automatizing this refactoring technique in the new Android IDE (Android Studio [35]).

A standard profiler is run by ARC upon the execution of each offloadable method (either if run locally or remotely).

The ARC profiler keeps track of several variables ranging from the execution time to the battery drain. In our implementation, ARC estimates the local footprint as a function of the dimensions of interest (execution time and energy). *E* is populated with remote node attributes and is advertised by each remote device in its broadcast control messages (ARC presupposes the existence of a beacon-based neighbor discovery protocol).

In our implementation, we employ the following attributes for any method *m*:

- the CPU and RAM usage and the battery usage on the remote device;
- the mean duration of the past contacts between the local device and the remote device (computed over a sliding window);
- the taxonomy of the remote device *m* is offloaded to (smartphone, tablet, laptop, or others);

Information about past contacts is used as an empirical estimate of the expected duration of the contact time between the local device and the remote device.

The information about the remote device is passed to the *Inference Engine* jointly with the information about the local execution of the method and the local status of the system (in terms of RAM usage and CPU usage) as input parameters. The inference process then returns an estimate of the probability that offloading to the remote device will be advantageous in the dimensions of interest. We use this inferred probability to make the offloading decision as explained in Section 3. In the remainder of this paper, we set T_{max} for each method *m* to coincide with the mean local run time; however, in general, T_{max} can be freely set by the application.

5. Performance evaluation

In this section, we evaluate the performance of ARC on real-world devices and emulate the variability of the network in our custom-made testbed.

5.1. Performance evaluation setup

In order to assess ARC's performance we employ the well-known MAUI framework for computational offloading (more details are offered in Section 2). As benchmarks, we employ Ant Colony Optimization (ACO), Face Recognition (FR), and Character Recognition (CR). ACO is an artificial intelligence technique based on the pheromone-laying, food-searching behavior of ants [36]. ACO has been applied to a wide range of combinatorial optimization problems and has also been used to find near-optimal solutions to the Traveling Salesman Problem (TSP). In the performance assessment, we use ACO applied to a TSP problem with 52 cities (Table 2). For FR, We use the eigenfaces approach proposed by Turk et al. [37]. For CR, we use a pre trained Kohonen Neural Network [38] trained with 32 pre-parsed sets of printed characters and then use the neural network to identify 1000 randomly chosen characters within the aforementioned sets. Following traditional approaches used to run controlled tests in wireless networks [39–41] to emulate a realistic, dynamic network topology, we employ DroidLab [42], a custom, scalable Android Emulation Testbed that enables protocol testing with real hardware under realistic conditions with connectivity patterns dictated by contact traces. In DroidLab, contact patterns are managed in a central server that periodically forwards the new pattern to the connected devices.

Table 2
Emulation variables, each possible combination has been tested.

Parameter	Values
Solutions compared	ARC, MAUI
Benchmark problem	Ant Colony Optimization, Face Recognition, and Charater Recognition
Network topologies	Local, Fully Connected, Dynamically
Number of devices	Single device, and Multiple Devices

The central server uses *iptables* [43], a user-space application that enables the configuration of the Linux Kernel firewall (usually implemented in the kernel as a Netfilter module). *Iptables* is currently offered in the majority of GNU/Linux distributions, including Google's Android. We modify DroidLab to forward connectivity patterns through the WiFi channel; this change is necessary in order to measure the energy consumption correctly.

We test ARC and MAUI[4] in four main scenarios. Two of them only use a single high-end device (the MacBook Pro) to carry out a fair comparison with MAUI, which is designed to work with a single piece of hardware located nearby. We do not simulate or emulate workloads on any of our test devices to isolate out the performance of MAUI and ARC.

The scenarios are:

- local execution: all benchmarks are run on the Nexus smartphone to acquire a performance baseline;
- fixed network topology with a single high-end device (labeled as **Full**): all benchmarks are run with full connectivity;
- dynamic network topology with a single high-end-device (labeled as **Dynamic**): all benchmark are run with an emulated dynamic network topology;
- dynamic network topology with multiple and heterogeneous devices (labeled as **Dynamic with multiple devices**): same as above, but with an augmented testbed.

The centerpiece of our custom testbed is the Google Galaxy Nexus smartphone [44] equipped with a 1.2 GHz ARM Cortex CPU, 1 GB of RAM, running Android 4.2 Jelly Bean. Our testbed also features an Apple MacBook Pro laptop with an Intel iCore 7 CPU, 4 GB of RAM, running Mac OS X Lion (10.7.5). The augmented testbed also includes two extra Galaxy Nexus smartphones and a Samsung Galaxy Tablet 2 equipped with 1GHz dual core Processor, 1GB of RAM, running Android 4.0 (Ice Cream Sandwich). We collect data regarding the various runs of the benchmarks and store it on the smartphone's SD card; runtime measurements are based on the internal clock. The dynamic scenarios are based on the Cambridge iMote data traces by Scott et al. [45] available on the Crawdad [46] archive offered by Dartmouth college. The iMote traces contain the data collected from 70 users during INFOCOM 2006, where users were asked to carry a set of Bluetooth-based devices that periodically scan the user's neighbors. We subsample the dataset choosing 5 devices and reduce the sampling ratio (from 5 to 2 min) to generate a more dynamic topology.

[4] We employ our own Android implementation of MAUI, available at http://code.google.com/p/maui-android.

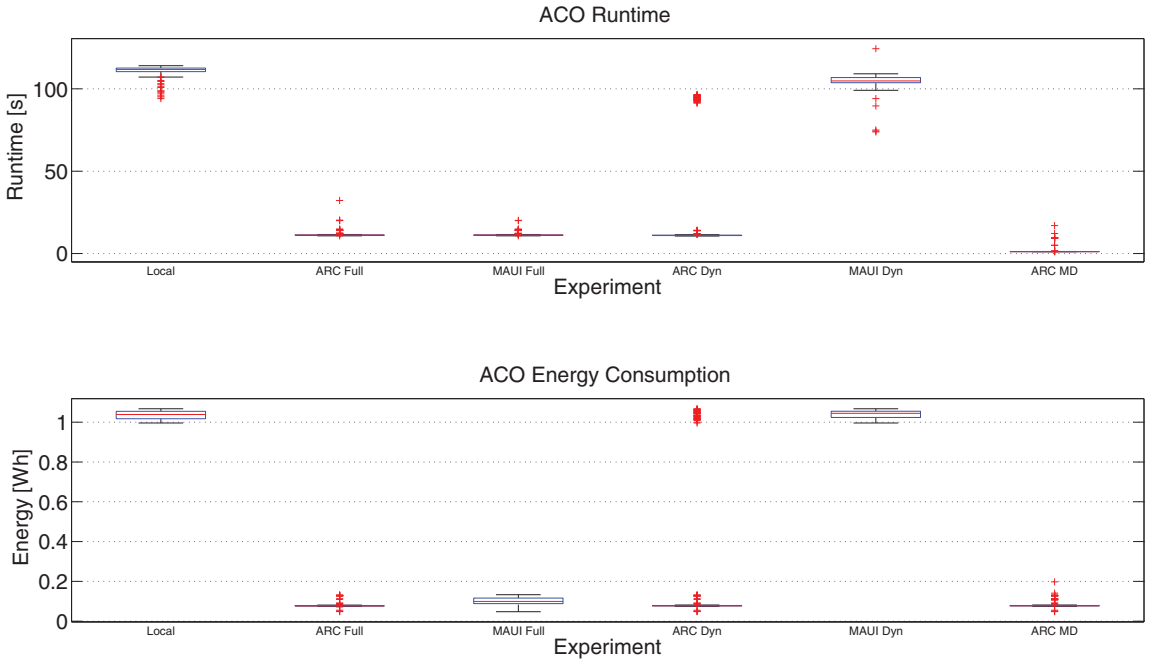

Fig. 3. ACO energy consumption and runtime. MD stands for **Multiple Devices**.

Table 3
ACO Results Table (MD stands for Multiple Devices and ED stands for Energy Drain)

Exp.	Scenario	Mean time [s]	Median time [s]	Mean ED [Wh]	Median ED [Wh]
Local		110.2	111.8	1.04	1.04
ARC	Full	11.4	11.2	0.08	0.08
	Dynamic	20.0	11.0	0.18	0.08
	Dyn. MD	19.7	11.5	0.08	0.08
MAUI	Full	11.3	11.1	0.10	0.10
	Dynamic	103.5	104.8	1.04	1.04

To collect the power consumption we use *POEM* (Portable Open Source Power Monitor [47]), a novel solution for energy profiling that allows developers to automatically test and measure the energy consumption of every single application component down to the control flow level. Because the power consumption is tied to the system status, energy annotation is also coupled with system activities. In the remainder of this section, we show the results of our experiments consisting of a set of benchmark runs totaling 75 h. of runtime.

5.2. ACO benchmark

For each scenario we run both ARC and MAUI over a complete battery cycle (ranging from 4 up to 7 h). Fig. 3 illustrates the distribution of the energy consumption and runtime of a single method and highlights the mean as well as the 95% confidence intervals, while Table 3 offers a comparison of the mean and median. With a fully connected topology, there is only a small set of differences that are probably due to system or network vagaries, as observed in [48]. Both mean and median values are comparable and the distribution of the runtime values and the energy consumption values are therefore very narrow. Key differences between MAUI and ARC emerge in the dynamic connectivity scenario, where ARC's ability to better manage challenging topologies becomes clear. In the presence of dynamic connectivity, ARC outperforms MAUI in both dimensions of interest by up to a factor of 10. With dynamic connectivity, MAUI adapts its offloading strategy based on network latency. In our emulated network, latencies are subject to wide scale changes ranging from milliseconds to minutes. MAUI maintains a moving average of the network latency, sampling it each time offloading is performed and at regular intervals (every minute) if there is no offloading. MAUI uses past method execution statistics to make offloading decisions; if it decides to offload and the network connectivity breaks down, MAUI eventually reverts on local execution upon reaching a timeout determined on the basis of past method execution statistics; such statistics are updated to reflect the penalty of having tried to offload in unfavorable circumstances, and even if network connectivity is restored, decisions remain strongly biased toward local execution, thus resulting in extended runtimes and increased energy consumption.

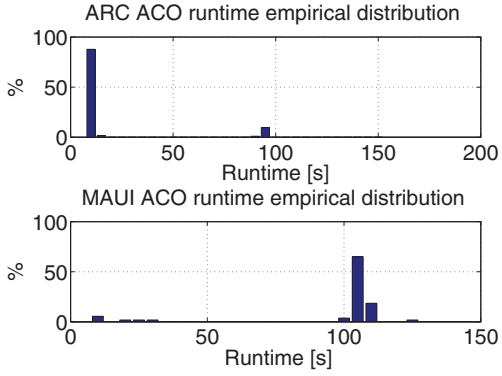

Fig. 4. ACO Dynamic: runtime distribution with ARC and MAUI.

This leads to the behavior illustrated in Fig. 4, which shows the distribution of the runtime for both ARC and MAUI with dynamic

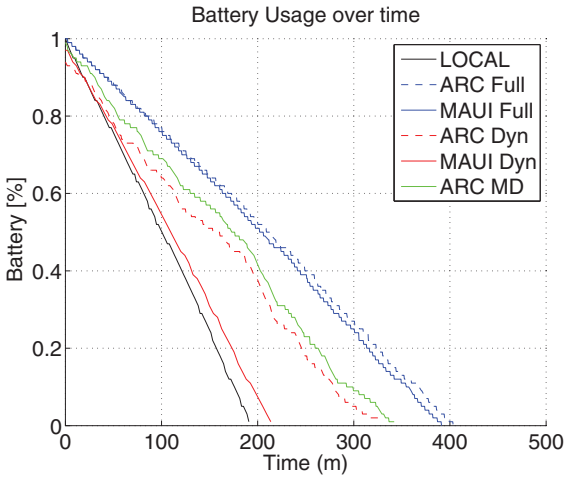

Fig. 5. Battery usage (as a proxy for energy consumption) over time in a sample ARC run.

connectivity. With ARC, we observe a bimodal distribution reflecting its ability to quickly adapt to changing connectivity conditions: local runs (responsible for the higher mode) are only performed when no other options are available due to the lack of connectivity, while offloading to the laptop is performed whenever possible. On the other hand, once MAUI reverts to local execution, it sticks with it independently of the connectivity conditions.

When ARC is run over multiple devices (3), the Galaxy Nexus 3 smartphones lead to timeouts (as the CPU load and the battery level of all our Galaxy Nexus 3 smartphones is roughly the same, the actual runtime is of course roughly equal to the local runtime, but it is augmented by the round-trip time for communication). Therefore, ARC correctly chooses to leverage the behavior of the higher-end devices in the testbed, namely the MacBook Pro laptop and the Samsung Galaxy Tab 2 tablet. (In practice, ARC might choose to offload to smartphones identical to the initiator in case to extend its lifetime.) When both the laptop and the tablet are reachable, the laptop is preferred; the tablet is picked only when the laptop is unavailable. This leads to the behavior illustrated in Fig. 3: ARC with multiple devices behaves similarly to ARC with dynamic connectivity; the extra data points that lie roughly in the middle of the distribution range and are only present in the **Multiple Device** results correspond to jobs offloaded to the tablet.

Fig. 5 illustrates the battery drain over a sample ARC run that covers a whole battery cycle. ARC and MAUI behave similarly with fixed connectivity (offloading results in a 110% extension of the battery lifetime compared to local execution). With dynamic conditions, ARC's battery lifetime gain is much more significant than MAUI's, amounting a battery lifetime extension of about 62% compared to MAUI and as much 71% with multiple devices; in such conditions, MAUI only achieves a modest 13% gain compared to local execution.

Fig. 6 shows a sample ARC run with dynamic connectivity and illustrates (Boolean) connectivity (in red) and the moving average of ARC's runtime (computed over a 2 min time window that matches the contact duration if the trace fed to the testbed for this run). We see that ARC takes full advantage of all the available connectivity opportunities.

5.3. Character recognition and face recognition

Fig. 7 shows the results for the character recognition benchmark in the dimensions of interest. Similarly to the ACO benchmark, both ARC and MAUI result in a significant offloading gain with fixed connectivity and ARC outperforms MAUI with dynamic connectivity. The results with multiple devices are also fairly close to the ones from the ACO benchmark. Fig. 8 shows the results for the face recognition benchmark in the dimensions of interest. Due to the nature of this benchmark, the results have a significantly wider statistical distribution. Each images is loaded and parsed, resulting in a large number of I/O operations that affect virtual memory management, resulting in numerous calls to the Garbage

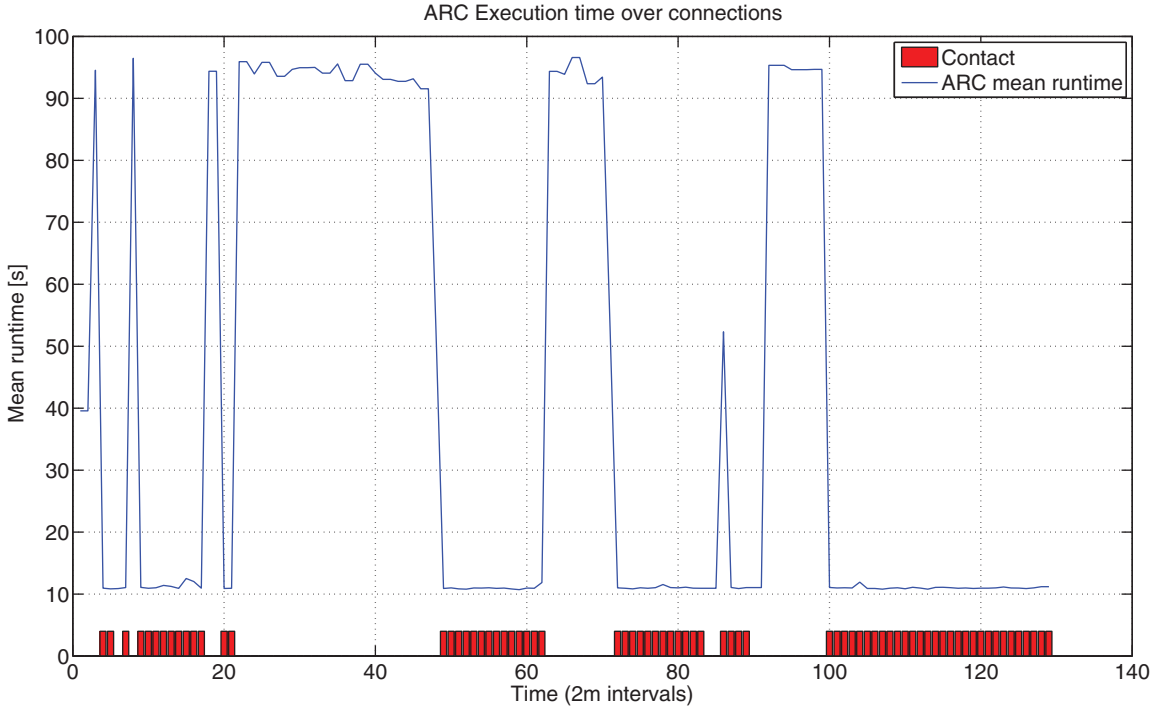

Fig. 6. Mean runtime as connectivity conditions fluctuate in a sample ARC run.

Please cite this article as: A. Ferrari et al., Reducing your local footprint with anyrun computing, Computer Communications (2016), http://dx.doi.org/10.1016/j.comcom.2016.01.006

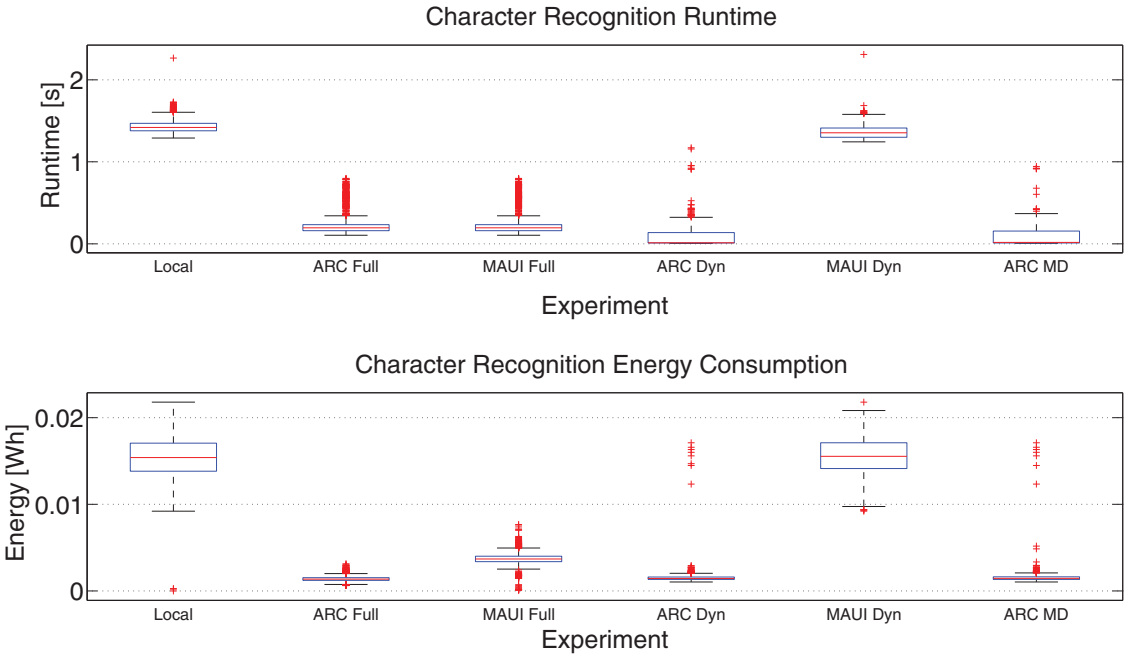

Fig. 7. Character recognition: energy consumption and runtime. MD stands for **Multiple Devices**.

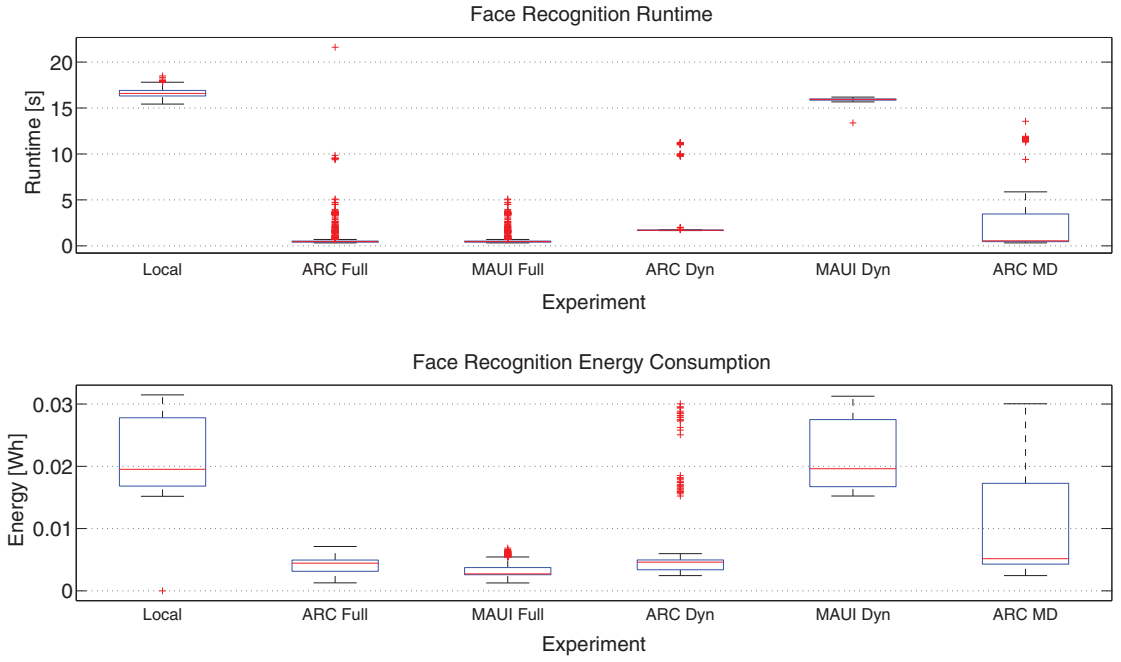

Fig. 8. Face recognition: energy consumption and runtime. MD stands for **Multiple Devices**.

Collector that create extra work, affecting both the runtime and the battery usage.

The key insight from these two benchmarks is that ARC is able to adapt to a different kind of algorithm that has different spatial and temporal needs in terms of computation. In fact, FR has higher communication costs compared to ACO but a comparable computational cost. In CR, instead, communication costs as well as computational costs are lower compared to FR.

6. Discussion and future work

Overall, ARC is a promising approach to cloud-free computational offloading. It matches the performance of the state-of-the-art under stationary conditions and improves it dramatically under the dynamic conditions emulated by our DroidLab testbed. We have observed several outlier values most likely due to system activity [48]. The key reason for ARC's significant performance gain in dynamic topologies is its flexibility: with ARC, a device can be ready to use any computing resource that can do the job more efficiently than it can do locally on its own. With the instability of real-world connectivity, being tied to a predetermined computing resource is not a sound strategy: when the nodes move, WLAN links become unstable to the detriment of offloading efforts. Conversely, by leveraging any offloading opportunity, ARC is able to cut both energy consumption and runtime. In spite of its merit, ARC has some important limitations, which are discussed in the

Please cite this article as: A. Ferrari et al., Reducing your local footprint with anyrun computing, Computer Communications (2016), http://dx.doi.org/10.1016/j.comcom.2016.01.006

remainder of this section. Some of them are prime candidates for future work.

Offloading to any device that is deemed suitable by the Inference Engine is a selfish approach to the minimization of the local footprint that completely ignores the impact of offloading on remote devices and may cause multiple remote devices to waste resources while running the same job concurrently for the benefit of our local device. The impact on remote user devices is only partially addressed in this work, where we assume that each user may protect her resources by enabling her own security policies depending on her specific needs. More sophisticated solutions may be adopted, but they would require learning the unique usage patterns of individual devices without the user's supervision. In this paper we also do not address a global cost optimization and therefore do not address fairness due to their intrinsic costs in terms of energy and latency. In fact, a global optimization would require a cloud-based approach with a global view of the activities of all the devices involved.

A further limitation is that this paper only considers offloading to devices within radio range, but there is nothing fundamental in ARC that precludes the possibility of operating over multiple hops in an opportunistic networking environment without end-to-end connectivity. The performance implications, especially in the dimension of latency, warrant a dedicated investigation.

One aspect of our work that it may seem natural to critique is the use of the Naïve Bayesian inference framework with its key assumption that the attributes are conditionally independent of one another given that the classification variable takes on a given value. Indeed, there is a significant body of literature (see, for instance, [26]) devoted to understanding why Naïve Bayesian models perform as well as they do in spite of the cavalier nature of their core assumption. Our choice of using a Naïve Bayesian model is dictated by its low complexity and lightweight computational footprint, and our results confirm that the model leads to sensible results in spite of its namesake naïve core assumption. We chose to run all of our code offload decisions on the same smartphone for the sake of repeatability, which means we did not study how different phones respond to ARC; this may be the subject of a future investigation.

Another key open point is the decision-making threshold λ, which is set to an arbitrary 0.5 in this study. The rationale is that we only decide to offload if ARC assesses at least a 50% chance that offloading will help. Ideally, λ should be calibrated dynamically by ARC based on its performance self-assessment. The choice of the cost function used to assess the local footprint is also a delicate point that remains open. Ideally, different application classes may employ different cost functions.

7. Conclusions

We have presented ARC, a novel framework for *anyrun* computing, whose objective is to decide whether computational offloading to any resource-rich device willing to lend assistance is advantageous compared to local execution with respect to a rich array of performance dimensions. As changing user trends dictate an ever increasing need for computational offloading and the energy and delay footprint of cloud usage becomes well understood, we believe that anyrun offloading gets more and more attractive. While the state of the art offers solutions that presuppose the deterministic existence of higher-end computing resources, we propose a novel, flexible approach inspired by recent work in opportunistic computing whereby offloading choices are made dynamically, opportunistically, and with complete awareness of the costs and benefits involved. In this paper, we have provided a comprehensive description of our approach and we have illustrated its performance evaluation based on a custom hardware testbed for trace-based mobility emulation.

Our comprehensive experimental results show that ARC proves to be extremely effective compared to MAUI [1], the state-of-the-art scheme for *unirun* cloud-free computational offloading. ARC virtually matches the performance of MAUI under stationary conditions and improves it by 50–60% under dynamic conditions, thus proving the potential of *anyrun* computational offloading.

Acknowledgments

This work was supported by the EU FP7 ERANET program under grant CHIST-ERA-2012 MACACO.

References

[1] E. Cuervo, A. Balasubramanian, D. Cho, A. Wolman, S. Saroiu, R. Chandra, P. Bahl, MAUI: Making smartphones last longer with code offload, in: MobiSys'10, San Francisco, CA, USA, 2010.
[2] J. Baliga, R. Ayre, K. Hinton, R. Tucker, Green cloud computing: balancing energy in processing, storage, and transport, Proc. IEEE 99 (1) (2011) 149–167.
[3] M. Satyanarayanan, P. Bahl, R. Caceres, N. Davies, The case for VM-based cloudlets in mobile computing, IEEE Pervasive Comput. 8 (4) (2004) 14–23.
[4] S. Kosta, A. Aucinas, P. Hui, R. Mortier, X. Zhang, Thinkair: dynamic resource allocation and parallel execution in the cloud for mobile code offloading, in: INFOCOM, 2012 Proceedings IEEE, 2012, pp. 945–953, doi:10.1109/INFCOM.2012.6195845.
[5] Uber (www.uber.com), July 1, 2015.
[6] S. Giordano, D. Puccinelli, The human element as the key enabler of pervasiveness, in: Proceedings of the 10th IEEE IFIP Annual Mediterranean Ad Hoc Networking Workshop (Med-Hoc-Net 2011), Favignana Island, Italy, 2011.
[7] E. Borgia, The internet of things vision: key features, applications and open issues, Comput. Commun. 54 (2014) 1–31.
[8] A. Passarella, M. Kumar, M. Conti, E. Borgia, Minimum-delay service provisioning in opportunistic networks, IEEE Trans. Parallel Distrib. Syst. 22 (8) (2011) 1267–1275.
[9] K. Kumar, J. Liu, Y.-H. Lu, B. Bhargava, A survey of computation offloading for mobile systems, Mob. Netw. Appl. 18 (1) (2013) 129–140.
[10] N. Lane, E. Miluzzo, H. Lu, D. Peebles, T. Choudury, A. Campbell, Mobile phone sensing: a disruptive technology for the app phone age, in: IEEE Communications Magazine, 2010.
[11] J. Flinn, D. Narayanan, M. Satyanarayanan, Self-tuned remote execution for pervasive computing, in: Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS VIII), Elmau/Oberbayern, Germany, 2001.
[12] M. Satyanarayanan, Pervasive computing: vision and challenges, IEEE Pers. Commun. Mag. 8 (4) (2002) 10–17.
[13] B. Chun, P. Maniatis., Augmented smart phone applications through clone cloud execution, in: Proceedings of the 12th Workshop on Hot Topics in Operating Systems (HotOS XII), Monte Veritá, Switzerland, 2009.
[14] A. Miettinen, J. Nurminen, Energy efficiency of mobile clients in cloud computing, in: Proceedings of the 2nd USENIX Workshop on Topics in Cloud Computing (HotCloud'10), Boston, MA, USA, 2010.
[15] A. Balasubramanian, R. Mahajan, A. Venkataramani, Augmenting Mobile 3G Using WiFi, in: 5th Annual International Conference on Mobile Systems, Applications and Services (MobiSys'10), San Francisco, CA, USA, 2010.
[16] M. Conti, M. Kumar, Opportunities in opportunistic computing, IEEE Comput. (2010) 42–50.
[17] M. Conti, S. Giordano, M. May, A. Passarella, From opportunistic networks to opportunistic computing, in: IEEE Communications Magazine, 2010.
[18] G. Shen, Y. Li, Y. Zhang, MobiUS: enable together-viewing video experience across two mobile devices, in: Proceedings of the 5th Annual International Conference on Mobile Systems, Applications and Services (MobiSys'07), San Juan, Puerto Rico, 2007.
[19] K.L. Huang, S.S. Kanhere, W. Hu, Preserving privacy in participatory sensing systems, Comput. Commun. 33 (11) (2010) 1266–1280.
[20] H. Li, Y. Yang, H. Yang, M. Wen, Achieving efficient and privacy-preserving multi-feature search for mobile sensing, Comput. Commun. (2015).
[21] I. Krontiris, T. Dimitriou, A platform for privacy protection of data requesters and data providers in mobile sensing, Comput. Commun. (2015).
[22] S. Trifunovic, F. Legendre, C. Anastasiades, Social trust in opportunistic networks, in: Proceedings of INFOCOM IEEE Conference on Computer Communications Workshops, 2010, IEEE, 2010, pp. 1–6.
[23] A. Ferrari, D. Puccinelli, S. Giordano, Gesture based soft authentication, in: proceedings of the 11th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob), 2015 IEEE, 37, 2015, pp. 311–324.
[24] F. Jensen, T. Nielsen, Bayesian Networks and Decision Graphs, Springer, 2007.
[25] D. Berry, Statistics: A Bayesian Perspective, Duxbury, 1996.
[26] H. Zhang, The optimality of naive bayes, in: Proceedings of the Seventeenth International Florida Artificial Intelligence Research Society Conference (FLAIRS 2004), 2004.

[27] M. Pitkänen, T. Kärkkäinen, J. Ott, M. Conti, A. Passarella, S. Giordano, D. Puccinelli, F. Legendre, S. Trifunovic, K. Hummel, M. May, N. Hedge, T. Spyropoulos, SCAMPI: service platform for soCial aware mobile and pervasive computing, in: Proceedings of ACM SIGCOMM Workshop on Mobile Cloud Computing (MCC), Helsinki, Finland, 2012.
[28] Scampi, service platform for social aware mobile and pervasive computing, July 1, 2015, (http://www.ict-scampi.eu/).
[29] E. Hyytiä, S. Bayhan, J. Ott, J. Kangasharju, On search and content availability in opportunistic networks, Comput. Commun. (2015).
[30] F. Li, Y. Rahulamathavan, M. Conti, M. Rajarajan, Robust access control framework for mobile cloud computing network, Comput. Commun. 68 (2015) 61–72.
[31] W. Liu, T. Nishio, R. Shinkuma, T. Takahashi, Adaptive resource discovery in mobile cloud computing, Comput. Commun. 50 (2014) 119–129.
[32] D. Ehringer, The Dalvik Virtual Machine Architecture, 2010. Technical Report http://show.docjava.com/posterous/file/2012/12/10222640-The_Dalvik_Virtual_Machine.pdf July 1, March 2015.
[33] Java SE Hot Spot at a Glance, 2014, (http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136373.html).
[34] The Eclipse Foundation, (http://www.eclipse.org), July 1, 2015.
[35] Android Studio, (https://developer.android.com/sdk/installing/studio.html), July 1, 2015.
[36] M. Dorigo, V. Maniezzo, A. Colorni, Ant system: optimization by a colony of cooperating agents, IEEE Trans. Syst., Man, Cybern., Part B: Cybern. 26 (1) (1996) 29–41.
[37] M. Turk, A. Pentland, Eigenfaces for recognition, J. Cognit. Neurosci. 3 (1) (1991) 71–86.
[38] T. Kohonen, The self-organizing map, Proc. IEEE 78 (9) (1990) 1464–1480.
[39] J. Beshay, K. Subramani, N. Mahabeleshwar, E. Nourbakhsh, B. McMillin, B. Banerjee, R. Prakash, Y. Du, P. Huang, T. Xi, et al., Wireless networking testbed and emulator (winetester), Comput. Commun. (2015).
[40] A. Dutta, P. Agrawal, S. Das, M. Elaoud, D. Famolari, S. Madhani, A. McAuley, B. Kim, P. Li, M. Tauil, et al., Realizing mobile wireless internet telephony and streaming multimedia testbed, Comput. Commun. 27 (8) (2004) 725–738.
[41] M. Carbone, L. Rizzo, An emulation tool for planetlab, Comput. Commun. 34 (16) (2011) 1980–1990.
[42] D.P. Alan Ferrari, S. Giordano, DroidLab: a Novel Android Based Testbed with Network Emulation, SUPSI, 2013 Technical Report (March 2013).
[43] Netfilter.org Project, (http://www.netfilter.org/l), July 1, 2015.
[44] Google Nexus Smartphones and Tables, 2013, (http://www.google.com/nexus), July 1, 2015.
[45] J. Scott, R. Gass, J. Crowcroft, P. Hui, C. Diot, A. Chaintreau, CRAWDAD trace cambridge/haggle/imote/cambridge (v. 2006-01-31), July 1, 2015, (http://crawdad.cs.dartmouth.edu/cambridge/haggle/imote/cambridge).
[46] Crawdad, a community resource for archiving wireless data at dartmouth, 2013, (http://crawdad.cs.dartmouth.edu/).
[47] A. Ferrari, D. Gallucci, D. Puccinelli, S. Giordano, Detecting energy leaks in Android app with POEM, in: 2015 IEEE International Conference on Pervasive Computing and Communication Workshops (PerCom Workshops), IEEE, 2015, March, pp. 421–426.
[48] T. Mytkowicz, A. Diwan, M. Hauswirth, P.F. Sweeney, Producing wrong data without doing anything obviously wrong!, SIGPLAN Not. 44 (3) (2009) 265–276, doi:10.1145/1508284.1508275.

Please cite this article as: A. Ferrari et al., Reducing your local footprint with anyrun computing, Computer Communications (2016), http://dx.doi.org/10.1016/j.comcom.2016.01.006