# Docker Enabled Virtualized Nanoservices for Local IoT Edge Networks

Johirul Islam*, Erkki Harjula*, Tanesh Kumar*, Pekka Karhula†, Mika Ylianttila*
*Centre for Wireless Communication, University of Oulu, Finland
†VTT Technical Research Centre of Finland
{firstname.lastname}@oulu.fi*, pekka.karhula@vtt.fi†

***Abstract*—Edge computing is a novel computing paradigm moving server resources closer to end-devices. It helps unleashing the full potential of high-performance access networks with respect to ultra-low latency and transfer rate and improve resilience to problems at core networks and data centers. Multi-access Edge Computing (MEC), is a standard solution by European Telecommunications Standards Institute (ETSI) for access network-level edge computing. MEC, operating at access network level, is an ideal solution for the most cases. However, there are still some challenges to address: first is related to the vulnerability to access network problems and the second is about the high load inflicted to access networks and MEC servers. This is a particular issue in massive-scale Internet of Things (IoT) use cases, where numerous sensors may produce high amounts of data, or where critical system functionalities must be ensured also during access network problems. In this paper, we study the feasibility of bringing some edge functions to the local level as virtualized and dynamically deployable components utilizing local hardware capacity. For the study, we have implemented a local edge networking prototype based on local microservices, called *nanoservices*, implemented using Docker containers and deployed using Docker Swarm-based orchestration. Since IoT networks typically consist of constrained-capacity devices, our focus is in optimizing the resources of the proposed nanoservices.**

***Index Terms*—Edge Computing; IoT; Microservices; Nanoservices; Virtualization; Orchestration.**

## I. Introduction

Internet of Things (IoT) has significant impact in almost each aspect of our daily life. It is mainly because IoT systems can provide practical, efficient and feasible solutions to various key applications such as healthcare, transportation, surveillance and banking, among many others [2]. Furthermore, IoT is considered as a driving force behind the vision of ubiquitous and pervasive computing to ensure the availability of digital services anywhere and anytime. Such IoT networks together with recent enabling technologies are progressing towards new digital paradigm where digital services can be acquired without using any hand-held gadgets. In this vision, gagdet-free users are able to access intelligent and context-aware services from nearby surroundings [3], [4]. Such IoT based smart applications would have high demanding requirements in terms of low latency, less cost, energy efficiency, scalability and better quality of services (QoS) among others.

Moreover, future smart environments require not only technological advancements but also transformation from the perspective of network architectures. For example, the introduction of Edge computing in the traditional IoT networks have been significant for providing low latency services to large-scale IoT applications [5]. Such applications are mainly delay-critical and require relatively faster data processing, computations and the decision/response making for the different phases. For example, a robotic arm working in a smart factory processes is time-critical and require minimum precision/delay at various phases. The traditional cloud based solutions may not very appropriate in such cases due to low-latency and high bandwidth requirements. The edge paradigms are crucial as they extend the functionalities, capabilities and services closer to the edge of the network.

To enhance the elasticity and scalability of applications, there have been various network architecture for IoT and Cloud computing that are proposed in the literature [6], [7]. One of the key requirements is the ability to provide critical services at the local level of the IoT networks, where the resources are limited. We have recently proposed a three-tier IoT edge architecture for accessing gadget-free digital services as shown in Fig. 1 [1], [8]. This architecture emphasizes the

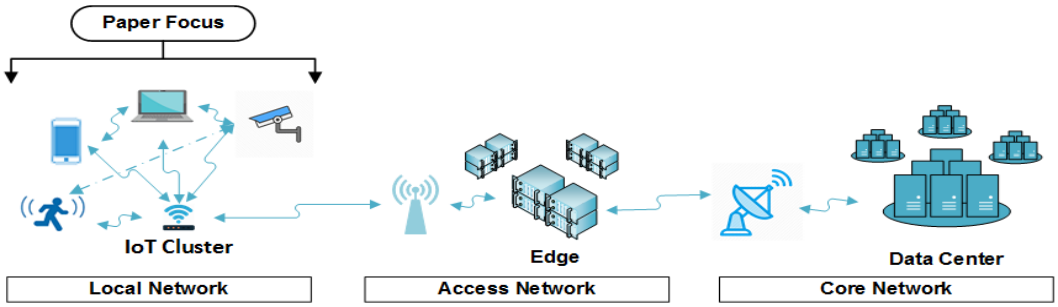


Fig. 1: Three-tier future IoT network [1].

significance of locally available nodes to efficiently utilize the resources and needed functionalities. The local IoT-layer generally has less resources, such as the minimal availability of hardware resources, limited network access and highly variable device environment, among others. Docker container based solutions are becoming popular to provide virtualized services for resource-constrained IoT systems [9].

This paper presents the implementation behind the gadget-free IoT edge architecture introduced in [1], and provides a detailed evaluation for it. The rest of the paper is organized as follows: section II explains the background and recent state-of-the-art in this domain. Then we describe the selected use case in section III. Section IV presents the Proof-of-Concept (PoC) implementation based on the defined use case. Results evaluations are highlighted in Section V and finally, we provide discussion and conclusion in the section VI.

## II. Background

### A. Edge and Fog Computing

Edge Computing (EC) introduces a new layer of services between the cloud and end-devices at the edges of networks. Compared to traditional cloud architectures, EC greatly improves the latency between IoT nodes and servers by reducing the physical distance between data sources and computational resources. Furthermore, it helps reducing load at core networks and data centers by providing computational capacity. It also helps to resolve security and privacy problems by limiting the scope of propagation of private data along the public networks [10].

MEC [11] provides an edge platform by re-equipping the cellular network base stations for application execution and local data storage. The distinct feature of MEC follows from this placement as real-time radio access network information is exposed in MEC hosts to improve their application execution. Fog computing (FC) [10] architecture is distributed a step further from the EC, although the common definitions of Fog varies. The edge applications in Fog are executed in a set of fog nodes, at any point in the network, forming a "cloud-thing continuum" [12]. Examples of fog nodes include legacy and edge devices connected through a Local Area Network, such as common household appliances, e.g. set-top boxes, and the network infrastructure components [12], [13], [14].

### B. Container-based microservice architectures

During the past few years, cloud services have been transforming from monolithic architectures towards microservice-based architectures, where each services taking care of a limited set of functions [15], [16]. The benefits of using microservice architectures include: flexibility, scalability, efficiency, reduced complexity and improved manageability. Microservice can be implemented using container technology [17], where only one or few processes run inside a single container. Docker containers provide a lightweight, low overhead and fast technology empowering the usage of microservice architectures [9]. When the microservices are smaller enough and could be managed locally, they can be referred as nanoservices [1]. In this paper, since microservices can be managed locally, then we will call them nanoservices instead of microservices.

Orchestration is a technology for controlling interactions between virtualized components such as containers and service management. The most commonly used container orchestration technologies are Docker Swarm, Kubernetes and Mesos, all of them provide automated support to, e.g., service discovery, load balancing, and software upgrades [18]. The orchestration at Edge and Fog layers is a complex problem due to the increased distribution and inter-distances of large number of nodes [19] [20].

### C. Our previous work

In our recent projects, we have initiated a path towards the paradigm shift in the relationship between people and the digital world [3], [4], [8], [21]. In this vision, user lives "naked" without gadgets and services materialize for the user when they are needed and disappear when not needed. The digital surroundings form an intelligent environment around users, providing all the information, tools, and services that users need in their everyday life. To realize the above-mentioned vision, we have proposed a novel virtualized and decentralized nanoservice-based model, *nanoEdge* [1], where nodes, based on their hardware capacity and load, collaboratively provide local processing, storage, security and privacy services without relying on centralized entities. With the concept of *nanoservice*, we refer to the miniature version of *microservices* used in cloud computing. In this paper, we present this Proof-of-Concept (PoC) implementation in more detail and deepen the evaluations by providing more detailed information on how we succeeded to optimize the nanoservice storage footprint and deployment time suitable for deployment on constrained-capacity IoT devices and networks.

## III. Example service

### A. Use case scenario

We use a smart meeting event use case including four participants who are going to attend the meeting event, namely; Alice, Bob, Carl and David. Alice is responsible for organizing the meeting and inviting other team members. She creates a meeting event and sends the request to her group members. Each of the team members is required to authenticate him/herself securely with smart surrounding to ensure that valid participants are allowed for the meeting and given access for confidential meeting topics. Therefore, Alice needs to deploy a secure system locally at the entrance (smart surroundings) of the building which is able to authenticate the user and lead the participant way from the entrance to meeting room.

Figure 2 shows the smart meeting event use case and numbers in the red circles highlights key services of the system. This secured system is required to provide three key services, i) to authenticate the valid participants for the meeting (1 to 4), ii) to provide guidance to valid participants from the main entrance to the meeting room (5), and iii) to present the meeting contents for the authorized participants (6,7). In this scenario, we assume that gadget-free devices such

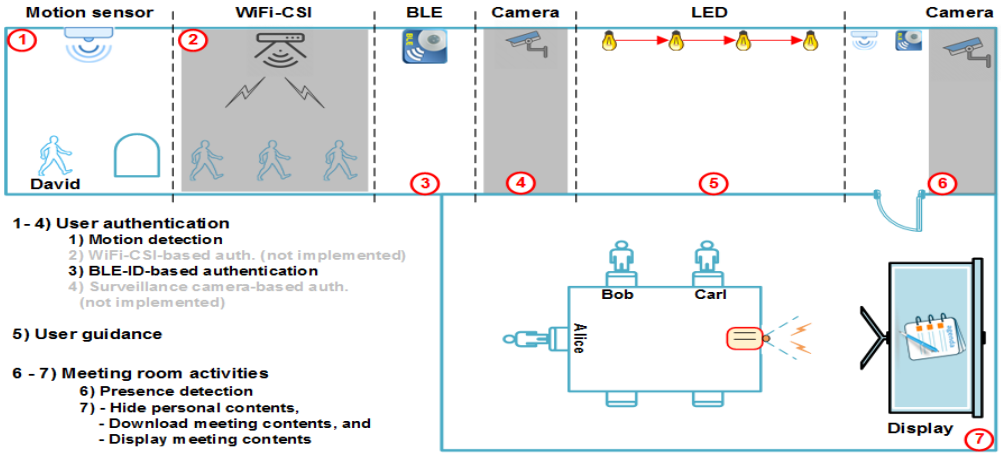


Fig. 2: Smart meeting event as a use case scenario [1].

as smart rings can be utilized for identification purposes. The main focus is stressed upon how local resources can be utilized efficiently due to limited resources at this level.

### B. Service structure

Following the service model given in [1], we divide the three key services further into six nanoservices as shown in Fig. 3. New six nanoservices are: i). Presence Detection (PD) service to detect motion around the effective area of the sensor node, ii). Main Controller Service (MCS) to control service-to-service interactions, iii). Bluetooth Scanner (BS) service to authenticate a participant, iv). Authentication Engine (AE) service to store events information, v). User Guidance (UG) to guide participant from entrance to meeting room; and vi). Meeting Room Service (MRS) to manage all tasks related to meeting room.

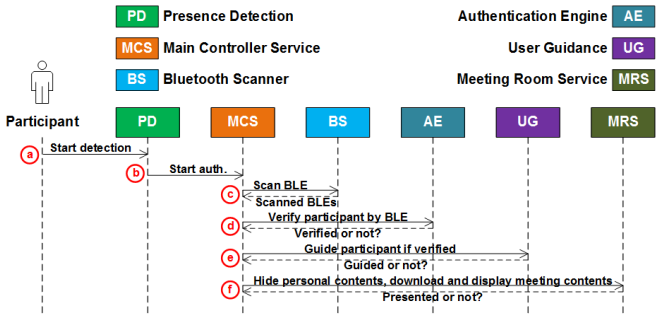


Fig. 3: Interactive nanoservices.

Figure 3 shows the service interaction for the selected use case scenario (small alphabet in red circles tell about interaction between two nanoservices). PD will be activated automatically if a participant is detected (a) and then sends a request to MCS (b) to start BS service. The BS will start scanning Bluetooth devices and sends the information an those back to MCS (c). Meanwhile, MCS sends request to AE to recognize a participant with his/her BLE ID (d). On successful recognition, MCS sends a request to UG to guide the recognized participant to the meeting room (e). When the participant arrives at the meeting room then PD and BS tell MCS to stop UG for that participant. Finally, MCS sends a request to MRS to hide participantal contents, download meeting contents and display those contents at a User Interface (UI) (f).

A service administrator (admin) can create a nanoservice, add and remove a new/existing nanoservice; and terminate any/all nanoservices as shown in Fig. 4 (numbers in red circles). Initially, there were five nanoservices running into the system (1). Next, the service admin adds sixth BS nanoservice to the system (2) and then the system admin removes the AE nanoservice from this system (3). In the system has an automatic backup for it's storage, all valuable data, code and so on before termination of any deployed services (3, 4).

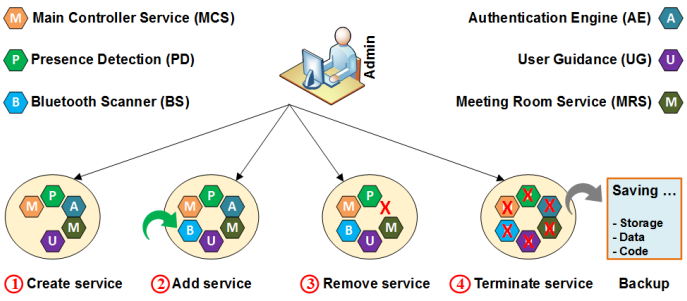


Fig. 4: Nanoservice creation, modification and termination [1].

## IV. Proof-of-Concept

### A. Hardware and software specifications

As mentioned above in the use case, six nanoservices were developed. In our implementation, UG nanoservice deployed into a Raspberry Pi 2 (RPi 2) Model B which has 512MB RAM, 700MHz CPU and 16GB SD card with 10MB/s data processing speed. On the other hand, AE nanoservice deployed into a Ubuntu 18.04 based OS laptop which has 4GB RAM, Intel Core i3 with 2GHz CPU. Rest of the four nanoservices deployed on four separate Raspberry Pi 3 (RPi 3) Model B+ having same configuration such as 1GB RAM, 1400MHz CPU and 16GB SD card with 10MB/s data processing speed. An

extra machine having 8GB RAM, Intel Core i7 1.6GHz CPU responsible to make and manage the cluster along all these resources. Here, all these nanoservices have been deployed under a single network access point. This access point has installed at the University of Oulu and City of Oulu with SSID '*panoulu*' which has 100 Mbps internet speed [22].

TABLE I: Hardware and software specifications

| Services | Hardware | Software |
|---|---|---|
| PD | PIR sensor, RPi 3 | Python 2, thThings |
| MCS | RPi 3 | Python 2, thThings |
| BS | BLE key, RPi 3 | Python 2, thThings |
| AE | Ubuntu 18.04 Laptop | Python 3, thThings, Flask, MySQL |
| UG | Arduino shield, LED, RPi 2 | Python 2, CoAPthon |
| MRS | RPi 3, UI | Python 2, thThings |

The implementation also includes some of software specifications as mentioned in Table I. All the nanoservices were developed with Python where Constrained Application Protocol (CoAP) enables machine-to-machine (M2M) communication among these IoT devices. In the implementation, AE has been developed with Python 3, Flask 1.0.2 (Flask RESTful 0.3.6); and MySQL 8.0.16 where UG has been developed with C++ (GCC 5.3); and Python 2. Four other nanoservices were developed with Python 2. Here, CoAPthon 4.0.2 [23] were used at the UG nanoservice where thThings 0.3.0 [24] were used at the rest five nanoservices for server side communication.

### B. Implementation of example nanoservices

In general, every Docker container requires a container image to deploy a service into the Docker system. In addition to that, a container image must have at least a special structure known as a *base image*. In the implementation, initially, official Docker base images (created by Docker community) has been introduced to create nanoservices.

Initially, Python script was used to handle request-response cycle between a client and a server separately. This process increases the number of containers which may cause for overhead and may be difficult to control those. Then *npm* based *coap-cli* package introduced to make client request in associated server with the help of python's *subprocess* [25] module as discussed in Soha's blog [26] which significantly reduced the number of containers those made for client and hence reduces extra memory consumption.

### C. Container orchestration

Container orchestration has been done with Docker Swarm and Kubernetes. Command Line Interface (CLI) based Docker Swarm is integrated into the Docker ecosystem with its own API while Graphical User Interface (GUI) based Kubernetes requires third party API to manage containers. Here, we will focus on the Docker Swarm based orchestration. Our required six nanoservices had to deploy into their respective position with different sensor specific host or node as illustrated in Fig. 2.

A dedicated host machine (as mentioned in A on VI) used as manager (having Docker) to make a cluster of nodes known as *swarm* where rest of the nodes used as workers (having

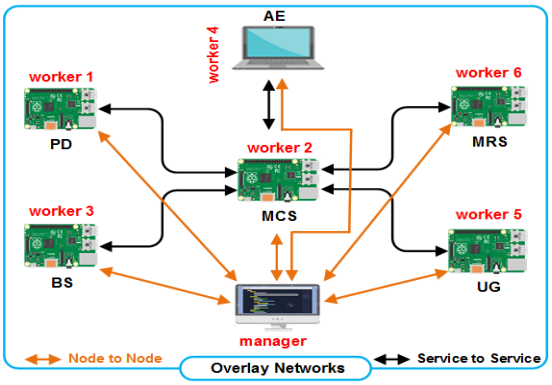


Fig. 5: Container orchestration for the PoC prototype.

Docker with required Docker container images). Orchestration were implemented on top of *Overlay Networks* as highlighted in Fig. 5, since it allows us to manage multi-hosted Docker containers [27]. The orchestration includes following steps: *(1) Swarm Initialization:* Cluster initialized at manager node. *(2) Joining Workers:* Each worker added to the manager with a *token*. *(3) Labeling Nodes:* In the manager, each node labeled with their respective *IP*. *(4) Nanoservices Deployment:* Services deployed into the cluster with their label specifications.

## V. Evaluations

We evaluate the measurement results based on the real life measurement for our implementation in terms of resource consumption and performance of deployed nanoservices.

### A. Resource consumption

Resource consumption is one of the key factors for any IoT devices. During the implementation, images created in Docker for nanoservices were managed by Docker Swarm and Kubernetes. The Docker system required 96 MB whereas Kubernetes took 196 MB. Here, we analyze the resource consumption based on three separate steps (images) for the feasible implementation namely; official, alpine and multi-stage-builds custom images. These measurement were taken for five times for each cases and gave same results as shown in Table II, III and IV.

*1) Using official images:* The size of a container image is calculated with the size of it's base image and the size of the dependencies required for a container. The nanoservices in Table II use official base image where AE requires 778.7 MB, which is the sum of Base image size 690.5 MB (column 3 in the table) + Unique Container size 88.18 MB (column 4), to build the nanoservice into the Docker system. Meanwhile, AE requires another 485.5 MB of disk space as it depends on MySQL component (version 8.0.12) as another nanoservice. However, rest of nanoservices such as MCS, PD, BS, MRS; and UG consumed 599.6 MB, 599.6 MB, 609.8 MB, 612.2 MB; and 404.7 MB respectively.

Nanoservices demonstrated in Table II consumed huge amount of disk space which is not good and suitable for low capacity IoT devices where official base image affects the implementation. Traditionally, an official base image includes

TABLE II: Resource consumption using official images

| Component or Service | Consumed Disk Space | | | | |
|---|---|---|---|---|---|
| | Base Image | | Container Size | | |
| | Name | Shared (MB) | Unique (MB) | Total (MB) | Run |
| AE + MySQL | python:3-onbuild + mysql:8.0.12 | 690.5 + 0 | 88.18 + 485.5 | 778.7 + 485.5 | 414 kB + 7 B |
| MCS | arm32v7/python :2.7.15-jessie | 557.3 | 42.31 | 599.6 | 231 kB |
| PD | | | 42.34 | 599.6 | 235 kB |
| BS | | | 54.49 | 609.8 | 860 kB |
| MRS | | | 54.94 | 612.2 | 1245 kB |
| UG | resin/rpi-raspbian :jessie | 128.2 | 276.5 | 404.7 | 148.3 MB |

all the dependencies even though those may not require to run an application.

*2) Using alpine images:* A tentative solution was required to reduce memory consumption that consumed by nanoservices developed for usecase scenario. Table III summarizes the memory consumption using suitable alpine images. At the time of implementation, there has no alpine-based Docker images for MySQL. However, MySQL version 8.0.16 is used in this case which reduces 50 MB compared to version 8.0.12. On the other hand, AE, MCS, PD, BS, MRS; and UG took the total (summing column 3 and 4) disk space of 273.8 MB, 184.6 MB, 184.8 MB, 273.8 MB, 193.8 MB; and 324.7 MB respectively. All these nanoservices are now able to save roughly 64.8%, 69.2%, 69.2%, 55.1%, 68.3%; and 52.7% of disk space respectively as compared to the implementation that uses official base images.

TABLE III: Resource consumption using alpine images

| Component or Service | Consumed Disk Space | | | | |
|---|---|---|---|---|---|
| | Base Image | | Container Size | | |
| | Name | Shared (MB) | Unique (MB) | Total (MB) | Run |
| AE + MySQL | python:3-alpine + mysql:8.0.16 | 86.98 + 0 | 186.8 + 443 | 273.8 + 443 | 394 kB + 7 B |
| MCS | arm32v7/python :2-alpine | 52.99 | 131.6 | 184.6 | 231 kB |
| PD | | | 131.8 | 184.8 | 231 kB |
| BS | | | 220.8 | 273.8 | 239 kB |
| MRS | | | 140.8 | 193.8 | 231 kB |
| UG | | | 138.4 | 191.4 | 17.6 kB |

Basically, alpine based image enables us to run an application just requiring dependencies to build and run an application. In the implementation, these alpine image based solution include only the required dependent libraries and hence reduce the total consumed memory space of IoT devices used to implement the use case.

*3) Using multi-stage-builds images:* Docker version 17.05 facilitates multi layer caching [28] which allows us to reduce the memory consumption even more. In general, multi-stage-builds create many layers to optimize a docker container image at the development time. In our case, we build an image in two separate layers where first one is for compiling dependencies and second one for managing dependencies to run the image. At the run-time, however, the final images size (sum of column 3 & 4) for AE, MCS, PD, BS, MRS and UG nanoservices took 137.13 MB, 92.19 MB, 92.36 MB, 93.71 MB, 92.14 MB; and 73.9 MB of memory space sequentially. That means, when we introduced multi-stage-builds feature to our simple alpine-based images then they can additionally saved 49.9%, 50.1%, 50.0%, 65.8%, 52.5% and 61.4% of disk space. On the other hand, these nanoservices for AE, MCS, PD, BS, MRS; and UG were able to save 82.4%, 84.6%, 84.6%, 84.6%, 85.0%; and 81.7% of memory space respectively when they were compared to those images which were built on top of an official base image as given in Table IV.

TABLE IV: Resource consumption using multi-stage-builds

| Component or Service | Consumed Disk Space | | | | |
|---|---|---|---|---|---|
| | Base Image | | Container Size | | |
| | Name | Shared (MB) | Unique (MB) | Total (MB) | Run |
| AE + MySQL | python:3-alpine + mysql:8.0.16 | 86.98 + 0 | 50.15 + 443 | 137.1 + 443 | 2.3 MB + 7 B |
| MCS | arm32v7/python :2-alpine | 52.99 | 39.2 | 92.2 | 1.55 MB |
| PD | | | 39.37 | 92.4 | 1.54 MB |
| BS | | | 40.72 | 93.7 | 2.42 MB |
| MRS | | | 39.15 | 92.1 | 1.55 MB |
| UG | | | 20.9 | 73.9 | 776 kB |

Alpine based image may need few compile time tools and libraries to build an application which may not be used to run an application. In our implementation, we neglect these compile time dependencies (C tools and libraries required by Python) but adding just runtime dependencies.

## B. Performance

The performance factor is measured based on the time taken at various deployment phases. Following, we discuss each of them briefly.

*1) Service deployment:* Service deployment time is the total required time to build a service into a Docker system. It is one of important performance matrix that has been measured at the development phase. The overall service deployment time has been presented in the Fig. 6.

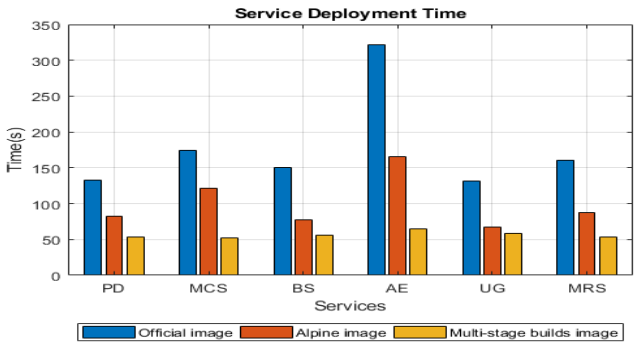


Fig. 6: Service deployment time.

Here, we measured the effect of images built on top of official, alpine; and multi-stage-builds based images in terms of deployment time when there is no base images inside the Docker system. In our implementation, the time taken by PD, MCS, BS, AE, UG; and MRS are 133s, 175s, 151s, 322s, 131s; and 161s respectively when official images introduced to these services. For alpine-based image, these services took time as 82s, 121s, 78s, 165s, 67s; and 88s respectively, whereas they took 54s, 52s, 56s, 65s, 59s and 53s respectively for multi-stage-builds images. During implementation, on an average,

40-50% deployment time has been reduced when there was a base image inside the Docker system. In the implementation, MySQL database were used for data persistence as a dependency service to AE and the service took 60-80s on an average at the deployment stage.

*2) Service initiation:* Service initiation time is the total required time to initiate a service into the Docker ecosystem. It is another important factor which is measured for all the three cases introduced at the development phase. Fig. 7 shows the overall service initiation time for all six nanoservices under three separate cases in terms of images.

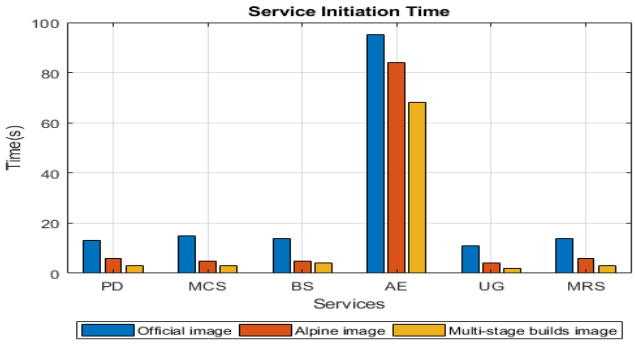


Fig. 7: Service initiation time.

For the use of official base images, PD, MCS, BS, AE, UG; and MRS have taken the time as 13s, 15s, 14s, 95s, 11s; and 14s respectively to initialize the nanoservices. On the other hand, for alpine image images, the time taken is 6s, 5s, 5s, 84s, 4s; and 6s respectively, and whereas for the images created with multi-stage-builds feature took 2-4s to start the nanoservices. These measurement were taken when there is no MySQL image inside the Docker system which usually takes 60-80s to build it as mentioned earlier. In our case, MySQL service took 12s, 7s; and 4s to initiate it on the Docker system for official, alpine; and multi-stage-builds feature based images respectively.

*3) Presence detection time:* In this paper, presence detection is the first nanoservice defined for the use case to detect a participant by using Passive Infrared Radio (PIR) motion sensor. We have measured the average required time in this detection process.

TABLE V: Presence detection and BLE authentication time

| | **Inspections** | **Mean(s)** | **Variance(s)** |
|---|---|---|---|
| **Presence detection** | 20 | 1.95 | 0.447 |
| **BLE authentication** | 20 | 2.75 | 0.487 |

In order to have more precise evaluations, we have performed the measurement about 20 times to analyze the average for the detection of participants. All these different instances took 1s (minimum) and 3s (maximum) during our measurement. On an average, all these took around 1.95s to detect a participant into the range of motion sensor. We also analyzed the variance of these results and it was around 0.447s as shown in Table V. This average time of detection fully depends on the sensing capacity of the PIR motion sensor.

*4) Authentication time and accuracy:* In the implementation, the autonomous smart environment required to authenticate a participant with BLE-ID along with WiFi-CSI or surveillance camera. Here, the smart system recognizes a participant by a BLE key carried by the participant. We measured the average authentication time at three separate inspections.

Table V presents the mean authentication times required by the presence detection and BLE scanner and their variance, based on 20 inspections. We achieved 100% presence detection and BLE scanning accuracy in all inspections. In our implementation, the authentication accomplished in two separate processes that includes BLE scanning and API authentication. Here, the BLE scanning took place in between MCS and BS nanoservices (c) whereas API authentication took place in between MCS and AE nanoservices (d) that mentioned in Fig. 3. In the experiment, the total time to authenticate a participant depends on the physical distance between the BLE authentication server and the BLE key; the quality of the packet advertising by BLE device, and the overall internal processing speed of the MCS-BS and the MCS-AE. Any of these can affect the authentication process and hence increase the total authentication time.

The corridor used for the measurement was about 14m long from the entrance to the meeting room where a user can cover the required distance in 10s with an average walking speed. Therefore, a participant can reach at meeting from entrance within 15s (2s for PD, 3s for BS + AE and 10s for walking).

## VI. DISCUSSION AND FUTURE WORK

This paper presents the feasibility study of deploying virtualized nanoservices at the local level of IoT edge networks. In the feasibility study, we have implemented a prototype based on Docker containers orchestrated by Docker Swarm, and optimized its operation by using Alpine base image and multi-stage-builds. By using Alpine base image combined with multi-stage-builds, we have been able to reduce the base image size from several hundreds of MBs to few tens of MBs while keeping the runtime consumption on feasible level. As a result, the service deployment and initiation times per nanoservice were reduced from several minutes to less than a minute and service initiation times ware squeezed in most of the cases from tens of seconds to only a few seconds.

In this paper, we successfully demonstrated the feasibility of static deployment of nanoservices for our use case, but we neither considered the impact of several running services at the host machines nor took into account other important features, such as scalability to larger setups and dynamic deployment of nanoservices. As future work, it would be therefore interesting to implement more dynamic use cases with larger setups consisting of multiple local sites and several services per device, with users moving across these sites.

Furthermore, the performance evaluations of our paper have been measured at Docker that shares same hardware resources with the host machine, such as RAM, CPU and storage. Among all the evaluations, multi-stage-builds based images show the best performance, because it includes only run-time

dependencies. In addition, most of the run-able sizes (6th column on Table II, III and IV) of images are measured to be highly varying. The reason for it is not much explored in depth during this implementation and we kept this as a part of our future work.

## VII. CONCLUSION

IoT is considered as a driving force behind the vision of ubiquitous and pervasive computing to ensure the availability of digital services anywhere and anytime. Edge computing has been recently introduced as an intermediate layer between local and public networks and provide vital low-latency based services by deploying computational tasks near the end-devices. However, there are still many challenges to address, such as vulnerability to access network problems and the high load inflicted to access networks and MEC servers, being a particular issue in massive-scale IoT use cases, where numerous sensors may produce high amounts of data, or where critical system functionalities must be ensured also during access network problems. Therefore, bringing some edge functions to the local level would be beneficial from the viewpoint of reliability and scalability. In this direction, our work provides significant contribution in terms of implementing a simple PoC deploying virtualized nanoservices at the local level of IoT edge networks and showing the feasibility of such scenario. With respect to hardware requirements and deployment times, the evaluations clearly indicate the feasibility of the nanoservice-based local virtualized service model. The future work includes e.g. expanding the study towards larger setups with dynamic deployment of nanoservices.

## ACKNOWLEDGMENT

This work was supported by Academy of Finland, under the projects Industrial Edge and WiFiUS Massive IoT; the Technology Industries of Finland Centennial Foundation and Jane and Aatos Erkko Foundation, under the project MEC-AI; and 6Genesis Flagship programme (grant no.318927).

## REFERENCES

[1] E. Harjula, P. Karhula, J. Islam, T. Leppänen, A. Manzoor, M. Liyanage, J. Chauhan, T. Kumar, I. Ahmad, and M. Ylianttila, "Decentralized iot edge nanoservice architecture for future gadget-free computing," *IEEE Access*, vol. 7, pp. 119 856–119 872, 2019.

[2] H. U. Rehman, M. Asif, and M. Ahmad, "Future applications and research challenges of iot," in *2017 International Conference on Information and Communication Technologies (ICICT)*. IEEE, 2017, pp. 68–74.

[3] J. aikio, v. pentikinen, j. hiki, j. hkkil, a. colley, on the road todigital paradise: The naked approach (2016). [Online]. Available: http://www.nakedapproach.fi/

[4] I. Ahmad, T. Kumar, M. Liyanage, M. Ylianttila, T. Koskela, T. Braysy, A. Anttonen, V. Pentikinen, J.-P. Soininen, and J. Huusko, "Towards gadget-free internet services: A roadmap of the naked world," *Telematics and Informatics*, vol. 35, no. 1, pp. 82 – 92, 2018.

[5] J. Pan and J. McElhannon, "Future edge cloud and edge computing for internet of things applications," *IEEE Internet of Things Journal*, vol. 5, no. 1, pp. 439–449, 2017.

[6] C. Stergiou, K. E. Psannis, B.-G. Kim, and B. Gupta, "Secure integration of iot and cloud computing," *Future Generation Computer Systems*, vol. 78, pp. 964–975, 2018.

[7] A. Botta, W. De Donato, V. Persico, and A. Pescapé, "Integration of cloud computing and internet of things: a survey," *Future generation computer systems*, vol. 56, pp. 684–700, 2016.

[8] T. Kumar, P. Porambage, I. Ahmad, M. Liyanage, E. Harjula, and M. Ylianttila, "Securing gadget-free digital services," *Computer*, vol. 51, no. 11, pp. 66–77, Nov 2018.

[9] D. Celesti, D. Mulfari, M. Fazio, M. Villari, and A. Puliafito, "Exploring container virtualization in iot clouds," in *2016 IEEE International Conference on Smart Computing*, 2016, pp. 1–6.

[10] F. Bonomi, R. Milito, P. Natarajan, and J. Zhu, *Fog Computing: A Platform for Internet of Things and Analytics*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 169–186.

[11] "Mobile edge computing a key technology towards 5G," http://www.etsi.org/images/files/ETSIWhitePapers/etsi_wp11_mec_a_key_technology_towards_5g.pdf, 2015, eTSI White Paper No. 11, accessed: 10-10-2016.

[12] A. Yousefpour, C. Fung, T. Nguyen, K. Kadiyala, F. Jalali, A. Niakanlahiji, J. Kong, and J. P. Jue, "All one needs to know about fog computing and related edge computing paradigms: A complete survey," *Journal of Systems Architecture*, no. February, 2019.

[13] K. Dolui and S. K. Datta, "Comparison of edge computing implementations: Fog computing, cloudlet and mobile edge computing," in *2017 Global Internet of Things Summit (GIoTS)*. IEEE, 2017, pp. 1–6.

[14] K. Bilal, O. Khalid, A. Erbad, and S. U. Khan, "Potentials, trends, and prospects in edge technologies: Fog, cloudlet, mobile edge, and micro data centers," *Computer Networks*, vol. 130, pp. 94–120, 2018.

[15] C. Esposito, A. Castiglione, and K. R. Choo, "Challenges in delivering software in the cloud as microservices," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 10–14, Sep. 2016.

[16] M. Villamizar, O. Garcs, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil, "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud," in *2015 10th Computing Colombian Conference (10CCC)*, Sep. 2015, pp. 583–590.

[17] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar, and M. Steinder, "Performance evaluation of microservices architectures using containers," in *2015 IEEE 14th International Symposium on Network Computing and Applications*, Sept 2015, pp. 27–34.

[18] C. M. Aderaldo, N. C. Mendonça, C. Pahl, and P. Jamshidi, "Benchmark requirements for microservices architecture research," in *Proceedings of the 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering*, ser. ECASE '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 8–13.

[19] L. Cominardi, O. I. Abdullaziz, K. Antevski, S. B. Chundrigar, R. Gdowski, P.-H. Kuo, A. Mourad, L.-H. Yen, and A. Zabala, "Opportunities and challenges of joint edge and fog orchestration," in *2018 IEEE Wireless Communications and Networking Conference Workshops (WCNCW)*. IEEE, 2018, pp. 344–349.

[20] A. Hegyi, H. Flinck, I. Ketyko, P. Kuure, C. Nemes, and L. Pinter, "Application orchestration in mobile edge cloud: placing of iot applications to the edge," in *2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS* W)*. IEEE, 2016, pp. 230–235.

[21] T. Kumar, A. Braeken, M. Liyanage, and M. Ylianttila, "Identity privacy preserving biometric based authentication scheme for naked healthcare environment," in *2017 IEEE International Conference on Communications (ICC)*. IEEE, 2017, pp. 1–7.

[22] T. Ojala, J. Orajärvi, K. Puhakka, I. Heikkinen, and J. Heikka, "panoulu: Triple helix driven municipal wireless network providing open and free internet access," in *Proceedings of the 5th International Conference on Communities and Technologies*. ACM, 2011, pp. 118–127.

[23] G. Tanganelli, C. Vallati, and E. Mingozzi, "Coapthon: Easy development of coap-based iot applications with python," in *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*. IEEE, 2015, pp. 63–68.

[24] M. Wasilak, Chrysn, P. Berndt, R. Nowakowski, and J. Kinestral. (2018, December) txthings - coap library for twisted framework. [Online]. Available: https://github.com/mwasilak/txThings

[25] Mkn and Tripleee. (2018, August) Running bash commands in python. [Online]. Available: https://stackoverflow.com/questions/4256107/running-bash-commands-inpython/51950538#51950538

[26] R. Sola. (2016, September) Coap: Get started with iot protocols. [Online]. Available: https://opensourceforu.com/2016/09/coap-get-started-with-iot-protocols/

[27] Docker. (2019, Oct) Use overlay networks. [Online]. Available: https://docs.docker.com/network/overlay/

[28] D. Docs. (2019, Jun) Use multi-stage builds. [Online]. Available: https://docs.docker.com/develop/develop-images/multistage-build/