A Microservice Store for Efficient Edge Offloading

Julien Gedeon, Martin Wagner, Jens Heuschkel, Lin Wang, Max Mühlhäuser
Telecooperation Lab, Technische Universität Darmstadt, Germany
Email: {gedeon, wagner, heuschkel, wang, max}@tk.tu-darmstadt.de

***Abstract*—Current edge computing frameworks require tight coupling between mobile clients and surrogates, i.e., the offloaded code has been preconfigured with its required execution environment. In many cases, this includes prior transfers of code blocks or execution environments from mobile devices to the offloading infrastructure. This approach incurs additional latency and is detrimental for the energy consumption of the mobile devices. In this paper, we propose the concept of a *microservice store*. Using the microservice abstraction common in software development and following the serverless paradigm, we envision a repository through which said services are made accessible to developers and can be re-used across applications. We implement a proof-of-concept edge computing system based on a microservice repository and demonstrate its benefits with real-world applications on mobile devices. Our results show that we were able to reduce latencies by up to 14× and save up to 94% of battery life.**

***Index Terms*—edge computing, fog computing, microservices, serverless, computation offloading, cyber foraging**

I. Introduction

The growing number of mobile devices and new applications such as augmented reality make compelling use cases for the deployment of close-by computing resources [1]. This trend of edge computing [2] or fog computing [3] has gained much attention both in academia and industry and consequently, a number of frameworks for edge computing have been proposed. A crucial part of edge computing is computation offloading, i.e., the remote execution of parts of an application. Many previous works investigate offloading approaches with varying granularity, e.g., code [4], threads [5] or VMs [6], while others focus on the offloading decision itself [7]. Common to all of them is the tight coupling between mobile clients and the edge infrastructure. Code mobility is realized by embedding the code and execution environment into virtual machines or containers that have to be transferred from the client to the surrogate that executes it. Besides the required energy for the transmissions, these transfers add to the end-to-end latency, i.e., the time it takes from the service request to returning the result to the user. For devices like smartphones, both of these factors are critical. Mobile users are often faced with unreliable low-bandwidth mobile networks and limited battery life. Performing traditional offloading hence has a negative impact on these factors and thus affects the overall quality of experience. In addition, current approaches do not support the re-use of service across application boundaries and therefore cannot use the overall resources efficiently.

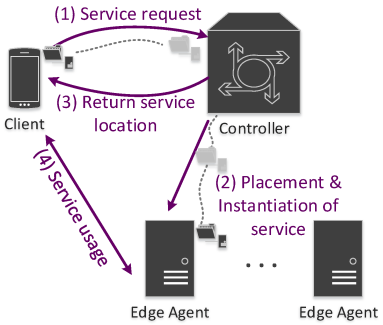
In order to mitigate these drawbacks, we advocate the idea of a *microservice store*. The microservice store is a repository to which developers can submit their code in the form of containerized applications. Developers of edge applications can incorporate these services into their application with the assurance that the code and execution environment will already be available at the edge. We believe this will largely simplify the development of innovative applications and encourage open source development. Client devices can request the instantiation of microservices for application execution and do not need to transfer code blocks and execution environments to the edge computing node. This can be done in two ways: (i) the client requests a specific microservice, addressed by a unique identifier from the microservice store or (ii) the client specifies a semantic description of the desired functionality and the format of the inputs and outputs, according to which an appropriate service will be instantiated automatically. While the former allows for an informed choice which microservice will be invoked, the latter enables developers to select a service solely based on the desired functionality. Microservice instantiation can also be customized, e.g., by overriding the default lifetime of the service. Microservices in the store can therefore be thought of as a blueprint, which we define by extending a common language for the orchestration of cloud services. Through the re-use of services for different applications, our approach also allows for a system-wide management of resources, e.g., by deciding which services are kept active, given request patterns from applications. To this end, we implement a prototype system to showcase the benefits of this concept and evaluate it using microservices accessed from applications on a smartphone.

In summary, this paper makes the following contributions: (i) We motivate the case for a new offloading mechanism based on the paradigm of microservices that are made available through a repository called the microservice store. We adapt a common language for cloud resource description to our edge computing scenario to provide a standardized description of microservices. (ii) We provide a proof-of-concept implementation that follows the paradigm of serverless computing using common technologies for edge computing such as container-based virtualization. (iii) Using three example microservices, we show the benefits of our approach w.r.t. latency and energy consumption. We further discuss architectural concerns for a wider-scale deployment, such as the distributed nature of a microservice store.

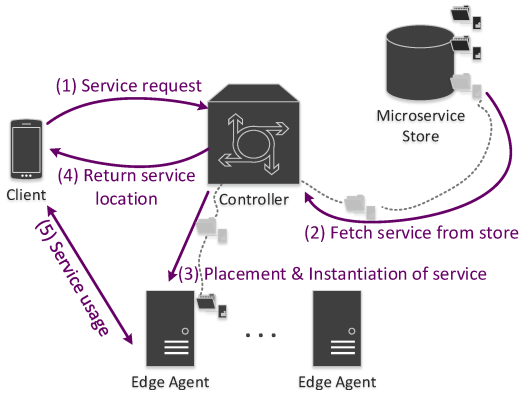
II. The Approach

We envision edge-enabled applications to rely on a repository of microservices in order to avoid the (prior) transfer of code and execution environments. Figure 1 contrasts these two

approaches of traditional computation offloading (Figure 1(a)) versus our proposed approach (Figure 1(b)). The individual steps of our approach will be described in Section II-D.



(a) Traditional offloading



(b) Proposed offloading through the microservice store

Figure 1. Comparison of offloading approaches

A. Characteristics of Microservices

Our offloading units are microservices, i.e., independent parts of an application. These services carry out a single, yet often computationally intensive task and can be composed into more complex applications. From an operations point of view, the individual services are developed and maintained independently from the applications that use them. The benefits offered by this approach have been widely acknowledged [8], [9] and include increased scalability and maintainability, which is crucial in dynamic edge environments.

B. Microservice Definition

In our system, a microservice is composed of its code and meta information about the service and its target execution environment (e.g., a virtualization technology such as containers or unikernels). The meta information is required to choose an appropriate service and to manage the services in the edge environment. We encapsulate both parts in a CSAR (Cloud Service Archive) file, a standard package format. For the meta information we extend TOSCA (Topology and Orchestration Specification for Cloud Applications), an open OASIS[1] standard for cloud service descriptions. Those additions include the definition of microservices, in particular microservices based on Docker containers and allow the definition of properties like expected memory usage, the network ports used by the microservice, its inputs and outputs, the alive time, and the category of the microservice in the store, among others. For Docker-based microservices, we also added a new TOSCA group to define Docker bridge networks, which the microservices can join. These properties are required by both the client and the edge computing framework. The former requires a description of the service and the latter information about expected memory usage, port mappings etc. to instantiate the service on edge nodes. An example microservice description is shown in Listing 1.

[1]Organization for the Advancement of Structured Information Standards

Listing 1. TOSCA description of a microservice

```
tosca_definitions_version:
    tosca_simple_profile_for_microservices_1_0_0
description: Template for a object detection application.
topology_template:
  node_templates:
    object_detection:
      type: tosca.nodes.microservices.docker_container
      properties:
        id: od01
        name: object_detection
        container_port: 5000
        mem_requirement: 1000
        directory: object_detection
        inputs: [ image ]
        outputs: [ image ]
        category: /image/object_detection
        alive_time: 1800
```

The attribute *alive_time* defines the default time a microservice should stay active on the edge agent. Contrary to common serverless platforms, where the lifetime is either limited to a single function execution or fixed by the provider (e.g., AWS Lambda currently has a maximum function lifetime of 15 minutes), this default lifetime can be overridden by the client in order to adapt it to the application. In addition, we support the idea of polling to monitor a microservice's activity and prevent early shutdowns or unnecessary restarts. The microservice can send an alive message to its agent to notify it of its activity status. This message in turn resets the alive timer of this microservice to prevent its automatic shutdown, thus allowing for a more efficient use of resources and avoidance of cold starts.

C. Microservice Store

The microservice store serves as the repository where services are uploaded and made available by developers. An entry for a microservice consists of a unique ID, the service name, a description, the category of the service, the types of its inputs and outputs, and the CSAR file. We model possible categories as hierarchies. A category may be the child of a parent category and the parent of several subcategories. For example, the category of our object detection microservice (see Section III-B) can be seen in Listing 1. This microservice belongs to the category *object_detection*, which in turn is a subcategory of the category *image*. The information about the category, the inputs and the outputs are especially important since they serve as the basis for our second microservice

addressing scheme that is based on the semantic description and allows for an automatic selection of services.

D. Control Flow

The control flow of our system is outlined in Figure 1(b). A client first issues a request for a service (step 1). The microservice is then fetched from the store (step 2) and instantiated on an edge agent (step 3). Alternatively, the controller can omit the former two steps if this service is already running. The controller then forwards the service location (e.g., the IP and port number from which it can be accessed) to the client (step 4). Because we want to decouple the mobile device from the agent, requests for microservices are sent to the controller. These requests are implemented as REST calls and must contain the ID of the microservice or a semantic description of a microservice. After receiving such a request, the controller first decides on which of the available edge agents to run the service. Afterward, a control protocol established between the controller and the agent is used to start the service. The agent then tries to start the microservice with the provided information and replies to the controller whether the start was successful or not. The controller then forwards this information along with the service location to the client.

Cold start vs. warm start: A cold start of a microservice happens when the client wants to use a microservice that is not yet running on an agent and therefore has to be transferred from the store to the agent and then started on that agent. In contrast, a warm start of a microservice happens when the client wants to use an already running microservice. In this case, the process of transferring the service from the store and starting it will be skipped, i.e., step 2 and step 3 in Figure 1(b), and the client will be informed about the running microservice.

Addressing of services: Microservices can either be referenced directly by their unique ID or semantically by matching their category. In the latter case, the client has to provide a semantic description consisting of a microservice category from the store and a definition of inputs and outputs in their request. Based on this information the controller selects a matching microservice from the store and informs the client if no matching service is found.

III. Implementation

We implement our approach described in an edge computing platform (Section III-A) and develop three microservices to showcase it (Section III-B).

A. Edge Computing Platform

Our system consists of (i) a (logically centralized) *controller*, to which clients submit a request for the execution of a microservice, (ii) *edge agents* that run the containerized microservices, and (iii) the *microservice store*. Controller and agents are implemented as Python applications. The controller exposes a REST-style API to clients in order to start microservices and control their execution. The controller maintains a list of edge nodes in the system and has a global view on the system, including which microservices are currently running on which nodes. For this, we implement the exchange of control messages between the controller and the edge agent. To parse the microservice descriptions, we adapt the TOSCA parser from the OpenStack project[2]. The edge agents consist of a Python application that maintains connectivity to and exchanges information with the controller. In addition, it controls the execution environment. We use Docker as a lightweight virtualization for the execution environment. The agent reacts to instantiation requests coming from the controller. If the corresponding Docker image does not exist on the agent yet, it will be built upon requesting the service. Future invocations use the pre-built image unless the client sends a *force_rebuild* flag. This is useful when using the semantic invocation of services to ensure newer versions of services are executed. Furthermore, clients are able to stop microservices without removing them and restart stopped microservices as well. Microservice developers can use Docker networking by defining Docker bridge networks, which the microservices are able to use. This way, microservices running on the same agent can be isolated from one another by joining different bridge networks. The microservice store is realized using MongoDB, a document-oriented database.

B. Demo Microservices

To evaluate our system, we develop three microservices. We make them available for the research community[3]. All microservices are implemented in Python, use Flask and Bottle to implement a REST-based Web-API and are shipped as Docker containers.

Object detection: Using TensorFlow, we perform object detection on an image captured by the phone's camera. The microservice returns the original image including the detected objects, enclosed by rectangles and labeled with the name of the object and the confidence value. The microservice code is adapted from the TensorFlow Object Detection API[4] and we use the *ssd_mobilenet_v1_coco* model trained on the COCO dataset.

Face detection: Using OpenCV and a *LBP Cascade* classifier, our second microservice detects faces in an image and returns the original image with the faces enclosed by rectangles.

Word count: Lastly, we use a simple word count application that counts the number of words in a given text file.

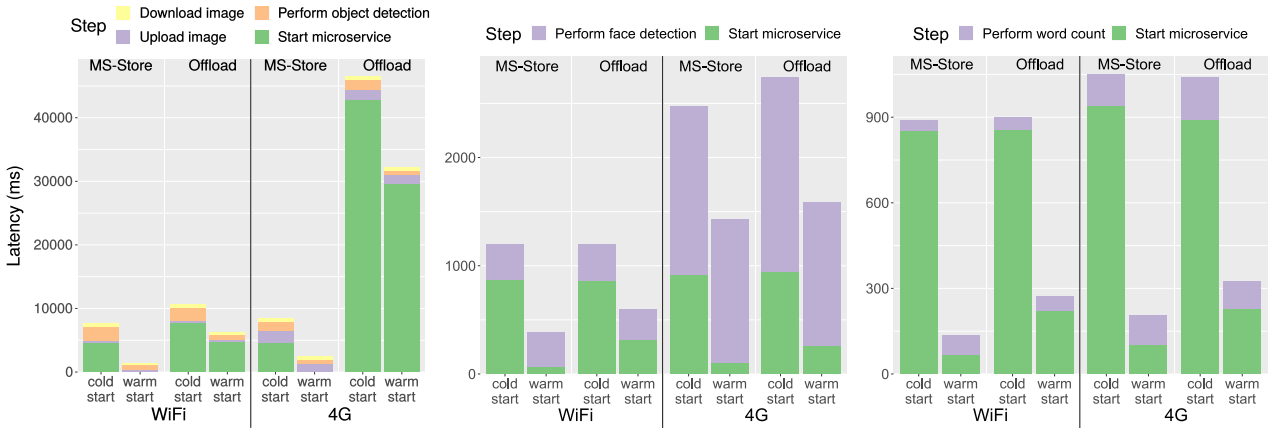
IV. Evaluation

In this section, we evaluate our microservice store approach in comparison to traditional offloading and the local execution on a phone. We further discuss future challenges of a microservice-style architecture for edge computing.

[2]https://github.com/openstack/tosca-parser (Accessed: 2019-04-08)

[3]https://github.com/Telecooperation/flexEdge-microservices (Accessed: 2019-04-08)

[4]https://github.com/tensorflow/models/tree/master/research/object_detection (Accessed: 2019-04-08)



(a) Object detection (b) Face detection (c) Word count

Figure 2. End-to-end latency for the different microservices

A. Experimental Setup

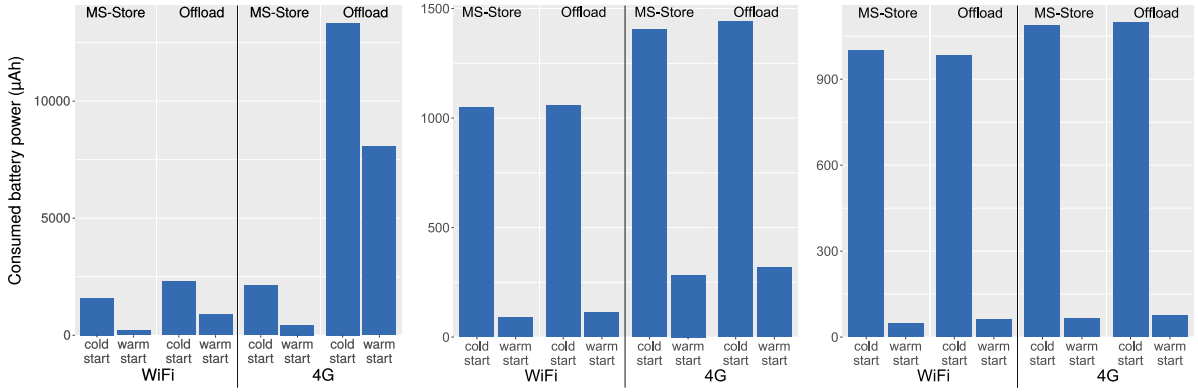
As an edge node, we use a Lenovo ThinkCentre M920X Tiny with an Intel Core i7-8700 and 16GB RAM running Ubuntu 18.04. The edge node is connected via Gigabit Ethernet to a Linksys WRT 1900 AC wireless access point that at the same time serves as a 802.11nac gateway for the mobile device. Besides WiFi connectivity, we also conduct our experiments using a 4G mobile network with a theoretical maximum bandwidth of 300/50 Mbit/s (down/up). The edge controller and the microservice store are colocated on the same machine, a Citrix Xen VM (AMD Opteron 6380, 8GB RAM) running Ubuntu 16.04. This VM runs in the same backend network as the edge node. As a mobile client device, we use a Google Pixel 2XL phone (Qualcomm Snapdragon 835, 4GB RAM) running Android 9. We assume the mode of operation in which microservices in the store are selected based on their IDs.

B. Latency

First, we evaluate the end-to-end latency, i.e., the time between the service request from the mobile application and the receiving of the result. We plot the corresponding mean values in Figure 2. Because of their different complexity and size, we plot the results for the different microservices individually. For our analysis, we split the overall execution into two steps: The time it takes to start the microservice and the actual task execution (including the transfer of input and result data). For the object detection, we further split up the second step into uploading the image to the microservice, performing the object detection, and the download of the resulting image to better analyze the impact of the individual steps on the overall latency. Using our approach in the warm start mode results in a reduction of the end-to-end latency at the first step in all microservices compared to the traditional offloading approach. Using a cold start, the end-to-end latency reduction depends on the size of the microservice. In our experiments, the CSAR file of the object detection microservice is the largest at 28MB, while the face detection and word count microservices are 12KB and 2KB in size, respectively. In the traditional offloading approach, the CSAR file has to always be transferred from the mobile client to the controller, which explains the latency reduction at the first step for the object detection microservice and the comparatively smaller reduction for the face detection microservice. In the case of the word count microservice, our approach leads to a small increase in end-to-end latency compared to the traditional approach using a cold start and the 4G network. Instead of transferring the CSAR file, our approach has to fetch the microservice from the store, which induces a short delay as well. The results suggest that this delay is comparable to the delay introduced by transferring small CSAR files as is the case for the face detection and word count microservices. The higher latency in the second step of the face detection and word count microservices using the 4G network in comparison to the WiFi network occurs because the second step includes the transfer of an image or text file. Overall, we see an average time reduction of 1.1–14 $\times$ for microservice startup, depending on the microservice, the start mode, and the type of network.

C. Energy Consumption

We now show how our approach saves energy on the mobile device. We measure the energy consumption by using the readings from the Android Battery Manager. In particular, we use the property `BATTERY_PROPERTY_CHARGE_COUNTER`, which provides us the remaining battery capacity. Figure 3 shows the results of the consumed battery power for a single execution, averaged from all measurements. In general, the results suggest a correlation between the end-to-end latency and the consumed battery power. Similar to the results from the latency evaluation, the energy savings depend on the microservice and whether the service is invoked in cold or warm start. Using the warm start, our approach results in reduced energy consumption for all microservices. The reduction depends on the size of the microservice. A larger CSAR file size leads to a larger upload in the traditional



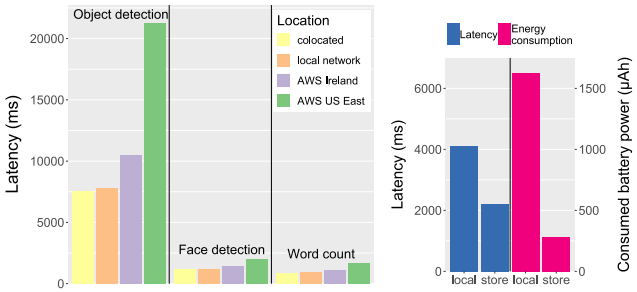
(a) Object detection (b) Face detection (c) Word count

Figure 3. Comparison of energy consumption

approach, which in return requires more energy. Using a cold start reduces the energy benefit of our approach, again depending on the size of the microservice, which is especially apparent when looking at the face detection and word count microservices. In our experiments, our approach led to a negligible increase in energy consumption (likely due to measurement inaccuracies) for the special case of the word count in cold start on WiFi. Comparing the differences between cold starts and warm starts for the face detection and word count services, we found that a warm start greatly reduces the consumed battery power, while the latency reduction is less substantial. As can be seen from Figures 2(b), 2(c), 3(b) and 3(c), the warm start decreases the latency by about 50–75%, while the battery consumption is decreased by about 80–95%. These results show that our approach is overall beneficial for the user in the sense that it increases the battery lifetime of mobile devices.

D. Microservice Store Location

In the previous experiments, the microservice store was colocated at the edge controller. If we envision a large-scale deployment of our system, we can make two observations: (i) ideally we would not have a single controller, but a (distributed) hierarchy of controllers, each one responsible for one (geographic) region and (ii) the microservice store might also be distributed and the individual instances not necessarily colocated with the controller. We now measure the impact of the store location on the latency. We consider the microservice store to be (i) colocated on the controller, (ii) in the same local network, and (iii) in two locations using Cloud services. For the latter, we use AWS EC2 instances in the regions US East and Ireland. Our location is in Darmstadt, Germany. The results are shown in Figure 4(a). Depending on the size of the microservice, the store location significantly affects the invocation latency. While a store located in the same network has a relatively small impact on the latency, using Cloud services increases the latency by 33%-200%. Hence, for a viable deployment, the microservice store should be close to the controller and agents to not negate the benefits of our offloading scheme. However, in practice, microservice store, controller and agents would likely be well-connected via wired networks, compared to more unreliable (and sometimes metered) wireless networks that mobile clients use. Hence, even with additional network hops to transfer the microservices to the agents, our approach is beneficial for the overall latency.



(a) Impact of the microservice store location on latency (b) Comparison with local execution on smartphone

Figure 4. Impact of the store location and local execution

E. Comparison with Local Execution

We now compare how a local implementation, i.e., an alternative version of the service that runs directly on the phone, performs against its counterpart executed on the edge agent via the microservice store. The code for the local execution is adapted from the TensorFlow Android Camera Demo[5]. In this experiment, we use warm start execution through WiFi and use *faster_rcnn_inception_v2_coco* as a model. Figure 4(b) shows the result. Our approach leads to a reduction in execution time of about 50%. Furthermore, the consumed energy of the local execution is almost six times higher than the consumed energy of the offloaded execution through the microservice store.

[5]https://github.com/tensorflow/tensorflow/tree/master/tensorflow/examples/android (Accessed: 2019-04-08)

F. Discussion and Future Work

Customization of services: One microservice can be implemented and executed in different ways. For example, object detections can rely on different trained models, each with varying complexity, accuracy, and resource requirements. For the future development of our system, we envision the TOSCA files to serve as blueprints for services. At invocation time, users could specify the desired characteristics of the service. Alternatively, in order to manage demands and resources, this could be managed by the controller.

Chaining of microservices: In the case that one microservice directly takes as an input the result of another, instead of passing the intermediate result through the client, these microservices should be jointly orchestrated and ideally executed on the same agent in order to avoid expensive transfer of data.

QoS-awareness and guarantees: Current edge computing frameworks do not include quality of service guarantees. Especially in the case of chained services as outlined above, we envision continuous monitoring of the entire execution pipeline. This monitoring can serve as a basis to adapt the service instances at runtime in order to meet QoS goals, e.g., by trading the quality of the computation.

Caching and pre-building of microservices: For a fully distributed design of the microservice store, we need to make decisions which services to cache at a location. Future work should also investigate the tradeoff between cold-start latencies and efficient resource usage. Depending on request patterns, it might be beneficial to keep popular services warm; however, idle services consume scarce resources on the edge nodes.

V. Related Work

Computation offloading or cyber foraging is the process of remotely executing parts of an application. Sharifi et al. [10] review this general concept and summarize research challenges. Balan and Flinn [11] argue that challenges like server setup and maintenance are still not solved. We believe our approach at least partly frees developers from these burdens, as our edge computing platform is responsible for the instantiation and management of services. A variety of offloading frameworks have been proposed. Common to all them is that code—and in some cases the entire execution environment—has to be transferred from the client device to the surrogate. CloneCloud [12] automatically partitions the application via dynamic profiling. Cuervo et al. present MAUI [4], an offloading framework for mobile phones that focuses on the energy benefit of offloading. Other works operate on a thread-level granularity [5], focus on stream processing applications [13] or propose parallel execution [14]. Close to our work is Paradrop [15], a platform for the dynamic orchestration of third-party services at the edge; however they do not consider the aspect of energy savings for the mobile device. Microservices are a way to develop and deploy software as independent parts, in contrast to monolithic software [8]. It is widely recognized that this offers many benefits regarding DevOps [9]. The concept of delivering microservices as containers has been suggested in [16].

VI. Conclusion

In this paper, we presented our approach of a *microservice store* as a contrast to common offloading techniques in edge computing. Our evaluation based on real-world applications such as object detection in images showed the benefits w.r.t. reduced end-to-end latency and prolonged battery lifetime for mobile devices.

Acknowledgement

This work has been cofunded by the German Research Foundation (DFG) and the National Nature Science Foundation of China (NSFC) joint project under Grant No. 392046569 (DFG) and No. 61761136014 (NSFC), and as part of the Collaborative Research Center 1053 - MAKI (DFG). This work was supported by the *AWS Cloud Credits for Research* program.

References

[1] H. Chang, A. Hari, S. Mukherjee, and T. V. Lakshman, "Bringing the cloud to the edge," in *Proc. of the IEEE INFOCOM Workshops*, pp. 346–351.

[2] M. Satyanarayanan, "The emergence of edge computing," *IEEE Computer*, vol. 50, no. 1, pp. 30–39, Jan. 2017.

[3] J. Gedeon, J. Heuschkel, L. Wang, and M. Mühlhäuser, "Fog computing: Current research and future challenges," in *1. GI/ITG KuVS Fachgespräche Fog Computing*, 2018.

[4] E. Cuervo, A. Balasubramanian, D. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "MAUI: making smartphones last longer with code offload," in *Proc. of ACM MobiSys*, 2010, pp. 49–62.

[5] M. S. Gordon, D. A. Jamshidi, S. A. Mahlke, Z. M. Mao, and X. Chen, "COMET: code offload by migrating execution transparently," in *Proc. of USENIX OSDI*, 2012, pp. 93–106.

[6] T. Verbelen, P. Simoens, F. D. Turck, and B. Dhoedt, "AIOLOS: middleware for improving mobile application performance through cyber foraging," *Journal of Systems and Software*, vol. 85, no. 11, pp. 2629–2639, 2012.

[7] X. Chen, L. Jiao, W. Li, and X. Fu, "Efficient multi-user computation offloading for mobile-edge cloud computing," *IEEE/ACM Trans. Netw.*, vol. 24, no. 5, pp. 2795–2808, 2016.

[8] M. Fowler, "Microservices - a definition of this new architectural term," https://martinfowler.com/articles/microservices.html, 2014, accessed: 2019-03-04.

[9] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices architecture enables devops: Migration to a cloud-native architecture," *IEEE Software*, vol. 33, no. 3, pp. 42–52, 2016.

[10] M. Sharifi, S. Kafaie, and O. Kashefi, "A survey and taxonomy of cyber foraging of mobile devices," *IEEE Communications Surveys and Tutorials*, vol. 14, no. 4, pp. 1232–1243, 2012.

[11] R. K. Balan and J. Flinn, "Cyber foraging: Fifteen years later," *IEEE Pervasive Computing*, vol. 16, no. 3, pp. 24–30, 2017.

[12] B. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "CloneCloud: elastic execution between mobile device and cloud," in *Proc. of EuroSys*, 2011, pp. 301–314.

[13] M. Ra, A. Sheth, L. B. Mummert, P. Pillai, D. Wetherall, and R. Govindan, "Odessa: enabling interactive perception applications on mobile devices," in *Proc. of ACM MobiSys*, 2011, pp. 43–56.

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

[15] P. Liu, D. Willis, and S. Banerjee, "Paradrop: Enabling lightweight multi-tenancy at the network's extreme edge," in *Proc. of IEEE/ACM Symposium on Edge Computing (SEC)*, 2016, pp. 1–13.

[16] A. Sill, "The design and architecture of microservices," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 76–80, 2016.