Large-scale Offloading in the Internet of Things

Huber Flores, Xiang Su,
Vassilis Kostakos
University of Oulu
firstname.lastname@oulu.fi

Aaron Yi Ding
Technical University of Munich
aaron.ding@tum.de

Petteri Nurmi, Sasu Tarkoma
University of Helsinki
firstname.lastname@helsinki.fi

Pan Hui
HKUST
panhui@cse.ust.hk

Yong Li
Tsinghua University
liyong07@tsinghua.edu.cn

Abstract—**Large-scale deployments of IoT devices are subject to energy and performance issues. Fortunately, offloading is a promising technique to enhance those aspects. However, several problems still remain open regarding cloud deployment and provisioning. In this paper, we address the problem of provisioning *offloading as a service* in large-scale IoT deployments. We design and develop an *AutoScaler*, an essential component for our offloading architecture to handle offloading workload. In addition, we also develop an offloading simulator to generate dynamic offloading workload of multiple devices. With this toolkit, we study the effect of task acceleration in different cloud servers and analyze the capacity of several cloud servers to handle multiple concurrent requests. We conduct multiple experiments in a real testbed to evaluate the system and present our experiences and lessons learned. From the results, we find that the *AutoScaler* component introduces a very small overhead of ≈150 milliseconds in the total response time of a request, which is a fair price to pay to empower the offloading architectures with multi-tenancy ability and dynamic horizontal scaling for IoT scenarios.**

Index Terms—**Edge Computing; Mobile Cloud Computing;**

I. Introduction

The Internet of Things (IoT) [1] is a computing paradigm that envisions a future where any low-power device, e.g., smartphone, smartwatch, sensors, etc., is connected to the Internet. The diversity of the devices, especially related to their computational power and battery, raises complex challenges on how to best utilize and run applications on the devices. *Offloading* is one of the key techniques for improving the resource usage (e.g., CPU, battery, storage) of low-powered devices. In offloading, a device outsources the processing of a task to a more powerful machine. To determine whether offloading is beneficial, the device needs to weigh during runtime the effort of executing an application locally and compare that to the potential benefits of offloading it. The cost of outsourcing the task is calculated by taking into consideration multiple parameters of the system [2], e.g., network latency, data transfer size, remote server capabilities, etc. The potential of the technique lies in the ability of aiding the low power devices with their processing [3].

While previous research has addressed offloading in mobile environments [4], currently the scaling of offloading into large-scale IoT scenarios is understudied. Indeed, existing IoT deployments are based on an ad-hoc approaches that cannot be easily scaled or migrated to new environments. Furthermore, there is a lack of standard platform that can be used for low-power devices (Arduino and Android being most popular) which hampers the development of offloading support into the IoT domain. Moreover, most offloading architecture rely on so-called neutral or heterogeneous execution models (see next section), which do not support flexible code execution both offline and online. IoT environments characterized by large heterogeneity among the devices and uncertain connectivity are better suited for *homogeneous* execution models that allow transparent migration of computations between client devices and cloud. Enabling support for homogeneous models, however, requires instrumentation of the client devices, which makes them difficult to scale compared with other architectures. Especially difficult is to ensure horizontal scaling, i.e., support for varying amounts of traffic from a large volume of client devices.

While offloading has been shown to be feasible on mobile technologies, there are still several fundamental challenges regarding cloud deployment and provisioning in real-world environments. The present paper addresses one key challenge in this process, the scaling of offloading support to large-scale IoT environments. Previous research has proposed designating one server per each device [5]. Such an approach is unrealistic and inefficient considering the potentially vast amount of devices that can operate in IoT environments. Thus, alternative solutions are required. Motivated by service-oriented architectures, and in particular the Amazon Autoscale, we design and develop an offloading architecture that integrates an *AutoScaler* mechanism. The AutoScaler operates as an interface between tasks arriving for offloading and computational resources available within the cloud by scheduling tasks amongs the different cloud computing resources. We also build an offloading simulator, which can emulate the offloading patterns of multiple devices (offloading workload). Since the adoption of Android for IoT devices is on the rise, we design and develop our system based on Android system. We show the feasibility of this approach by building a Dalvik-x86 surrogate and performing benchmark experiments through a real testbed in Amazon EC2, we study the capacity that an offloading architecture has to handle offloading workload in different scenarios. We find that our *AutoScaler* introduces a small overhead in the total response time of an offloading request, but this is negligible compared to the benefits of having an offloading architecture with multi-tenancy ability and dynamic horizontal scaling.

The rest of the paper is organized as follows. We introduce the background about offloading in Section II. We then present the design of our proposed offloading architecture, which integrates our proposed *AutoScaler* component in Section III. In Section IV, we evaluate the benefits obtained by leveraging the *AutoScaler* component in a real testbed. In the light of the results, we discuss the lessons learned and experiences in which we highlight new directions for the design of future offloading architectures supported by cloud computing in Section V and conclude the paper in Section VI.

II. Background and Related Work

In this section, we introduce the fundamentals of the offloading approach and cloud provisioning.

A. *Offloading*

Offloading is a technique that enables a low power device, e.g., smartphone, wearable, to outsource the processing of a task, e.g., code, service, job, etc., to a device with higher capabilities and resources [3], [5], [6]. A task is opportunistically outsourced from a device when in the presence of network connectivity the device can reach the server at low latency rates of transfer. However, offloading is not always beneficial if the computational requirements are small compared to communication costs (especially latency) and devices should only offload when the benefits of offloading are significant compared to cost of running the task on the device. The ultimate goal of the technique is to reduce the overall amount of processing of the device to extend battery life [7].

Multiple offloading models can be implemented in practice as shown in Figure 1. We classify the models into three groups. Firstly, (a) an *homogeneous model*, where the the Runtime Environment (RE) of the device and the server are the same and the code of the task is present in both locations. In this model, the device does not depend on the cloud to execute the task as it can execute the code when there is no network connectivity. Naturally, when the device is connected to the network and the context is suitable to offload [3], instead the mobile can rely on its server counterpart to execute the task. One key aspect of this model is the same RE in the device and server that is necessary to encapsulate the application state (AS) of the device, such that AS can be transferred in the network and reconstructed in the cloud in order to execute the task. Secondly, (b) an *heterogeneous model*, where both, the device and cloud have different RE. Therefore, both have a different implementation of the task. In this model, the device also is independent of a server counterpart. However, the device (typically) has a simpler implementation of the task compared to the server, i.e., the device can execute the task without network connectivity but result is not necessarily as good. Moreover, in this model, only the input parameters of the code are transferred in the communication. Thirdly, (c) a *neutral model*, where the RE is not relevant when outsourcing a task. The code of the task is uniquely located in the server, but it can be called from a device. This means that the device cannot provide independent functionality when there is no network connectivity. Our work uses a homogeneous model as it provides different code granulaties which aids to reduce data transfer, e.g., Method, Thread, Class, etc.

B. *Computational Provisioning*

Cloud computing is a computing paradigm where scalable resources are provided on demand *"as a service (aaS)"* over the Internet to users who need not have knowledge of, expertise in, or control over the cloud infrastructure that supports them [8]. Cloud provisioning can occur at different levels, where each level enables the user to interact with the service at a certain granularity. The most common levels are the infrastructure level (IaaS), platform level (PaaS) and the software level (SaaS).

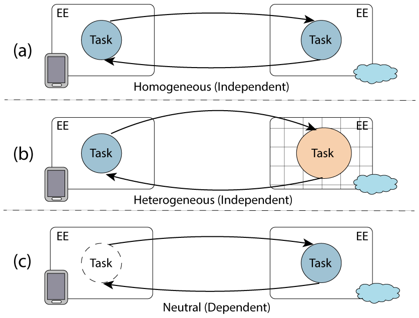


Fig. 1: Implementation models of offloading in practice.

Computational provisioning for smart devices takes place at infrastructure level through instances, which are physical or virtualized servers associated to a specific type of resource [4]. An instance follows an utility computing model, where consumers pay on the basis of their usage and type preferences. The cost of the instance is proportional to its type. The type of the instance is important as it determines the acceleration in which a task is executed, which affects the overall response time of the offloading request [9]. Moreover, the type of the instance also defines its capacity to handle multiple offloading requests at once. Speeding up code execution certainly depends on how code is written. However, higher types of instances can process a task faster than lower ones as higher types can rely on larger memory span and higher parallelization in multiple processors.

Horizontal scaling is essential for large-scale offloading scenarios as it enables cloud provisioning as a service to support multi tenacity [8], [10], [11]. Besides research that focuses on scaling up server to parallelize the code of computational requests (i.e, vertical scaling) [12], we have not found architectures that can scale horizontally to account for different traffic volumes and varying number of clients. This clearly can be seen as current frameworks do not take into consideration the utility computing features of the cloud, which is translated into server selection based on provisioning cost [13]. In this paper we further expand the traditional cloud offloading model to include an *AutoScaler* component that acts as a load balancer for different computational resources.

III. Large-scale offloading for IoT

Usually an offloading system consists of two parts, a mobile client and a server in the cloud. Such a model, however, is not well suited for large-scale offloading, such as those encountered in IoT environemnts, as it does not allow efficient utilization and distribution of resources on the cloud end. To address this challenge, in this section we present the design and development of a novel offloading architecture.

The key novelty of our work is the integration of a *Autoscaler* component, which as a gateway for offloading requests from the clients and as a load balancer for computing resources contributing to the cloud.

A. System overview

An overview of the system is presented in Figure 2a. The system consists of three parts, back-end, front-end, and load simulator. The back-end contains the servers that act as surrogates. Each surrogate can be dynamically added or removed (scale out). The front-end receives the incoming requests and distributes them among the available surrogates. The load simulator generates multiple offloading requests. As shown in Figure 2b, each component has the same runtime environment and the means for task reconstruction, which includes APK files and compiled dependencies as JAR files. We develop our system based on Android. The rest of the section describes each component of our system[1].

B. Implementation details

1) Back-end: Each surrogate of the back-end is a customized Dalvik-x86[2]. Dalvik is the virtual machine of Android to execute dalvik bytecode. Dalvik-x86 is built by downloading and compiling the source code of *A*ndroid *O*pen *S*ource *P*roject (AOSP) over the instance to target a x86 server architecture, and removing the *Applications* and *Application Framework* layers from the Android software architecture.

Our Dalvik-x86 is lighter when compared with other surrogates used by other works, e.g., Android-x86. We reduced the storage size required by our Dalvik-x86 surrogate in 40%. Moreover, it does not activate any default processes from the OS, e.g., Zygote, GUI Manager, etc, which are not needed by the surrogate. The surrogate creates a *dalvikvm* process in the host machine per each offloading request that needs to be handled. The advantage of this approach is that when an offloading request fails or gets stuck, it is possible to troubleshoot the request by process id without restarting or stopping the system.

Dalvik-x86 implements an executable script wrapper at the core of libraries that boot the compiler. The wrapper provides an interface to push APK files into the virtual machine, such that the code inside of the APK can be executed. When the server initiates, the available APK files (in a OS folder) are pushed into the Dalvik-x86 as a process waiting for a request. Each APK file can be instantiated multiple times and on different ports. Thus, when a request to execute code reaches the server, it is forwarded to any port that is listening.

When compared with the complete virtualization of the mobile platform, the main benefit of our light design is that the surrogate is released from the limitations imposed by the mobile operating system, such as activating multiple instances of the same application, or executing multiple applications concurrently. The surrogate is stored as an image in the cloud, and thus, the underlying resources can be dynamically allocated on demand to increase the throughput of the system. This means that the server can increase the resources to handle multiple Dalvik processes simultaneously (scale out) or to speed up the execution of code (scale up).

2) AutoScaler (Front-end): The *Autoscaler* implements a round robin scheduler to distribute the load among the available servers in the back-end. The *AutoScaler* contains a back-end descriptor in JSON format, which contains per each server, the information about where the *apk* files are located and the available ports for offloading. The descriptor (Listing 1) is updated when a server is added or removed. The descriptor is loaded during runtime using GSON. The *AutoScaler* is also able to allocate servers in the cloud via ec2-amitools of Amazon. For simplicity, the allocation of servers is triggered when the response time of the offloading workload surpasses a pre-defined threshold.

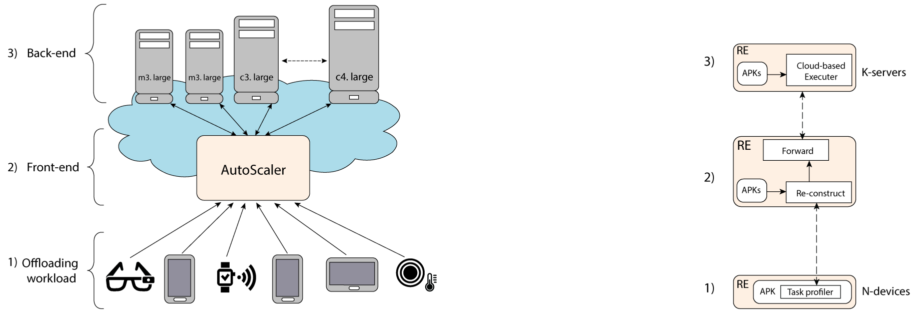
Listing 1 Example of back-end descriptor in JSON format

```
{
    "Chess_app": {
        "Surrogate: 54.73.45.xxx": {
            "ports": [
                "6001",
                "6002",
                ...
                "600N"
            ],
            "location": [
                "/home/ubuntu/android-x86/"
            ]
        }... others
    },
    "Nqueens_app": {
        "Surrogate: 172.16.32.xxx": {
            "ports": [
                "5001",
                ...
                "500N"
            ],
            "location": [
                "/home/ubuntu/dalvik-86/"
            ]
        }... others
    }
}
```

3) Simulator: To generate different loads of devices offloading to cloud, we have developed a simulator that creates workload in two different operational modes, 1) concurrent and 2) inter-arrival rate. The simulator is implemented using Java reflection, which allows it to capture the details of code invocation during runtime, e.g., name of the method, parameters, etc, and to define customized parameters in the request, e.g., type of acceleration. In the concurrent mode, based on an input parameter n, the simulator creates n concurrent threads. Each of the thread offloads a task loaded from a pool of available algorithms. Our simulator is equipped with a minimax algorithm, which can be used within a chess game application. The algorithm analyzes how the chess pieces are located in the board, such that it can enumerate all the possible moves and determine the best move to be executed. Each thread depicts a device offloading a task to a remote server. This mode is utilized to benchmark the instances of the cloud. In an inter-arrival rate mode, the simulator takes as parameter the number of devices (workload), the inter-arrival time between the last and the next request that reaches the system and the time that the workload is active or in other words, the time of the experiment. This mode is utilized to produce synthetic workload to the system.

[1] https://github.com/mobile-cloud-computing/ScalingMobileCodeOffloading

[2] Released as public in Ireland region of Amazon EC2 as *ami-ac8813df*



(a) (b)

Fig. 2: Large-scale offloading system, (a) System overview, (b) Component overview

IV. Evaluation and results

To evaluate our system, we deployed the server side and the *AutoScaler* in Amazon cloud (Ireland region / eu-west-1). As cloud servers, we selected general purpose instances (m1.small, m1.medium, m1.large, m3.medium, m3.large, m3.xlarge, m3.2xlarge) and an optimized memory instance (m2.4xlarge). We used these instances as they can be launched on demand indefinitely. Other higher capabilities instances, e.g., c3.8xlarge, require explicit request to the provider to be launched multiple times by a single account.

A. Performance

Setup and methodology: — Since our system introduces the *AutoScaler* component in the architecture. We explore first how this new component influences the response time of an offloading request. Thus, the goal of the experiment is to demonstrate the response time of an offloading request that uses our system.

Results, experiences, and lessons learned: — Timestamps are taken across the system as the request is processed in each of its components. Figure 3a models the response time $T_{response}$ of an offloading request. The response time consists of the time that takes to connect from the device to the front-end T_{m-f}, the time that takes to route from the front-end to a particular surrogate T_{f-b}, the execution time of the code in the surrogate T_{cloud}, the time that takes to send the result from the surrogate to the front-end T_{b-f}, and finally, the time that takes to send the result back from the front-end to the device T_{f-m}. We assume that $T_{m-f} = T_{f-m}$ and $T_{f-b} = T_{b-f}$ are equal as the same communication channel remains open both ways until the operation finishes. In this context, we define $T_1 = T_{m-f} + T_{f-m}$ and $T_2 = T_{f-b} + T_{b-f}$. Thus, the response time $T_{response} = T_1 + T_2 + T_{cloud}$.

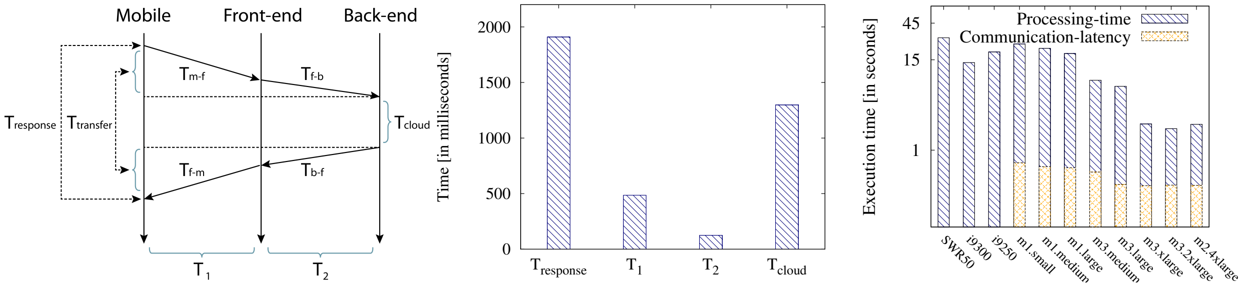
Figure 3b shows the timestamps taken across the system. We can observe that the extra time introduced by the front-end is ≈ 150 milliseconds and the total communication time $T_1 + T_2$ is less than a second. Naturally, higher latency from mobile to front-end can influence the communication time and vice-versa, which impacts T_1. T_2 is less likely to change drastically as the latency depicts the internal cloud communication, which is wired between servers in the same private network. Finally, the diagram shows that T_{cloud} is the dominant operation that impacts the total response time. From Figure 3b, we can observe that the total time of the code invocation in the cloud can be decreased by adjusting the tradeoff between utilization price and computational capabilities of the surrogate server. In specific, Figure 3c shows the execution of a minimax algorithm in different instance types. The input of the algorithm is fixed to a specific state of the chessboard. For comparison purposes, we also measure the execution effort of the same algorithm on multiple devices. The execution of the algorithm in other devices is ≈ 16 seconds for i9250 (Samsung Nexus), ≈ 14 for i9300 (Samsung Galaxy S3) and ≈ 29 seconds for SWR50 (Sony SmartWatch 3).

B. Scalability

Setup and methodology: — The capabilities of a system for handling multi-tenancy are crucial in an environment that follows an utility computing model. Previous work about offloading have proposed a one instance per mobile architecture [5], which is unrealistic in practice. The goal of this experiment is to verify the capabilities of our system to handle multiple active devices. As a result, we analyze the effect of handling multiple offloading requests in different instance types. We also measure the capacity of the *AutoScaler* to distribute requests among the surrogates in the back-end.

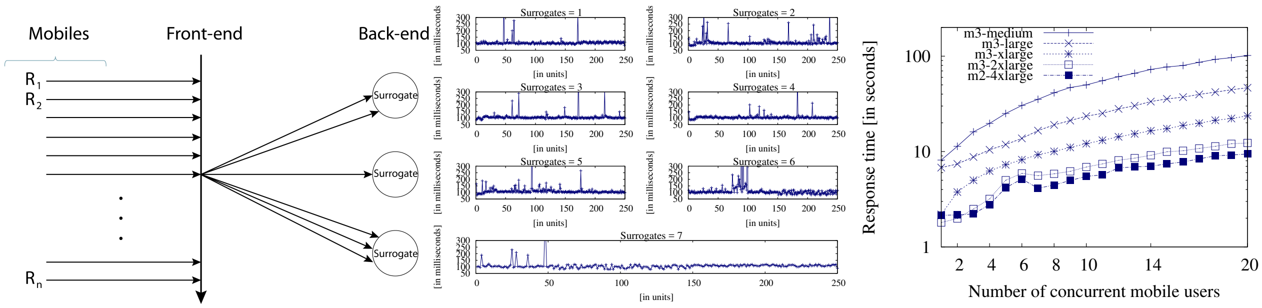
Figure 4a illustrates the experimental setup, where R_1 to R_n requests reach the *AutoScaler* in order to be distributed to the back-end based on instance type. To benchmark all the surrogate types, we utilize our offloading simulator described in Section III-B3. The simulator uses the minimax algorithm to create the load of requests. The input of the algorithm is fixed with the same specific state of the chessboard that yields the results shown in Figure 3c.

Results, experiences, and lessons learned: — The capacity of the *AutoScaler* is measured in terms of processing time of the request, which depicts the time taken by the *AutoScaler* to decide the surrogate to route the request. Since the *AutoScaler* receives and distributes the incoming load of requests, it is important to know the maximum capacity of the *AutoScaler* before it turns into a bottleneck for the system. Usually, the front-end is provided free of charge by the cloud vendor, e.g., Amazon autoscale. In our experimental setup, we assume that the *AutoScaler* is a m3.xlarge instance. Figure 4b shows the



(a) (b) (c)

Fig. 3: (a) Timestamps taken across the system in each of its components, (b) Actual times to handle the request in each component, (b) Acceleration of a task based on instance types.



(a) (b) (c)

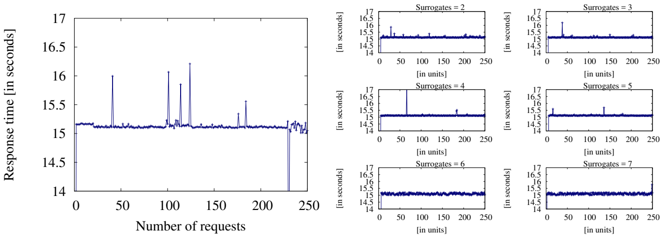
Fig. 4: (a) Load of incoming offloading requests, (b) Extra overhead introduced by the *AutoScaler* to route the requests among the multiple servers (1 to 7) in the back-end, (b) Number of active devices that are concurrently handled by an instance type.

number of requests that the *AutoScaler* can handle when facing a load of 250 requests with an interarrival rate of three seconds. The interarrival rate is the time between creating one and the next request. From the result, we can observe that the processing time to distribute requests remain in average ≈150 milliseconds even as the load increases.

On the other hand, unlike the *AutoScaler*, instances in the back-end face higher CPU utilization caused by the processing of the task. Offloading requests suffer from non-determinist execution of code, which means that the processing time of the request can vary abruptly based on different parameters, e.g., input variability in the code. Thus, the requests cannot be easily characterized. We conduct two different experiments to measure the capabilities of the server to respond to multiple requests. In the first experiment, we generate requests concurrently to a specific instance, and in the second, we generate requests based on an interarrival rate of three seconds. We also scale out gradually the servers in the second experiment to demonstrate horizontal scaling from 1 to 7, which means that servers are added on demand dynamically to the back-end. We used as surrogates m3.2xlarge instances, because they are the most powerful servers we can get without requesting explicitly to the cloud provider.

Results of the first experiment are shown in Figure 4c, which presents the number of concurrent requests that can be offloaded to a specific instance. From the results, we can observe that the response time of a request increases as the number of users/CPU utilization augment. As a result, each instance type has a maximum capacity to handle requests. For example, from the diagram, we notice that a m3.xlarge instance can handle up to six active requests while keeping the total response time of the request below the actual time required by the device to process the same operation. In principle, the instance should be able to handle more requests. However, the acceleration of tasks could fail in certain cases. For example, when a m3.xlarge instance handles around 20 active devices, the processing time of the request increases up to 17 seconds. Fortunately as shown in the figure, a higher capability instance like m2.4xlarge can easily cope with such capacity demand and response time requirements.

Results of the second experiment are shown in Figure 5. Interestingly, the response time of the requests drops up to 15 seconds and remains at that level even when adding more servers. Initially, we thought that the *AutoScaler* could become a bottleneck. However, after a close inspection (Referring to Figure 4b), we realize that the processing time of a request increases as a server receives the requests at different points in time. This means that unlike concurrent requests, the response time of those requests that reach the system in long interarrival are processed slowly by the server. This suggests that to achieve short response time when handling multiple requests, all the requests that reach the system must be scheduled by the *AutoScaler* to be processed at once. Offloading requests that contain code that can be parallelized can benefit from this characteristic, such that a request is parallelized within the same server instead of splitting it into multiple servers.



(a) 1 surrogate

(b) 2 to 7 surrogates

Fig. 5: Response time of the requests when scaling horizontally

V. Discussion

Based on the results of our experimental testbed, we present in this section the benefits, drawbacks and lessons learned for scaling an offloading architecture.

A. Offloading in practice

Many mobile games implement resource intensive tasks in smartphone devices, e.g., image processing. Our research relies on a minimax algorithm where the execution is in seconds. While it is arguable that our use case is not realistic in practice as a smartphone requires the execution of tasks in milliseconds, we can extrapolate several similar use cases to an IoT environment.

- Let us consider a microcontroller, e.g., Arduino, which analysis air samples to estimate pollution. Multiple samples can be taken to calculate an average pollution estimation, which requires long processing. Instead, the device could send the samples to the cloud, which will perform the processing faster.
- By relying on a near infrared sensor, devices can scan organic objects to detect its composition. Multiple applications are envision with this sensor embedded in a smartphone, e.g., healthy diet app. By offloading the data scanned by the sensor, the analysis time is improved.

As the IoT devices such as wearables are getting equipped with more sophisticated sensors, e.g., air pollution sensor, near infrared sensor, etc., we envision the need of processing tasks that need long execution time in order to ensure accuracy.

B. Optimal resource allocation

Offloading architectures that scale (as a service) require capacity planning policies to determine the right amount of back-end servers, which are required to handle a particular load of devices [14]. Moreover, since the acceleration of a task depends on the type of the instance, the optimization should be oriented to minimize provisioning costs while maximizing the capacity of the system to handle requests. Definitively, capacity is constrained by the response time of the request, which cannot be longer than the processing time in the low-power device. We leave the study of cost for our future work as it requires a deep analysis of cloud provisioning.

VI. Conclusions

Offloading is a key technique in augmenting the computational capabilities of IoT devices with elastic cloud resources. In this paper, we investigated the effect of large-scale offloading for IoT. We introduced a offloading architecture an *AutoScaler* and component that can efficiently distribute the load of requests among the available surrogates in the back-end. We also developed an offloading simulator that can facilitate the generation of different offloading workloads. We presented the results of the evaluation along with the experiences and lessons learned from developing and using the framework in a real deployment and testbed. Finally, we released our system in GitHub along with the Dalvik-x86 in Amazon EC2.

References

[1] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, "Internet of things: A survey on enabling technologies, protocols, and applications," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 4, pp. 2347–2376, 2015.

[2] H. Flores and S. Srirama, "Mobile code offloading: should it be a local decision or global inference?" in *Proceeding of the 11th annual international conference on Mobile systems, applications, and services (MobiSys 2013).* ACM, pp. 539–540.

[3] H. Flores, P. Hui, S. Tarkoma, Y. Li, S. Srirama, and R. Buyya, "Mobile code offloading: From concept to practice and beyond," *Communications, IEEE*, no. 4, 2015.

[4] H. Flores and S. Srirama, "Adaptive code offloading for mobile cloud applications: Exploiting fuzzy sets and evidence-based learning," in *Proceeding of the 4th ACM MobiSys workshop on Mobile cloud computing and services*, 2013, pp. 9–16.

[5] P. Bahl, R. Y. Han, L. E. Li, and M. Satyanarayanan, "Advancing the state of mobile cloud computing," in *Proceedings of the third ACM workshop on Mobile cloud computing and services*, 2012, pp. 21–28.

[6] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: making smartphones last longer with code offload," in *Proceedings of the 8th international conference on Mobile systems, applications, and services.* ACM, 2010, pp. 49–62.

[7] R. Balan, J. Flinn, M. Satyanarayanan, S. Sinnamohideen, and H.-I. Yang, "The case for cyber foraging," in *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, 2002, pp. 87–92.

[8] H. Flores and S. N. Srirama, "Mobile cloud middleware," *Journal of Systems and Software*, vol. 92, pp. 82–94, 2014.

[9] 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.

[10] L. M. Vaquero, L. Rodero-Merino, and R. Buyya, "Dynamically scaling applications in the cloud," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 1, pp. 45–52, 2011.

[11] M. Mazzucco and M. Dumas, "Achieving performance and availability guarantees with spot instances," in *IEEE 13th International Conference on High Performance Computing and Communications (HPCC)*, 2011.

[12] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan, "Odessa: enabling interactive perception applications on mobile devices," in *Proceedings of the 9th international conference on Mobile systems, applications, and services.* ACM, 2011, pp. 43–56.

[13] Y. W. Ahn, A. M. Cheng, J. Baek, M. Jo, and H.-H. Chen, "An autoscaling mechanism for virtual resources to support mobile, pervasive, real-time healthcare applications in cloud computing," *IEEE Network*, vol. 27, no. 5, pp. 62–68, 2013.

[14] M. Mazzucco, D. Dyachuk, and R. Deters, "Maximizing cloud providers' revenues via energy aware allocation policies," in *IEEE 3rd International Conference on Cloud Computing (CLOUD)*, 2010.