This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3102867, IEEE Access

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.DOI

Resource-aware Dynamic Service Deployment for Local IoT Edge Computing: Healthcare Use Case

JOHIRUL ISLAM, TANESH KUMAR, IVANA KOVACEVIC, ERKKI HARJULA
Centre for Wireless Communication, University of Oulu, Finland

Corresponding author: Johirul Islam (e-mail: johirul.islam@oulu.fi).

This research work was supported by Academy of Finland, under the projects: DigiHealth and 6G Flagship Projects (grants 326291, 318927).

ABSTRACT Edge computing is a novel computing paradigm moving server resources closer to end-devices. In the context of IoT, Edge computing is a centric technology for enabling reliable, context-aware and low-latency services for several application areas such as smart healthcare, smart industry and smart cities. In our previous work, we have proposed a three-tier IoT Edge architecture and a virtual decentralized service platform based on lightweight microservices, called nanoservices, running on it. Together, these proposals form a basis for virtualizing the available local computational capacity and utilizing it to provide localized resource-efficient IoT services based on the applications' need. Furthermore, locally deployed functions are resilient to access network problems and can limit the propagation of sensitive user data for improved privacy. In this paper, we propose an automatic service and resource discovery mechanism for efficient on-the-fly deployment of nanoservices on local IoT nodes. As use case, we have selected a healthcare remote monitoring scenario, which requires high service reliability and availability in a highly dynamic environment. Based on the selected use case, we propose a real-world prototype implementation of the proposed mechanism on Raspberry Pi platform. We evaluate the performance and resource-efficiency of the proposed resource matching function with two alternative deployment approaches: containerized and non-containerized deployment. The results show that the containerized deployment is more resource-efficient, while the resource discovery and matching process takes approximately 6-17 seconds, where containerization adds only 1–1.5 seconds. This can be considered a feasible price for streamlined service management, scalability, resource-efficiency and fault-tolerance.

INDEX TERMS IoT, edge computing, distributed computing, virtualization, resource discovery, microservices, nanoservices.

I. INTRODUCTION

DURING the past decade, microservice architectures (MSA) [1], [2] have superseded the monolithic service architectures as the foundation of modern cloud computing systems. MSA decomposes monolithic applications into smaller independent services or processes that can be distributed in the cloud computing infrastructure, which ensures the performance optimization of applications and the whole system in terms of flexibility, scalability, and maintainability [2], [3]. Some well-known and promising example enterprises adopting MSA architectures are Netflix and Amazon [1]. Recently, Edge computing has extended the cloud architecture by bringing parts of the microservice architecture from data centers to edge servers, closer to the end user and IoT devices [4]. Typically, these edge servers reside within the access network infrastructure, e.g. co-located with RAN base stations [5].

In many IoT cases, the connection between the IoT devices and the access network is intermittent and/or low in capacity. As a result, the unstable data path between the sensor and actuator devices and the service components causes problems in the service availability. This is highly problematic with mission-critical tasks, such as health monitoring or industry process control [6], [7]. For the delay-sensitive IoT applications, such as continuous remote monitoring of patients in healthcare domain, it is important

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3102867, IEEE Access

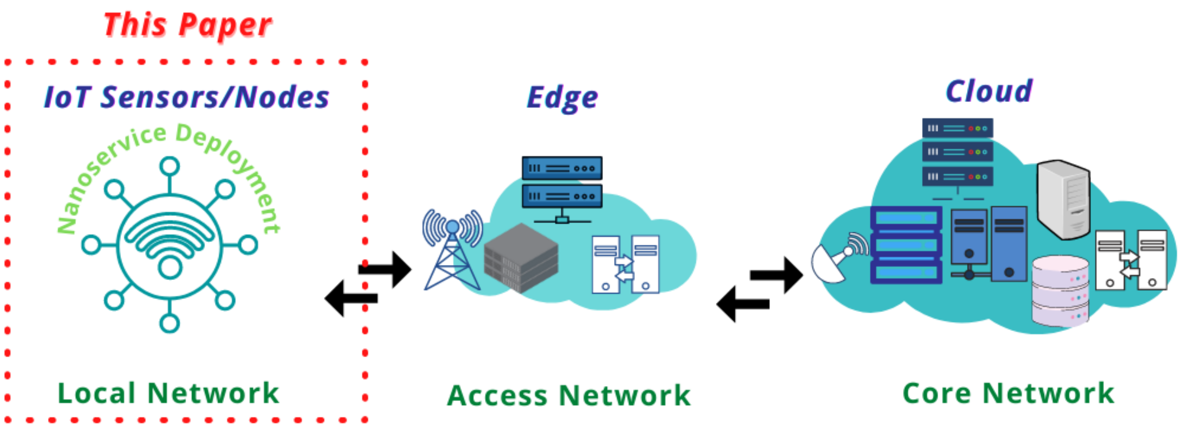


Figure 1: Focus of the paper.

that the most critical functions can be provided locally if needed [8]. In situations where local monitoring devices and sensors are disconnected from the rest of the network, it is highly important to ensure the availability of enough capabilities and resources needed to execute the critical tasks using local computing. Bringing edge computing to local level by deployment of microservices to local nodes with sufficient computational capacity is a prominent way to address the problem described above. However, as IoT nodes are typically capacity-constrained, there is a need to specify a lightweight version of microservices to enable their deployment on the IoT nodes. Recently, the authors of this paper have developed a concept of nanoservice [9], [10], which can be seen as a miniature version of a microservice, specifically dedicated to perform a single-purpose granular operations, such as periodically reading sensor data and sending it further to a gateway node, performing a computational task when requested, accessing a local database, or controlling an actuator based on a request.

In our proposed decentralized nanoservice architecture, the nanoservices are mainly designed based on request-response paradigm that is traditionally implemented in a centralized manner. In the architecture, a number of decentralized components, operated by various stakeholders, need to be seamlessly integrated to provide services based on the current need in a dynamic operation environment consisting of capacity-constrained, in many cases also mobile, devices.

This operational environment generates high requirements for resource discovery and matching the discovered resources to the needs of the system and the applications. Deploying nanoservices at non-uniform, resource constrained local IoT networks is much more complex since the service requirements are highly dynamic. Therefore, the traditional resource discovery and orchestration mechanisms used in current microservice architectures are not suitable in resource-limited, highly dynamic and decentralized local environments [11]. Hence, this paper extends our previous prototype implementation of the nanoservice architecture [10], by developing further its orchestration mechanism to fulfil the requirements of highly dynamic and decentralized operation environment.

In summary this paper provides the following key contributions:

- We develop further the concept of dynamic nanoservice deployment mechanism to fulfil the needs of service requirements in a non-uniform, resource constrained, and highly dynamic local IoT environment.
- We propose a nanoservice and resource discovery and matching algorithm to enable dynamic resource allocation, and develop our prototype implementation further by integrating dynamic resource discovery and matching functionalities in it.
- We provide a performance evaluation of the proposed dynamic nanoservice deployment and analyze the feasibility of the prototype by comparing non-containerized and containerized deployment scenarios.

The rest of the paper is organized as follows. Section II elaborates the background, on-going work and relevant concepts. We describe the selected use case in section III. Section IV explains the dynamic nanoservice deployment mechanism and section V provide the required system configuration for the deployment. We perform the Proof-of-Concepts (PoCs) implementation based on the defined use case in section VI. Evaluation results are presented in section VII. Finally, we provide discussion and future directions in section VIII and conclude the paper in the section IX.

II. BACKGROUND AND RELATED WORK

A. CLOUD-EDGE CONTINUUM

Cloud computing refers to the delivery of different electronic services through the Internet. In cloud computing, the functional service components, such as data storage, servers, databases, software platforms and applications typically reside in large centralized computing clusters, called data centers. Cloud computing has been widely used as the brains of many IoT based applications as it provides practically unlimited computational, processing and storage capabilities

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3102867, IEEE Access

and global access. In addition to these undeniable benefits, cloud computing also faces many challenges, such as high latency and dependency on always-on network connections.

Edge computing is a concept developed to address these challenges by bringing cloud computing resources near the local devices. It enables features previously not available for cloud computing, such as low-latency communication, but also helps enhancing privacy protection by providing means to process sensitive data close to its source and therefore reduce the need to propagate it to public networks. It also improves scalability and resource-efficiency with possibility for data preprocessing and reduction near the source of data [4], [12]. Multi-access Edge computing (MEC) is a standard solution proposed by European Telecommunications Standards Institute (ETSI) for enabling faster data processing, analytics, storing, decision making and local offloading for the next generation 5G and beyond systems.

In local edge computing [13], the key idea is to push processing even further to the network edge, involving the sensor and actuator devices in processing computational tasks. Local edge is also known as extreme edge, and local edge computing is also known as mist computing [13]. Local edge computing paradigm implies that some of the needed communication can be performed at the local devices to reduce the burden at access and core networks [14]. This also ensures the local availability of the services for the end devices, even if there is no connection available/established with the higher tiers. The cloud - edge continuum, consisting of the three tiers (cloud computing (core), edge computing (access) and local edge computing (local)) is illustrated in Fig. 1.

B. MONOLITHIC VS MICROSERVICES VS NANOSERVICES

Monolithic service architecture was used in the first-generation cloud systems [15]. In monolithic service architecture, various services and functions are encapsulated into a single functional unit. The drawback of the monolithic architecture is that a developer must build and deploy an new version of the whole unit, when updates - even minor ones - are needed. As a consequence, the management of such architecture is resource consuming and leads to limited scalability, maintainability and feasibility [16].

Microservice architecture (MSA) paradigm has emerged during the past decade to address the above-mentioned problems of the monolithic service architectures. Microservice architecture allows the developer to build, manage and update an application easily as separate independent units [16]. It brings a number of advantages in terms of flexibility and scalability. A few studies have been performed in the context of deployment of the MSA for various IoT applications based on the edge [17], [18] and mist computing [19]. Microservices are widely deployed in cloud data centers or edge cluster nodes in the last few years. Furthermote, microservices provide the foundation for distributed cloud computing and edge computing by allowing logical and geographical distribution of system components.

As mentioned in the previous subsection, local edge computing allows bringing edge cloud services to local tier. However, since the IoT devices and sensors at the local tier have very limited hardware resources and computational capabilities, most of the legacy cloud microservices are too heavy to be managed on these local devices. Therefore, we can recognize a clear need for a lightweight version of microservices to perform local edge computing. In our previous work, we have proposed a nanoservice-based conceptual service model, 'nanoEdge' [9], [10], where local functions are virtualized as nanoservices - lightweight versions of microservices - that integrate local functions in the cloud-edge continuum. The proposed nanoservice architecture allows dynamic deployment of the local services according to the needs of the specific functions.

C. IOT EDGE-CLOUD MODELS

In [20], [21], we have introduced three different architectural models for IoT-based applications. Fig. 2 depicts these three models. The first one is the 'traditional Cloud-IoT' model, where the sensors/devices at local networks sense and gather the needed information and send most of the data to the centralized cloud for the required processing, management and storage purposes. However, this model suffers from long latency and high network load. The centralized cloud also forms a single point of failure. Therefore, it cannot fulfil the delay and mission-critical application requirements that are typical in e.g., healthcare applications.

To address these challenges, the second model '2-tier Edge IoT' model integrates edge computing to the Cloud-IoT architecture as an intermediate tier located at the access network between the local and cloud tiers, bringing a part of the cloud computing infrastructure closer to the end-users or devices to address the challenges related to latency, efficiency and reliability.

The third model, '3-tier Edge-IoT' model integrates the local devices as a part of the cloud computing infrastructure. The local tier processing is highly crucial for mission-critical applications that require continuous operation in every situation, including the network outage. It also helps improving resource-efficiency by allowing data reduction functions at local tier to save network capacity in data-intensive applications. Furthermore, local processing of data can help improving data privacy and security by reducing the need to deliver sensitive raw data further from the local tier.

D. LIGHTWEIGHT VIRTUALIZATION TECHNOLOGIES

Virtualization refers to a technology that generates a virtual instance of different parts of a computer system, which can be accessed through an interface. Since the lower functional layers are abstracted behind an interface, the application or service developers do not need to consider the complexity of the underlying systems. Virtualization also

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3102867, IEEE Access

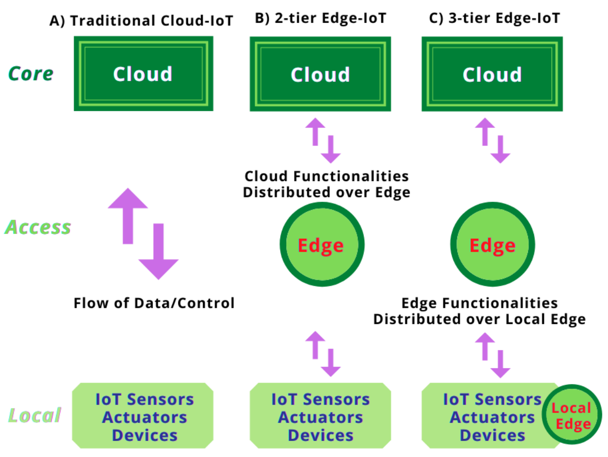

A) Traditional Cloud-IoT
B) 2-tier Edge-IoT
C) 3-tier Edge-IoT
Core
Cloud
Cloud
Cloud
Cloud Functionalities Distributed over Edge
Access
Edge
Edge
Flow of Data/Control
Edge Functionalities Distributed over Local Edge
Local
IoT Sensors Actuators Devices
IoT Sensors Actuators Devices
IoT Sensors Actuators Devices
Local Edge

Figure 2: IoT Edge-Cloud models [9], [10].

allows modifying and updating the underlying technologies without disturbances to users and applications above the virtualization interface.

Hypervisor systems, such as Oracle Virtual Box and VMware, is a traditional and widely adopted virtualization technology. It creates an isolated environment that separates the operating system from the underlying real physical hardware to wrap and run one or many applications. As a drawback, this virtualization technique is resource-hungry and also introduces performance degradation when run on the host machine, because each of the virtual machine is required to run on the separate/individual guest operating system.

Container-based virtualization, on the other hand, shares the OS kernel in addition to hardware resources (e.g CPU, memory, storage etc) to run an application on top of the host and separates the virtualized instances by their processes. Container runtime engine, such as Docker, LXC, LXD or etc is required to run a containerized application. Among these, docker technology is seen crucial for empowering the microservices architecture mainly due to its low overhead, faster boot-up time, and less memory requirements. To manage these containers, Docker may cooperate various orchestration engines, such as Docker Swarm, Kubernetes, Apache Mesos, and etc.

E. NANOEDGE CONCEPT

The NanoEdge concept takes the conventional Edge Computing (MEC) vision even a step ahead by deploying some of the edge services to the resource constrained local IoT nodes. Nanoservices utilizes local computational resources for deploying parts of cloud services in proximity of data sources and/or service consumers. The nanoservices have certain requirements in terms of node hardware capacity and capabilities. For example, a oxygen saturation level detecting nanoservice requires a pulse oximeter attached to the computational node running it, and sufficient computational capacity to run the function.

In our previous work [9], [10], we have developed a PoCs implementation of a virtualized nanoservice architecture for local IoT edge networks. We have analyzed the feasibility of deploying some edge services from the higher tiers to the local tier, by utilizing the local hardware as the computing platform. The prototype was implemented using docker and docker swarm technologies that enabled the deployment and orchestration of nanoservices on low-capacity IoT nodes. At the first phase, the composition and deployment of these nanoservices were done in a static manner. In such deployment, the needed service requirements are checked manually before the actual deployment. In this paper, the aim is to further extend this prototype implementation for dynamic deployment of the nanoservices at the local IoT edge networks.

III. USE CASE: COVID-19 PATIENT MONITORING

As use case, we consider a digital healthcare scenario, where a patient has been diagnosed with a contagious COVID-19 disease. To minimize the risk of spreading this dangerous and highly contagious disease, it is beneficial to treat patients at home as long as it is possible. In this scenario, advanced remote monitoring using modern IoT equipment is needed to determine if and when an infected person should be hospitalized. In this use case, a home treatment patient is given a medical device or wearables, e.g. smart/sport watches, smart clothing [22], or skin-mounted biosensors [23] capable of measuring a patient's health data.

When the patient is at a home quarantine, the sensor information is analyzed locally to avoid extra load on hospital systems, which is crucial in widespread pandemics, where tens of thousands of patients may be taken care by a single hospital. If a patient's condition gets worse, the automatic analysis of the monitored data can alert the personnel, who, if needed, can make the decision of hospitalizing the patient. In the scenario, the need for remote monitoring continues in hospital (with increased intensity), since face-to-face care needs to be minimized to avoid exposing the medical personnel to the disease.

For simplicity, we have divided the remote COVID-19 patient monitoring system into four sub-tasks. These four sub-tasks are indicted with alphabetic notation from **A** to **D** and highlighted with red circles as shown in Figure 3. The initial sub-task is related to the ***(A)*** *data acquisition* where different types of medical sensors/devices are used to monitor the current health status of a patient, e.g. the oxygen saturation level (SpO_2) of the blood, heart rate (HR) and body temperature (BoT). When the patient is hospitalized, actuators such as oxygen controller (OC) dosing oxygen for the patient through an oxygen mask and a screen (i.e. monitor) are used for keeping the oxygen in certain required level and for displaying the patient status. All the nanoservices and the devices used in this use case are shown in Table 1.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3102867, IEEE Access

Table 1: Nanoservices, devices and alert thresholds in our case.

Sub-Task	Nanoservices	Devices (sensors / actuator)	Threshold
A	Oxygen Level (SpO_2)	Pulse Oximeter [24]	$< 90\%$
	Heart Rate (HR)		rate < 40 or > 120 bpm, variability < 20 ms
	Body Temperature (BoT)	Armband using MCP9808 (BLE Temperature) [25]	$> 40^o$ C
	Oxygen Controller (OC)	Oxygen mask with DC motor & Oxygen cylinder [26]	$SpO_2 < 90\%$
B	Alerts	Email, SMS, Call	–
C	Data Display (DD)	Any screen / monitor	–
D	Advanced nanoservices for continuous monitoring	Any advanced medical services/facilities e.g. ECG	Defined as per specific services

The next sub-task ***(B)*** *alerts activation* checks if the measured health values go beyond the threshold limits for longer periods, e.g. 30 minutes. If this happens, an alert nanoservice will be activated to send an alert notifying the medical staff about the patient status, through e.g. an SMS or email. The medical staff can now start ***(C)*** *remote data monitoring* of the patient. If the data shows signs of severe symptoms, the medical staff can fetch the patient with an ambulance to the hospital and will arrange advanced medical services in the last sub-task ***(D)*** *Hospital monitoring services*.

In this use case scenario, it is crucial that the system remains functional even when the quality of network connectivity is occasionally low or even completely down. In these situations, the continuous sensor data analysis can be ensured using local sensors/devices computational capacity. Therefore, e.g. alerts generated during network outages can be sent to the hospital system when the network connection is restored. At the same time, local analysis relieves the network load and computational load on the hospital system, which can be a significant factor to ensure the online healthcare system functionality when there is a risk of system overload, e.g. during severe pandemics such as COVID-19.

IV. DYNAMIC NANOSERVICE DEPLOYMENT

In this paper, we propose a dynamic orchestration solution for our nanoEdge concept [9]. In its first PoC implementation [10], the node capabilities and the service requirements needed to be manually checked before the deployment. The mechanism proposed in this paper, makes the PoC follow better the original nanoEdge concept, by providing an automatic nanoservice deployment for dynamically changing environments, based on node capabilities and service requirements. In the following subsections, we describe the cluster formation and service deployment of our proposal in more detail.

A. CLUSTER FORMATION

In the nanoEdge concept, the nanoservices are deployed to the cluster of local nodes with different capabilities. We distinguish two type of nodes, manager and worker nodes. The manager nodes are responsible for maintaining the cluster and deploying services to the worker nodes, whereas the worker nodes are executing those services. If a cluster is made of multiple manager nodes, one is set to active. Active manager node initiates the cluster, while other manager nodes are set to reachable status and used as the backup manager nodes. The active manager node may demote from manager role or leave a the cluster at anytime. When the active manager node is demoted from the manager role to the worker role or left the cluster, then one of the manager nodes is activated as a new leader from the other remaining manager nodes. By decentralizing the cluster management this way, we can avoid reliability problems that would arise from a single point of failure. The whole orchestration process is done by the orchestration

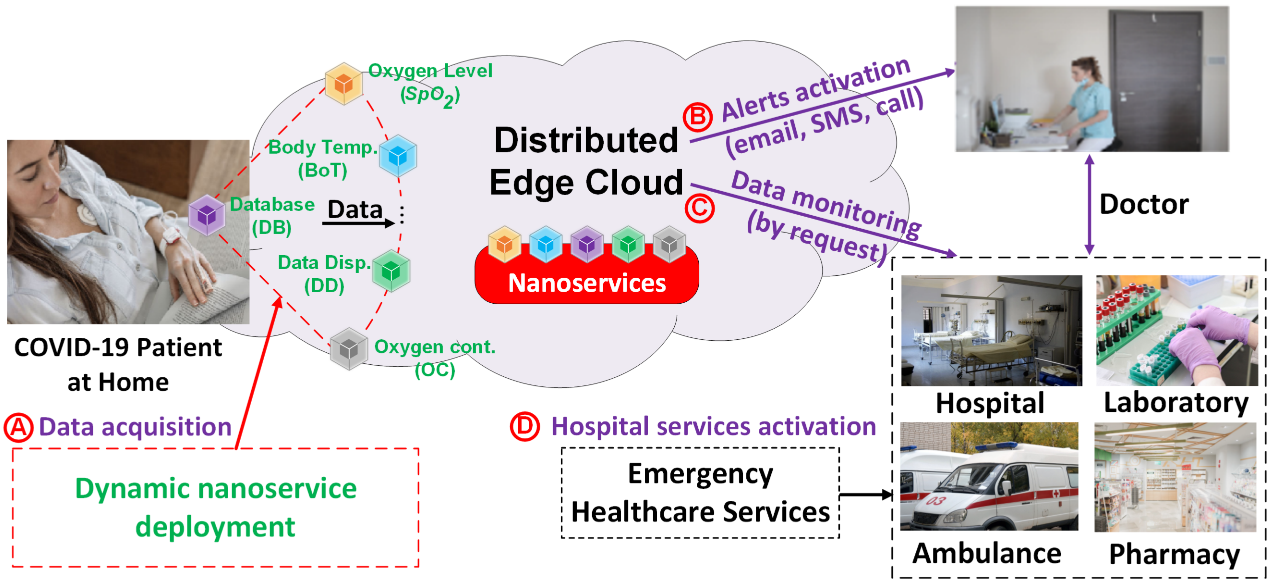


Figure 3: COVID-19 patient remote monitoring use case.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

engine.

In our proposed model, as shown in Fig. 4, the active manager node distribute the nanoservices to the associated worker nodes based on the service requirements. We denote the set of manager nodes as $\mathcal{M} = \{M_1, ...M_j, ..., M_J\}$. It maintains a list of nanoservices in the service repository that should be deployed: $\mathcal{S} = \{S_1, S_2, \ldots S_k \ldots S_K\}$. Each nanoservice is characterised with a *service profile* P_k that specifies the type and amount of resources necessary for the service to be executed at the worker node. The set of worker nodes is denoted by $\mathcal{W} = \{W_1, ...W_i, ..., W_I\}$. Similarly, each worker node is characterised with a *resource profile* P_i that specifies the amount and type of resources available for execution of nanoservices.

Table 2: Symbolic representation

Notation	Meaning	Notation	Meaning
$\mathcal{M}$	Set of manager nodes	t_k / t_i	Node type (sensor / actuator)
$\mathcal{W}, i$	Set and index of worker nodes	c_k / c_i	CPU (number of cores, etc.)
$\mathcal{S}, k$	Set and index of nanoservices	m_k / m_i	Memory (RAM)
S_k	Nanoservice k	s_k /s_i	Storage (SD / SSD / HDD)
W_i	Worker node i	b_k / b_i	Operating power (AC / DC)
P_i	Resource profile of W_i	n_k / n_i	Network (WiFi, Bluetooth)
P_k	Service profile of S_k	–	–

Service S_k has a service profile $P_k = \{T_k, c_k, m_k, b_k, n_k, s_k\}$ where T is the type of node, i.e. the set of node capabilities $T = \{t_1, t_2..\}$ necessary to execute the service. Node capability t denotes the type of a node, i.e. a sensor, an actuator or a computational resource. Parameter c represents the number of CPU cores as an integer value; m represents the memory requirement in MB; b represents the battery requirement, i.e. whether the node is operating with DC mode or AC power; n represents the type of the network connection required (WiFi, Bluetooth, etc.); and s represents the storage requirement in MB. This is a general service profile, so if the service does not have the requirement for certain worker node property, its value is set to 0. Worker node W_i has a node profile in the same format $P_i = \{t_i, c_i, m_i, b_i, n_i, s_i\}$, representing the available resources at the time of allocation. The notation is summarized in Table 2.

B. NANOSERVICE DEPLOYMENT

The dynamic nanoservice deployment mechanism is performed in the following five steps, which are also marked with red circles in Fig. 4 .

Step 1: Resource Discovery — At the beginning, all available resource information is gathered to generate a local resource profile P_i at a worker node W_i. The local resource profile includes configuration or properties of a worker node. This step is performed by each worker node right after joining the cluster.

Step 2: Resource Reporting — The worker node W_i initiate resource reporting procedure to store the resource configuration as a resource profile P_i at the active manager node M_j. This step is also performed by each worker node right after the completion of the resource discovery (step 1).

Step 3: Service Discovery — The active manager node discovers the available nanoservices (i.e container images) from a remote service repository (for example, from a Docker Hub or a private registry) after the resource reporting (step 2). In this step, an active manager node generates

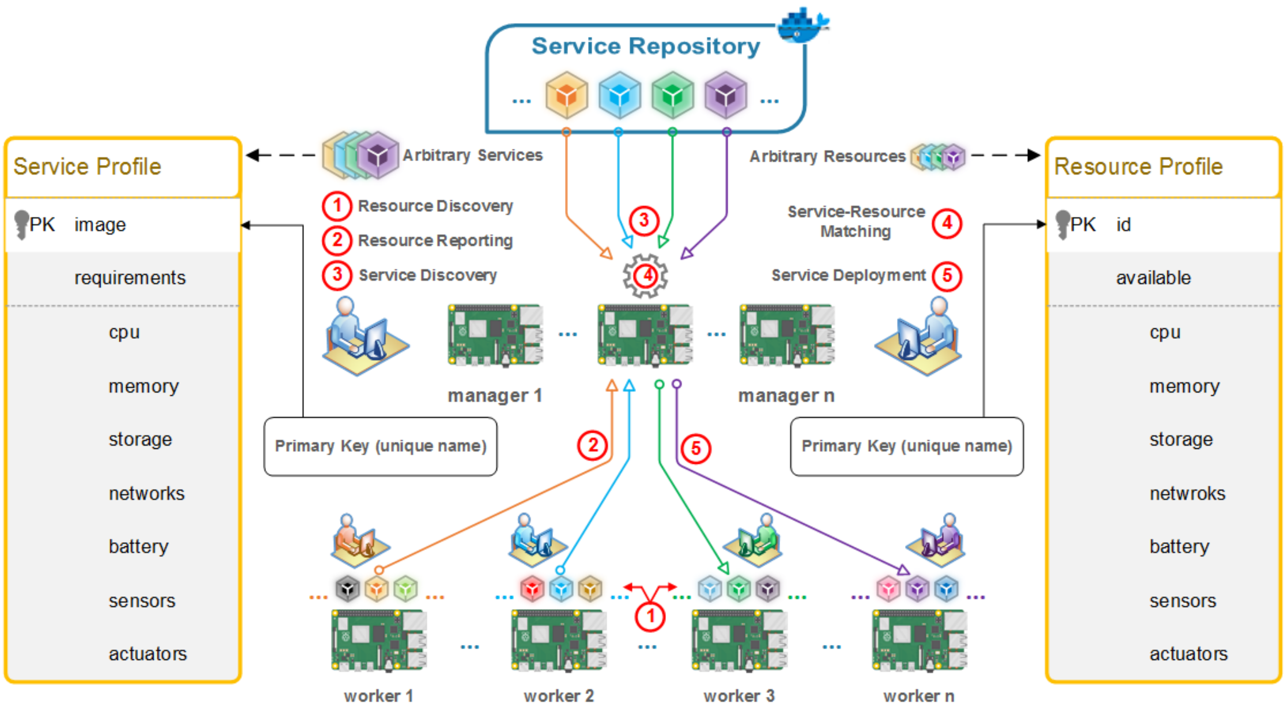


Figure 4: Service deployment process.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3102867, IEEE Access

a list of service profiles $\mathcal{S}$ for each nanoservice located at the remote service repository including all the resource requirements.

Step 4: Service-Resource Matching — Service requirements need to match the worker node capabilities. The active manager node allocates a nanoservice S_k to a suitable worker node W_i. Worker node can execute more than one nanoservice at the time only if it has sufficient capacities. This matching process is a continuous process for all of the nanoservices. The details of this procedure are given in the section IV-C.

Step 5: Service Deployment — The active manager node starts deploying the matched nanoservices S_k (chosen in step 4) to the suitable worker nodes W_i.

C. SERVICE-NODE MATCHING AND RESOURCE ALLOCATION ALGORITHM

In our deployment we use a simple algorithm to select the worker nodes to execute the different nanoservices. The selected active manager node M_j allocates the nanoservices, one by one, to the worker node that fulfill their requirements. Thus, several nanoservices can be executed by one worker node. After each assignment, the available capacities (e.g. remaining storage) of the worker node is updated.The steps of the node-matching and resource allocation algorithm are presented in Algorithm 1:

Algorithm 1 Resource Allocation

```
1:  FOR $k = 1$ to $K$
2:  i=1; $Match = 0$
3:  WHILE $Match = 0$ and $i < I + 1$
4:  IF $t_k \in t_i, c_k \leq c_i, m_k \leq m_i, b_k = b_i, n_k = n_i, s_k \leq s_i$
5:      $Match = 1$; allocate $s_k$ to $W_i$
6:      $P_i = t_i, c_i - c_k, m_i - m_k, b_k, n_k, s_i - s_k$
7:      ELSE i=i+1
8:  END IF
9:  END WHILE
10: END
```

Algorithm 1 creates a resource-service configuration profile by matching the nanoservice requirements over the resources available at the worker nodes. The active manager node sends a deployment request to the selected worker node once the active manager node has the resource-service profile along the list of worker nodes and the associated deployable nanoservices. All these steps and the deployment of nanoservices are done asynchronously. Each worker node continuously informs to the active manager node about the current state of the deployment.

V. SYSTEM CONFIGURATION

In this section we give an overview of the system configuration of our use case scenario, with the purpose to evaluate the proposed dynamic nanoservice deployment solution.

In the deployment, we use two machines (i.e. $J = 2$) as manager nodes ($\mathcal{M}$) and five Raspberry Pis (i.e. $I = 5$) as worker nodes ($\mathcal{W}$). These devices are non-uniform in terms of hardware functionalities and computational capacities, in order to demonstrate the resource-based selection algorithm in action. Depending on the use case scenario, these nodes can host different nanoservices based on types of sensors and actuators having diverse computational resources.

Table 3: Cluster nodes used in the implementation

Notation	Node role	Host OS	Description	Quantity
M_1	Manager	Ubuntu 18.04	Desktop	1
M_2	Manager	Ubuntu 18.04	Laptop	1
W_1 & W_2	Worker	Raspbian 10	RPi 3	2
$W_3 - W_5$	Worker	Raspbian 10	RPi 4	3

Master node M_1 is a laptop computer with 4 GB RAM having Intel Core i3 2 *GHz* CPU, while master node M_2 is a desktop computer with 16 GB RAM having Intel Core i7 1.6 *GHz* CPU. Both are running with Debian based 64-bits Ubuntu 18.04 Operating System (OS). All worker nodes are running with *Debian* based 32-bits *Raspbian GNU/Linux 10 (buster)* OS. The summary of the cluster nodes used in out evaluations is represented in Table 3, while the detailed configuration parameters of the worker nodes are given below.

1) CPU

The capacity of a CPU depends on the number of cores and their clock speed along with other parameters e.g. BogoMIPS that is explained at the end of this section. The limits of these parameters are defined by the device manufacturers. Table 4 shows the CPU properties of the five RPis used as the worker nodes in the cluster.

Table 4: CPU properties

Resources	Model	Cores (c_i)	Speed (MHz) Max	Speed (MHz) Min	BogoMIPS
W_1, W_2	Cortex-A53	4	1200	600	38.40
W_3 - W_5	Cortex-A72		1500		108.00

The number of instructions that could be processed by a computational node can be measured in millions instructions per second (MIPS). The manufacturer of a CPU usually defines the maximum limit of MIPS handled by the CPU. When a CPU crosses this maximum threshold value, it faces a deadlock causing the processor to hang on, and a reboot is required to restore its functionalities. This limit is defined as BogoMIPS (Bogus + MIPS).

2) Memory

The execution of tasks or the number of instructions also depend on the random access memory (RAM) used by a computer system. The overall memory footprint of our worker nodes is shown in Table 5.

The total memory (i.e. first column in Table 5) is fixed by the manufacturer. The memory value for the rest five columns rely on the concurrent running processes at a worker node (W_j). "Buffered/Cached" indicates the memory used by kernel and applications in several I/O operations

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3102867, IEEE Access

Table 5: Memory (RAM) properties (in MB)

Resources	Total (m_i)	Used	Free	Buffered / Cached	Shared	Available
W_1	434	198	126	108	40	178
W_2	926	239	204	481	50	581
W_3	924	280	273	370	94	488
W_4	1939	813	394	731	198	1063
W_5	3776	477	2550	748	226	2936

respectively, whereas ”Shared” indicates the memory used by TMPFS (Temporary File System mounted in some operations).The total memory and and the available memory are calculated as follows:

$$m_t \cong m_u + m_f + m_{bc}$$

$$m_a \cong m_{bc} + m_s$$

Here, the total memory (m_t) is fixed for a RPi while the rest of the memory, i.e used (m_u), free (m_f), buffered / cached (m_{bc}), available (m_a), and shared (m_s) memories are occupied by various processes currently running in the RPis.

3) Storage

We have used an SD card on each RPi that contains a host OS (along with other utilities) to operate the system. An important thing to note is that all the deployable nanoservices are required to occupy certain amount of blank storage to save its data onto the host machine. Table 6 shows the measured SD card properties during the nanoservice deployment.

Table 6: Storage (SD card) properties (in MB)

Resources	File System Type	Size (MB)		
		Total	Used	Available (s_i)
W_1	ext4	12045	8164	3253
W_2		12053	8158	3260
W_3		26569	8186	17011
W_4		26941	9284	16266
W_5		26941	10081	15469

4) Power source

Low energy consumption is vital for resource-constrained IoT nodes since most of them have high battery-life requirements. Therefore, the nanoservice deployment for such nodes should be made in an energy efficient manner. In this implementation, we have used both the DC and AC-powered worker nodes, as shown in Table 7.

Table 7: Battery properties

Resources	Operating (b_i)		State	Percent (%)	Capacity (%)
	AC	DC			
W_1	–	✓	charging	75	68.7831
W_2	–		fully-charged	100	
W_3	–		discharging	95	
W_4, W_5	✓	–	–	–	–

Furthermore, we use in our implementation the information on the charging status of the battery with the remaining battery percentage. Here, we need to take into account that the battery capacity degrades over its life-cycle. Therefore the last column in Table 7 indicates the ratio between the last observed energy (Watt-hour, Wh) and the observed energy (Wh) at the manufacturing lab when the battery was new.

5) Network Connection

The worker nodes have WiFi and Bluetooth network interfaces for communication. Table 8 describes the WiFi network properties. All worker nodes are equipped with 802.11 WiFi network interface, giving 100 Mbps maximum speed and 30 meters of maximum coverage.

Table 8: WiFi network properties

Resources	Standard (n_i)	Max	
		speed *(Mbps)*	coverage *(meters)*
W_1, W_2	IEEE 802.11n	100	30
$W_3 - W_5$	IEEE 802.11ac/n		

With Bluetooth, the maximum transfer unit (*MTU*) is an important factor that defines e.g. the maximum allowed data transfer rate (i.e *receiving:sending*) between two Bluetooth devices. Table 9 summarizes the Bluetooth capacities of our worker nodes on the maximum transfer unit (*MTU*) for asynchronous connection-less link (*ACL*) and synchronous connection-oriented link (*SCO*). *ACL MTU* used to denote the *receiving:sending* rate for non-voiced data packets while *SCO MTU* is used to denote the *receiving:sending*) rate voiced data packets.

Table 9: Bluetooth network properties

Resources	HCI version (n_i)	ACL MTU	SCO MTU
W_1, W_2	4.1 (0x7)	1021:8	64:1
$W_3 - W_5$	5.0 (0x9)		

Bluetooth devices usually connect via either a *USB* (i.e universal serial bus) dongle or a *UART* (i.e universal asynchronous receiver/transmitter). A communication device may use a *USB* dongle if it does not have built-in Bluetooth (or *UART*) support. In our setup, both RPi 3 and RPi 4 use built-in Bluetooth (with *UART*) with host controller interface (or *HCI*) version 4.1 and 5.0. According to column 3 in Table 9, for the non-voice data transfer (i.e in *ACL MTU*) all the worker nodes can receive 1021 packets and send 8 packets in a single transmission. In the case of voice data transmission, the ratio between the receiving and sending data rate is 64:1 for all the worker nodes.

6) Sensors and actuators

In our previous work, we deployed containerized nanoservices into the worker nodes in a static manner [10]. In this paper, we introduce Sensor-Actuator Detection API service to detect the PROM, EPROM or EEPROM memory based HAT (Hardware Attached on Top) and pHAT (partial HAT) based sensors and actuators belonging to our worker nodes by applying DiCola’s technique [27]. The Sensor-Actuator

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3102867, IEEE Access

Detection API service detected the following sensors and actuators from the worker nodes of our setup.

Table 10: Sensors and actuators discovered at cluster nodes

Resources	**Type (t_i)**			**Description**
	Sensor	**Actuator**	**Notation**	
W_1	✓	–	bot	Body Temperature
W_2	–	–	–	–
W_3	–	✓	oc	Oxygen Controller
W_4	✓	–	pom	Pulse Oximeter (SpO_2 & HRV)
W_5	–	✓	dd	Data Display (a screen)

VI. PROOF-OF-CONCEPT IMPLEMENTATION

A. PROPOSED MECHANISM

1) Resource Discovery

During the Resource Discovery, each worker node periodically looks for available number of CPU cores (c_i), memory (m_i), storage (s_i), battery (b_i), networking capacities (n_i) and sensors & actuators (t_i). The overall worker nodes configuration is presented at Table 11.

Table 11: Overall cluster node configuration (P_i)

Resources	**Available at W_i**					
	c_i	m_i (MB)	s_i (MB)	b_i	n_i	t_i
W_1	4	434	3253	DC	BT, WiFi	bot
W_2		926	3260			–
W_3		914	17011			oc
W_4		1939	16266	AC		pom
W_5		3776	15469			dd

According to resource discovery, the body temperature (bot) and pulse oximeter (pom) sensors are belongs to worker node W_1 and W_4 respectively. On the other hand, the oxygen controller (oc) as well as data display (dd) actuators are belongs to worker node W_3 and W_5 respectively. All the discovered sensors and actuators are presented at 7th column in Table 11. Rest of other discovered configurations are summarized at columns 2 to 6. However, after the successful resource discovery, each worker node stores own resource configurations as a local Resource Profile (P_i) in JSON format.

2) Resource Reporting

A CoAP RESTful API nanoservice is implemented with *txThings* framework and deployed at the active manager node. Each worker node maintains its own local resource configuration profile (P_i), which is periodically sent to the active manager node. Different dynamic URL endpoints are introduced to detect and store the resource profile (P_i) at the manager node.

3) Service Discovery

Our active manager node periodically looks for nanoservices from the Docker Hub service repository, specified by us. Table 12 shows the dynamically deployable nanoservices (defined in Table 1) used in our use case scenario.

In our implementation, we put the requirements of a nanoservice in JSON format into the description field of the

Table 12: Target nanoservices to be deployed (P_k)

Nanoservices (S_k)	**Requirements for S_k**					
	c_k	m_k (MB)	s_k (MB)	b_k	n_k	T_k
SpO_2 & HRV	1	< 500	59.3	–	BT	pom
BoT			56.2		WiFi	bot
DB		> 500 & < 1000	72.0			–
OC		< 500	55.3			oc
DD		> 500 & < 1000	73.1			dd
SMS		< 500	56.2			–
email						
call						

specific docker image. The active manager node stores all the nanoservices with their requirements in a JSON database as a Service Profile (P_k).

4) Service-Node Matching

The service-node matching nanoservice selects a nanoservice S_k from $\mathcal{S}$ and allocate it to the suitable worker nodes. As we discuss in Algorithm 1, this process is iterated for all available nanoservices. Table 13 presents the overall matching results with the nanoservices and the corresponding selected nodes.

Table 13: Nanoservices (P_k) with matched resources (P_i)

Nanoservices (P_k)	**Major requirements**	**Selected nodes (W_i)**
SpO_2 & HRV	pom	W_4
BoT	bot	W_1
DB	–	W_1, W_2, W_3, W_4, W_5
OC	oc	W_3
DD	dd	W_5
SMS	–	W_1, W_2, W_3, W_4, W_5
email		
call		

The service-node matching algorithm generates a new JSON database as a Service-Node Profile ($S_k - W_i$), including the combination of nanoservice and a node that satisfies the nanoservice requirements.

5) Service Deployment

At the service deployment phase, the containerized nanoservices, given in Table 13, are deployed into the worker nodes. Here, the active manager node initiates the deployment of nanoservices into the optimal worker nodes, based on the Service-Node Profile ($S_k - W_i$).

B. NANOSERVICE IMPLEMENTATION

During the PoC implementation, we developed the required nanoservices for the previously mentioned steps as shown in Table 14. For example, at the step 1, the sensor-actuator detection engine (X_{11}) nanoservice is developed to detect the type of a sensor or an actuator used in the cluster node. This dedicated CoAP RESTful API nanoservice is made with *python*, *txThings* [28] and *tinydb*. Furthermore, the resource discovery engine (X_{12}) nanoservice is developed to detect all available resources that belongs to a cluster node. *upower*, *lshw*, *bluez* and *jq* are used to build the X_{12} nanoservice.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3102867, IEEE Access

At step 2, the *libcoap* is used at each worker node to create resource reporting engine (X_{21}) which report all the discovered resources. Moreover, the resource register server (X_{22}) nanoservice is used for saving the reported resource configurations. This nanoservice is built with *Python* on top of *txThings* framework.

Table 14: Enabling nanoservices used by proposed algorithm

Steps	Nanoservices	Notations
Step 1	Sensor-Actuator Detection Engine	X_{11}
	Resource Discovery Engine	X_{12}
Step 2	Resource Reporting Engine	X_{21}
	Resource Register Server	X_{22}
Step 3	Service Discovery Engine	X_3
Step 4	Service-Resource Matching Engine	X_4
Step 5	Service Deployment Engine	X_5

On the step 3, the service discovery engine (X_3) is used to discover the nanoservices required by use case. Initially *docker* native *search* tool [29] is used to explore the nanoservices from our predefined Docker Hub service repository. The *docker* native *search* tool is unable to detect recent changes in the nanoservice images. Later, we use the *docker* engine API which can fetch new changes to the *docker images* [30]. In this step, *jq* is used at the active manager node, to filter the requirements. The *curl* tool is used to make HTTPS request during the service discovery through the *docker* engine API.

During the step 4, the service-resource matching engine (X_4) nanoservice consists of *Python* and is used to select the suitable worker nodes for each nanoservice. Finally, at the step 5, the service deployment engine (X_5) is responsible to start deploying the nanoservices into the suitable nodes through *docker-compose* utility.

C. DEPLOYMENT APPROACHES

We deploy the proposed the aforementioned enabling nanoservices in two alternative approaches: non-containerized and containerized deployment.

Non-containerized deployment is directly involved with different host machines. This means that the required nanoservices are deployed directly without having a separate virtualized layer. Therefore, for each worker node, an additional continuous monitoring system is required to ensure the availability of the needed enabling nanoservices required by all five steps. Moreover, laborious service upgrades, rolling back and auto-scaling are the clear weaknesses when the deployment is performed without containers.

Containerized deployment, for one, boosts the formation of a sustainable cluster by generating a lightweight virtualized environment on top of each machine to ensure the efficient use of system resources and low-effort upgradeability [31], [32]. Container Orchestration engine ensures the availability of the needed nanoservices whenever a node joins to the cluster. Thus, additional continuous monitoring system, which would be required at non-containerized deployment approach, is not needed. Common orchestration features such as service upgrading and rolling back, auto-scaling and self-healing are achieved by this deployment.

In a nutshell, containerized approach brings several application lifecycle management-related benefits over the non-containerized approach with some expected reduction in the performance and efficiency of the deployment. Therefore, we identify the containerized approach as more favorable approach with respect to the desirable above-mentioned features.

D. DEPLOYMENT OF THE SCENARIO

We deploy the nanoservices defined in Table 14 which enable our proposed dynamic deployment model. At first, our mechanism discovers the worker node configuration presented in Table 11. For example, pulse oximeter device (BerryMed BM1000B) reads the SpO_2 and HRV from its sensors. During the resource discovery, our proposed mechanism successfully detects the sensor at the worker node W_4. At the step 1, W_4 node is labelled with type *pom* to indicate that the pulse oximeter sensor is discovered at W_4.

During the step 2, all worker nodes send their discovered resources to notify the active manager M_1. The active manager node is aware of the computational capacity of a worker node once the worker accomplished its' reporting. After the reporting of the W_4, the active manager node knows that W_4 requires SpO_2 and HRV nanoservices.

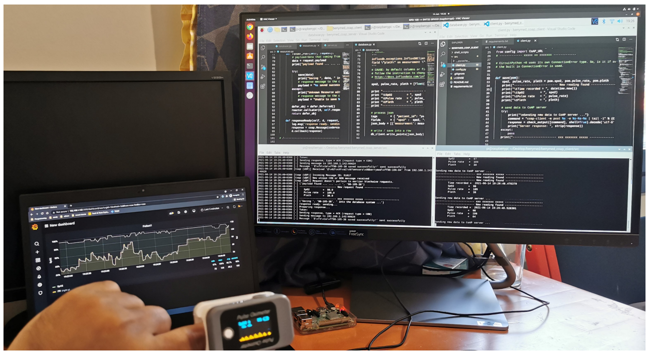


Figure 5: Testbed setup.

We set the requirements of the nanoservices according to use case scenario. For instance, SpO_2 and HRV nanoservices require pulse oximeter. We add *pom* to the requirement at both nanoservices to indicate that these nanoservices require the pulse oximeter to read the data. In this manner we build all nanoservices according to their requirements and push them into our Docker Hub nanoservice repository. At the step 3, our proposed mechanism obtains the nanoservices requirements from the service repository i.e. Docker Hub. Nanoservice discovery result is presented in Table 12.

At the step 4, our proposed algorithm performs service-node matching. The nodes selected for each nanoservice, are presented in Table 13. During the step 5, the active manager node starts deploying each nanoservice defined Table 1 into

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3102867, IEEE Access

the selected nodes. The successful dynamic deployment of nanoservices is presented in Figure 5.

VII. EVALUATION RESULTS

In this section, we compare the resource consumption and end-to-end latency for the containerized and non-containerized deployment approaches to evaluate the resource-efficiency and performance of the proposed mechanism.

A. STORAGE CONSUMPTION

Nanoservices consume different amount of storage based on their requirements e.g software packages or tools. An OS executes and runs various applications through one or many binary executable programs [33]. Depending on the nature of an executable, our nanoservices require Python interpreter or GCC compiler. In the use case, nanoservices X_{12} and X_{21} require *GCC* while rest require *Python*. Besides that, nanoservices require additional software tools and components to run the nanoservices. However, different package management tools are used to install, update, upgrade, configure and remove these binary executables [34], [35].

We deploy the nanoservices defined in Table 14 into different suitable host machines. According to the Table 15, resource discovery and reporting related X_{12} and X_{21} nanoservices are deployed into each work node. Furthermore, different managerial X_{22}, X_3, X_4 and X_5 nanoservices are deployed into active manager node M_1. The sensor-actuator detection nanoservice i.e X_{11} is deployed at M_2 manager node, though it could be deployed at any node. Based on the deployment approach, these nanoservices consume different storage in their target nodes. The storage consumption is summarized in Table 15.

1) Non-containerized deployment

In this case, nanoservices mentioned in Table 14, are built with debian based *apt* package management tool and deployed directly into each host machine without having a separate virtual layer. The nanoservices deployed at both manager node are built with Python 2.7.16 where each worker node uses GCC to build the target nanoservices required by each worker node. Through column 4 & 5, storage consumption for the non-containerized deployment is presented in the Table 15. Here, column 4 represents the consumption for the base tool or component e.g. Python or GCC whereas column 5 indicates the consumption for the additional tools or components required by a nanoservice. The total storage consumption is the sum of column 4 and 5. According to the Table 15, for non-containerized deployment approach, nanoservices deployed at M_1 and M_2 consume 109.96 MB and 112.06 MB respectively while the nanoservices required by a worker node consume 35.11 MB.

2) Containerized deployment

With docker, we build custom lightweight container images for the nanoservices presented in Table 14. These

Table 15: Storage consumption

Steps	Nanoservices	Deployed at	Storage consumption (MB)			
			without container		with container	
			base tool	others	base image	others
Step 1	X_{11}	M_2	77.61	+ 34.45	61.65	+ 36.04
	X_{12}	$W_1 - W_5$	26.94	+ 8.01	3.77	+ 17.84
Step 2	X_{21}			+ 0.16		+ 1.88
	X_{22}	M_1	77.61	+ 31	61.65	+ 31.93
Step 3	X_3			+ 1.35		+ 2.69
Step 4	X_4			+ 0		+ 0
Step 5	X_5					

lightweight images are generated from an existing structure known as base image [36], [37]. Docker generates a base image including a fundamental software component with basic OS related commands. For example, a python related base image contains a Python interpreter along with basic OS command such as *cd*, *ls* etc. We build the nanoservice container images from associated base alpine images depending on their required base tool. X_{12} and X_{21} container images are built from *alpine:3.12.0* base image whereas other nanoservices built from *python:2.7.16-alpine* base image. According 6th column of Table 15, the *alpine:3.12.0* base image consumes 3.77 MB whereas *python:2.7.16-alpine* consumes 61.65 MB disk space. The additional software package or components are managed by *apk* alpine package manager [38]. The disk consumption for additional required software packages is presented in column 7. With additional required software packages, both base images build custom lightweight container images for different nanoservices. The total storage consumption for a containerized nanoservice is the summation of column 6 and 7. Storage consumption for each step is shown in Table 15. According to the table, for containerized deployment approach, the nanoservices deployed at M_1 and M_2 consume 96.27 MB and 97.69 MB respectively while nanoservices deployed at each worker node consume 23.49 MB.

3) Summary of Resource Consumption Analysis

In both non-containerized and containerized deployments, common packages are shared among the nanoservices with *apt* package manager and Docker respectively. In containerized deployment scenario, the base requirement is less consuming as compared to non-container one. We also notice that, in the containerized nanoservices, the size of the depending packages/tools are slightly increasing. However, related nanoservices deployed at M_1 and M_2 are 14.22% and 14.71% more storage consuming while we choose non-containerized deployment over containerized one. In the worker nodes, associated containerized nanoservices are 49.47% less storage consuming as they compared with non-container deployment approach.

B. LATENCY

Each step of the dynamic nanoservice deployment introduces processing delay (i.e latency, measured in milliseconds *ms*). Figure 6 shows deployment steps and their associated latencies. The steps are depicted with numeric circles while latencies are depicted with capital letter L with

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3102867, IEEE Access

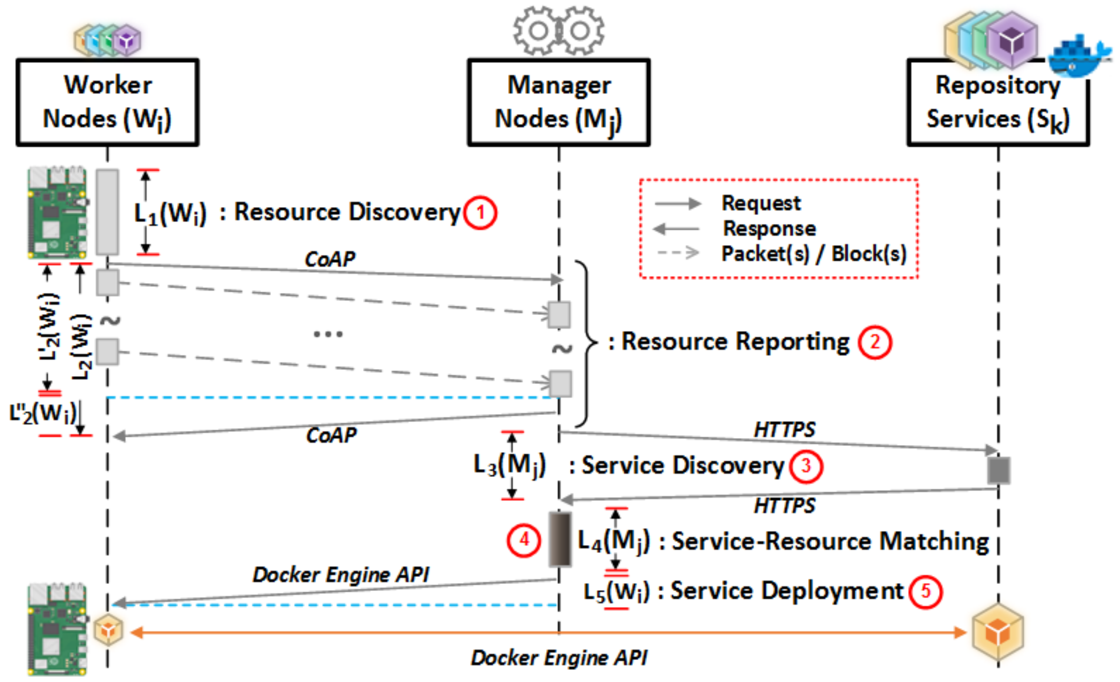
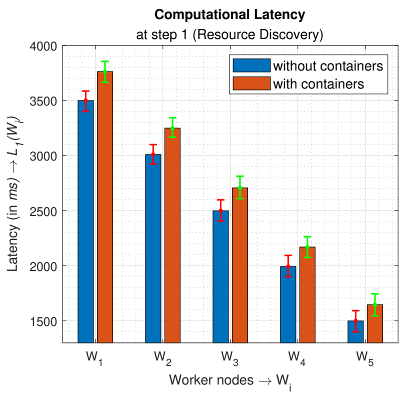


Figure 6: Overall latency profile.

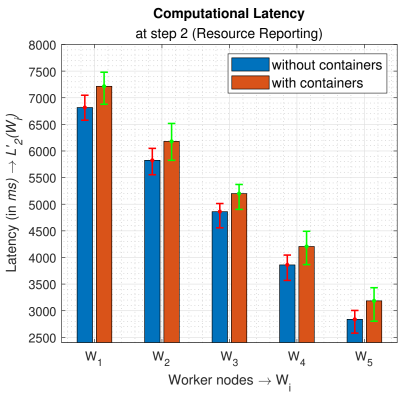
the index of the corresponding step and the processing node (W_i or M_j).

The $L_1(W_i)$ represents the time required to discover all the available resources at the worker node W_i during the step 1 whereas $L'_2(W_i)$ represents the time required to send all the discovered resources to the active manager node (M_1) during the step 2. In the step 2, block-wise CoAP data transfer mechanism over UDP is introduced during the resource reporting to process the large data volume. However, the active manager node M_1 sends a confirmation message to the W_i worker node once the reporting is done successfully which introduces latency $L''_2(W_i)$. At the step 3, the active manager node M_1 requires $L_3(M_1)$ during discovery of the nanoservices from a docker hub repository and $L_4(M_1)$ to match the nanoservices and selects appropriate worker nodes during the service-resource matching at step 4. At the step 5, the active manager node M_1 commands the docker engine to deploy the nanoservices into the appropriate worker node W_i, which takes $L_5(M_1)$ ms.

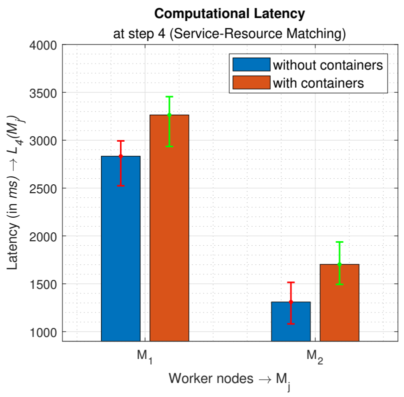
Total latency consists of *(i) computational latency* at step 1, step 2 and step 4 and *(ii) communication latency* at step 2, step 3 and step 5. We analyse computational and communication latency for both non-containerized and containerized deployment approaches. During the measurement at every steps, we take 20 samples and calculate the mean latencies for an associated node (W_i or M_j). The mean latencies and their variations for the non-containerized and containerized deployment are presented in Fig. 7 and Fig. 8.



(a) at Step 1



(b) at Step 2

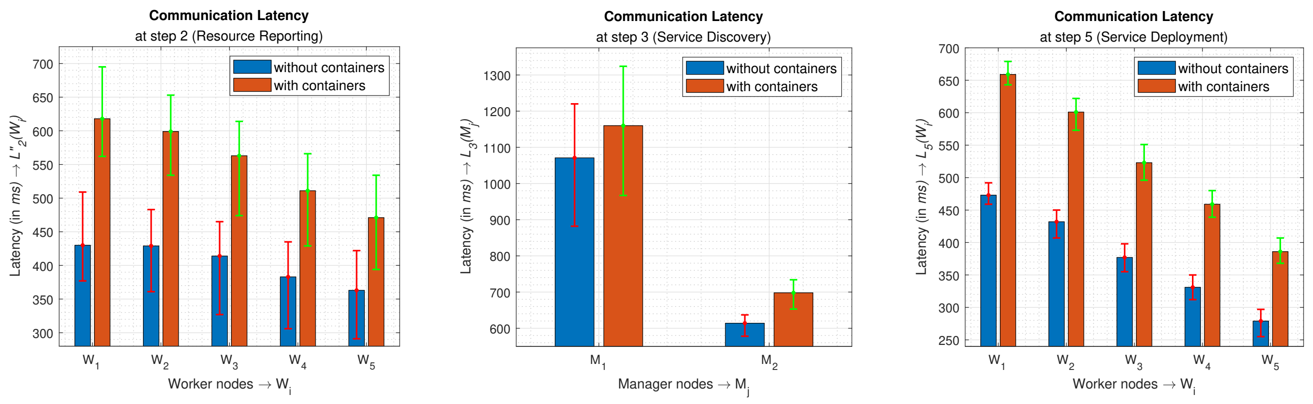


(c) at Step 4

Figure 7: Computational latency without and with containers.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3102867, IEEE Access



(a) at Step 2 (b) at Step 3 (c) at Step 5

Figure 8: Communication latency without and with containers.

1) Computational latency

The computational latency comprises of $L_1(W_i)$, $L'_2(W_i)$ and $L_4(M_1)$ for resource discovery (step 1), resource reporting (step 2) and service-resource matching (step 4) respectively. The computational latencies for these steps are presented in Fig. 7.

During the resource discovery at step 1, X_{12} nanoservice populates a resource profile (P_i) which is nearly 158.2 *kB* at each worker node. We measure the total time required to generate P_i and calculate the mean computational latency for step 1. The overall mean computational latency $L_1(W_i)$ at step 1 for containerized and non-containerized deployments are shown in Fig. 7(a).

During step 2, with X_{21} nanoservice, each worker node initially sends the discovered resource profile P_i to the active manager node with the CoAP's default block size of 64 bytes per message block. With the default block size Worker node W_1 takes 247000 ms (i.e 247s) requiring 2445 message blocks. To speed up the resource reporting, we increase the block size to CoAP's maximum block size, i.e 1024 **bytes**. In this case, we require 153 message blocks only. We measure the mean latency for the reporting process at each worker node. For the non-containerized and containerized X_21 nanoservice, the mean computational latency $L'_2(W_i)$ at each worker node W_i are shown in Fig. 7(b).

At step 4, with X_4 nanoservice, the service-resource matching engine matches requirements of nanoservices with the available resources of worker nodes. The average computational latency $L_4(M_1)$ measured at the active manager node M_1 for the non-containerized and containerized deployments are presented in Fig. 7(c).

Total computational latency:

We observe the computational latencies for both non-containerized and containerized deployment approaches of the nanoservices required by steps 1, 2 and 4. In a certain node, the container-based nanoservices introduce more computational latency as compared to non-containerized nanoservices. For instance, at the step 1, worker node W_1 takes 6980 ms (the orange bar) with non-containerized deployment, and 7175 ms (the blue bar) with containerized deployment. Nanoservices used at these steps are served from virtualized server components. When a host machine receives a request, it forwards the request to a containerized nanoservice. As a result, these containerized nanoservices require extra time in response to the coming request from a host. Non-containerized nanoservices do not require this extra time as they execute directly from the host machine.

2) Communication Latency

The communication latency comprises of $L''_2(W_i)$, $L_3(M_j)$ and $L_5(W_i)$ for resource reporting (step 2), service discovery (step 3) and service deployment (step 5) respectively. The communication latencies for these steps are presented in Fig. 8.

At the step 2, we use the CoAP protocal during the resource reporting. However, CoAP uses the unreliable UDP protocol when the payload is exchanged in between the server and client. As a result, in the reporting phase, there is a chance for a packet loss. Our CoAP server sends a confirmation message to ensure the delivery of the payload. This confirmation process introduces additional latency $L''_2(W_i)$. The observations of $L''_2(W_i)$ for different worker nodes are summarized in Fig. 8(a).

In step 3, nanoservices are discovered from Docker Hub repository through HTTPS request. At each container image, the nenoservice requirements are written in the *description* field. These requirements are extracted by the active manager node during the step 3. We summarize our observations related to Step 3 in Fig. 8(b).

At step 5, the active manager node sends a nanoservice deployment request to the docker engine at the worker nodes. The worker immediately starts downloading the nanoservice from the docker-hub repository and deploys the nanoservice at their own. For each worker node, we observe the communication latency at step 5 for the containerized deployment of the X_5 nanoservice over the non-containerized deployment. We summarize the observations in Fig. 8(c).

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3102867, IEEE Access

Total communication latency:

Total communication latency is higher with containerized nanoservices than without it. In case the X_3 is deployed at the manager node with a container, the mean communication latency increases 8.31% and 13.68% at manager nodes M_1 and M_2 respectively. The reasons for this such as port forwarding or mapping from host to container are already discussed in VII-B1.

3) Total latency

During the resource reporting phase at step 2, worker node W_i splits the whole 152.8 kB payload into 153 blocks having 1024 B payload size for each block. Each worker node sends message blocks to the active manager node. In some of our measurements, the active manager node starts the *service discovery process* as soon as it gets the last massage block from worker node W_i. In this scenario, the active manager node does not wait for the response message that is to be sent to the worker node W_i. Therefore, the communication latency $L''_2(W_i)$ should not consider during total latency calculation. Overall latency for a worker node W_i and manager node M_j is:

$$L = L_1(W_i) + L'_2(W_i) + L_3(M_j) + L_4(M_j) + L_5(W_i)$$

Table 16: Total deployment latency

Total latency (ms) without containers					
	Worker nodes (W_i)				
Manager nodes (M_j)	W_1	W_2	W_3	W_4	W_5
M_1	14693	13170	11640	10088	8520
M_2	12713	11190	9660	8108	6540
Total latency (ms) with containers					
	Worker nodes (W_i)				
Manager nodes (M_j)	W_1	W_2	W_3	W_4	W_5
M_1	16061	14455	12854	11259	9641
M_2	14038	12432	10831	9236	7618

The total latencies are presented in Table 16. Total latency increases by approximately 1-1.5s when container is used compared to non-containerized deployment. Similar observations are made for both M_1 and M_2 manager nodes. When we take into account $L''_2(W_i)$, the total latency $L(W_i)$ at worker node W_i is increases for approximately $0.5s$ in both deployments.

VIII. DISCUSSION AND FUTURE DIRECTIONS

This work provides a potential solution for a dynamic deployment of IoT services, consisting of decentralized nanoservices, in a heterogeneous cluster of IoT nodes. In this paper, we consider a remote healthcare monitoring use case for a COVID infected patient to highlight the benefits of dynamic nanoservice deployment in a real-world scenario. For this purpose, we demonstrated the feasibility of dynamic resource allocation through PoCs implementation. The proposed resource-aware dynamic nanoservice deployment mechanism shows how the needed nanoservices are deployed from a Docker-Hub repository to resource-constrained IoT cluster nodes based on service requirements.

With the proposed resource-matching mechanism, the nearby available hardware resources can be dynamically discovered and matched with medical service requirements to deploy different parts of the medical service to the most suitable nodes in the cluster of available local nodes. Dynamic resource availability is vital for ensuring continuous monitoring of a patient in our scenario, where the patient along with the attached medical sensors and needed equipment/local computing hardware moves from home to ambulance and then from ambulance to the hospital. In this case migration/orchestration of the services/resources (from home to ambulance and then to hospital) are managed by the edge servers.

We have evaluated the storage consumption and the nanoservice deployment latency in two different deployment approaches: with and without containerization. Both approaches have their own merits and demerits. The deployment of container-based nanoservices takes 1-2 seconds longer than with non-containerized services. On the other, non-containerized nanoservices are more resource consuming compared to containzerized nanoservices. This comes from the additional overhead of sending the requests from the host to the container application. However, container-based deployment has significant benefits in terms of better upgrading mechanism, scalability, self-healing and automated bin packing with minimal downtime. Therefore, additional 1-2 seconds in the deployment phase are tolerable.

This evaluation work was performed in the context of a remote monitoring healthcare scenario for a COVID patient. However, the results can be generalized to any IoT application scenario with dynamically changing service requirements. For example, in the case when there is unstable access network connectivity, the deployment of dynamic nanoservices is vital for executing the local processes/tasks until the connection to access networks get stable.

Our work has several future directions. In this work, we have enabled dynamic nanoservice deployment, based on the availability of the hardware resources in the cluster. However, we did not yet consider the current load or performance of the cluster nodes in the nanoservice deployment. Therefore, using AI/ML approach to enable dynamic load and performance-aware service deployment would further improve the performance of decentralized nanoservice architectures on clusters of resource-constrained IoT nodes, and would therefore be an interesting direction for future research. Future work also includes distributed mechanisms to establish trust between different nanoservice providers, such as DLT and Blockchain approaches, to ensure sufficient privacy and security of the local IoT services.

IX. CONCLUSION

This article proposes and evaluates a model for enabling dynamic resource-service matching in distributed local edge computing. This work extends our previously developed lo-

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3102867, IEEE Access

cal edge computing architecture (nanoEdge) for constrained IoT setups, by enabling automatic resource discovery and deployment on highly dynamic IoT scenarios. To showcase the feasibility of the model in real-life scenarios, we have chosen a topical Covid-19 patient monitoring use case as a basis for evaluation. The proposed dynamic resource-service matching mechanism is evaluated by implementing and comparing two alternative approaches, a containerized approach providing on-the-fly configuration of nanoservices, and a simple non-containerized approach with fixed service configuration.

According to the results, the service deployment takes slightly more time when containers are used, compared to the non-containerized approach. On the other, containerized nanoservices are more resource efficient. Overall, containerization provides clear advantages in terms of service management, such as effortless upgrading, rolling back and auto-scaling. It also ensures the sufficient scalability, resource-efficiency and fault-tolerance required by highly dynamic yet resource-constrained IoT scenarios. Although the evaluation was made for a healthcare scenario, the results can be generalized to any IoT scenario with dynamically changing service requirements and available resources in unstable access network connectivity. The future work includes taking into consideration the current load and performance of devices in the nanoservice deployment and distributed DLT/Blockchain mechanisms to establish trust among various service providers.

ACKNOWLEDGMENT

The authors would like to thank Mr. Jude Okwuibe and Mr. Muneeb Ejaz for productive discussions during the writing process of this paper.

References

[1] C.-Y. Fan and S.-P. Ma, "Migrating monolithic mobile application to microservice architecture: An experiment report," in *2017 IEEE International Conference on AI & Mobile Services (AIMS)*. IEEE, 2017, pp. 109–112.

[2] S. Newman, *Monolith to microservices: evolutionary patterns to transform your monolith*. O'Reilly Media, 2019.

[3] D. Taibi, V. Lenarduzzi, and C. Pahl, "Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation," *IEEE Cloud Computing*, vol. 4, no. 5, pp. 22–32, 2017.

[4] G. Premsankar, M. Di Francesco, and T. Taleb, "Edge computing for the internet of things: A case study," *IEEE Internet of Things Journal*, vol. 5, no. 2, pp. 1275–1284, 2018.

[5] "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.

[6] H. Ning, F. Farha, Z. N. Mohammad, and M. Daneshmand, "A survey and tutorial on "connection exploding meets efficient communication" in the internet of things," *IEEE Internet of Things Journal*, vol. 7, no. 11, pp. 10 733–10 744, 2020.

[7] M. S. Aslanpour, S. S. Gill, and A. N. Toosi, "Performance evaluation metrics for cloud, fog and edge computing: A review, taxonomy, benchmarks and standards for future research," *Internet of Things*, p. 100273, 2020.

[8] F. Wu, C. Qiu, T. Wu, and M. R. Yuce, "Edge-based hybrid system implementation for long-range safety and healthcare iot applications," *IEEE Internet of Things Journal*, vol. 8, no. 12, pp. 9970–9980, 2021.

[9] 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.

[10] J. Islam, E. Harjula, T. Kumar, P. Karhula, and M. Ylianttila, "Docker Enabled Virtualized Nanoservices for Local IoT Edge Networks," in *2019 IEEE Conference on Standards for Communications and Networking (CSCN)*, 2019, pp. 1–7.

[11] Z. Houmani, D. Balouek-Thomert, E. Caron, and M. Parashar, "Enhancing microservices architectures using data-driven service discovery and qos guarantees," in *The 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing CCGrid*, 2020, p. 10.

[12] S. P. Singh, A. Nayyar, R. Kumar, and A. Sharma, "Fog computing: from architecture to edge computing and big data processing," *The Journal of Supercomputing*, vol. 75, no. 4, pp. 2070–2105, Apr. 2019. [Online]. Available: http://link.springer.com/10.1007/s11227-018-2701-2

[13] J. Portilla, G. Mujica, J. Lee, and T. Riesgo, "The extreme edge at the bottom of the internet of things: A review," *IEEE Sensors Journal*, vol. 19, no. 9, pp. 3179–3190, 2019.

[14] E. M. Dogo, A. F. Salami, C. O. Aigbavboa, and T. Nkonyana, "Taking cloud computing to the extreme edge: A review of mist computing for smart cities and industry 4.0 in africa," in *Edge computing*. Springer, 2019, pp. 107–132.

[15] R. Chen, S. Li, and Z. Li, "From monolith to microservices: A dataflow-driven approach," in *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, 2017, pp. 466–475.

[16] T. Prasandy, Titan, D. F. Murad, and T. Darwis, "Migrating application from monolith to microservices," in *2020 International Conference on Information Management and Technology (ICIMTech)*, 2020, pp. 726–731.

[17] S. Pallewatta, V. Kostakos, and R. Buyya, "Microservices-based iot application placement within heterogeneous and resource constrained fog computing environments," in *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, 2019, pp. 71–81.

[18] T. Leppanen, C. Savaglio, L. Lovén, T. Jarvenpaa, R. Ehsani, E. Peltonen, G. Fortino, and J. Riekki, "Edge-based microservices architecture for internet of things: Mobility analysis case study," in *2019 IEEE Global Communications Conference (GLOBECOM)*. IEEE, 2019, pp. 1–7.

[19] A. Sattari, R. Ehsani, T. Leppänen, S. Pirttikangas, and J. Riekki, "Edge-supported microservice-based resource discovery for mist computing."

[20] M. Ejaz, T. Kumar, M. Ylianttila, and E. Harjula, "Performance and efficiency optimization of multi-layer iot edge architecture," in *2020 2nd 6G Wireless Summit (6G SUMMIT)*, 2020, pp. 1–5.

[21] T. Kumar, E. Harjula, M. Ejaz, A. Manzoor, P. Porambage, I. Ahmad, M. Liyanage, A. Braeken, and M. Ylianttila, "Blockedge: Blockchain-edge framework for industrial iot networks," *IEEE Access*, vol. 8, pp. 154 166–154 185, 2020.

[22] [Online]. Available: https://www.hexoskin.com/pages/hexoskin-connected-health-platform

[23] [Online]. Available: https://www.philips.fi/healthcare/product/HC989803196871/wearable-biosensor-wireless-remote-sensing-device

[24] A. Industries, "Finger Pulse Oximeter with Bluetooth LE." [Online]. Available: https://www.adafruit.com/product/4582

[25] "Wearable Continuous Temperature Monitor with Adafruit IO." [Online]. Available: https://learn.adafruit.com/wearable-temperature-monitor/assembly

[26] A. J. Puspitasari, D. Famella, M. S. Ridwan, and M. Khoiri, "Design of low-flow oxygen monitor and control system for respiration and spo2 rates optimization," in *Journal of Physics: Conference Series*, vol. 1436, no. 1. IOP Publishing, 2020, p. 012042.

[27] T. DiCola. (2019, apr) Sensors and data logging with embedded linux - the ultimate guide part 1. [Online]. Available: https://www.balena.io/blog/sensors-and-data-logging-with-embedded-linux-the-ultimate-guide-part-1/

[28] 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

[29] D. Docs. docker search. [Online]. Available: https://docs.docker.com/engine/reference/commandline/search/

[30] ——. Engine api v1.24. [Online]. Available: https://docs.docker.com/engine/api/v1.24/#32-images

[31] R. Dua, A. R. Raja, and D. Kakadia, "Virtualization vs containerization to support paas," in *2014 IEEE International Conference on Cloud Engineering*, 2014, pp. 610–614.

[32] J.-P. Rodrigue and T. Notteboom, "Looking inside the box: evidence from the containerization of commodities and the cold chain," *Maritime Policy*

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3102867, IEEE Access

& Management, vol. 42, no. 3, pp. 207–227, 2015. [Online]. Available: https://doi.org/10.1080/03088839.2014.932925
[33] "Executing Binary Programs & Shell Scripts in Linux." [Online]. Available: https://study.com/academy/lesson/executing-binary-programs-shell-scripts-in-linux.html
[34] "PackageManagementTools - Debian Wiki." [Online]. Available: https://wiki.debian.org/PackageManagementTools
[35] "Package Management." [Online]. Available: https://ubuntu.com/server/docs/package-management
[36] "Glossary," Jun. 2021. [Online]. Available: https://docs.docker.com/glossary/
[37] "Dockerfile reference," Jun. 2021. [Online]. Available: https://docs.docker.com/engine/reference/builder/
[38] "Alpine Linux package management - Alpine Linux." [Online]. Available: https://wiki.alpinelinux.org/wiki/Alpine_Linux_package_management

JOHIRUL ISLAM received his bachelor's degree in Information and Communication Technology from Mawlana Bhashani Science and Technology University, Bangladesh, in 2014, and master's degree in Wireless Communications Engineering from the University of Oulu, Finland, in 2019. He is doing his doctoral research under the supervision of Asst. Prof. Erkki Harjula at CWC-NS Networks and Systems research group in the University of Oulu, Finland. His research interests include Internet of Things (IoT), Cloud and Edge computing; and virtualization technologies for intelligent environment.

TANESH KUMAR is currently working as a postdoctoral researcher at the Centre for Wireless Communications (CWC), University of Oulu, Finland. He received his D.Sc. degree in communications engineering from the University of Oulu, Finland, in 2016, the M.Sc. degree in computer science from South Asian University, New Delhi, India, in 2014 and the B.E. degree in computer engineering from the National University of Sciences and Technology (E&ME), Pakistan, in 2012. He has coauthored over 40 peer-reviewed scientific articles. His current research interests include security, privacy and trust in the IoT, 5G/6G edge computing, Blockchain and Medical ICT.

IVANA KOVACEVIC received the bachelor's degree in electronics and telecommunication engineering from the University of Belgrade, Serbia, in 2012 and the M.Sc. degree in communications engineering from the University of Oulu, Finland, in 2015, where she is currently pursuing the Ph.D. degree in wireless networks. In 2015, she joined the Centre for Wireless Communications, University of Oulu. Her research interest is in the area of network slicing, low latency communications, radio resource management, edge computing, network optimization theory, game theory and machine learning.

ERKKI HARJULA works as an Assistant Professor (tenure track) at the Centre for Wireless Communications - Networks and Systems (CWC-NS) research group, University of Oulu, Finland. He focuses on wireless system level architectures for future digital healthcare, where his key research topics are wrapped around intelligent trustworthy distributed IoT and edge computing. Dr. Harjula has background in the interface between computer science and wireless communications: mobile and IoT networks, distributed networks, cloud and edge computing and green computing. He has also long experience as a research project manager. He received his D.Sc. degree in 2016, and his M.Sc. degree in 2007 at University of Oulu. He is a Member of IEEE.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/