Context-Aware Kubernetes Scheduler for Edge-native Applications on 5G

Michael Chima Ogbuachi, Anna Reale, Péter Suskovics, and Benedek Kovács

Original scientific paper

***Abstract*—This paper is an extension of work originally presented in SoftCOM 2019 [1]. The novelty of this work reside in its focused improvement of our scheduling algorithm towards its usage on a real 5G infrastructure. Industrial IoT applications are often designed to run in a distributed way on the devices and controller computers with strict service requirements for the nodes and the links between them. 5G, especially in concomitance with Edge Computing, will provide the desired level of connectivity for these setups and it will permit to host application run-time components in edge clouds. However, allocation of the edge cloud resources for Industrial IoT (IIoT) applications, is still commonly solved by rudimentary scheduling techniques (i.e. simple strategies based on CPU usage and device readiness, employing very few dynamic information). Orchestrators inherited from the cloud computing, like Kubernetes, are not satisfying to the requirements of the aforementioned applications and are not optimized for the diversity of devices which are often also limited in capacity. This design is especially slow in reacting to the environmental changes. In such circumstances, in order to provide a proper solution using these tools, we propose to take the physical, operational and network parameters (thus the full context of the IIoT application) into consideration, along with the software states and orchestrate the applications dynamically.**

***Index Terms*—Edge, IIoT, Kubernetes, 5G.**

I. Introduction

THE 5G and Edge Computing paradigms [2], [3] reflect the current direction of both industry and academia, towards highly responsive, so-called edge-native applications [4]. Edge-native applications, are the subset of cloud native applications requiring proximity to the users, thus Edge, and 5G capabilities: greater data-bandwidth, ultra-low-latency or massive/continuous communication.

The definition applies for highly demanding settings that cannot rely on data-centers because of the inherent latency caused by their physical distance [5]: autonomous vehicles and industry 4.0 manufacturing; advanced media use cases like real-time gaming, augmented reality, virtual/mixed reality and tele-holo-phoning. The 3GPP community refers to these requirements as the Ultra Reliable Low Latency and develops a network standard to fulfill these in both the RAN and core network parts. [6]

Furthermore, in some specific setups, it may be important to run different logic and service parts in the same distributed cluster, e.g. for a distributed robotic control application in a smart factory.

The purpose of this work is to define how to ensure deployment and orchestration of edge-native applications when the cluster is spread among communicating devices at the edge, keeping in mind the peculiarities and challenges of such a new infrastructure. We treat such scenarios as resource management and allocation problems for distributed clusters.

In previous work [1] we proposed an initial scheduling algorithm for Kubernetes that integrates real-time information about the edge devices in the cluster. We validated it on a limited setup, a WiFi-based cluster of Raspberry Pis, comparing its performance to the default solution.

Leveraging on the learning from our first experiments, we reformulated the scheduling scoring system and finalized a new algorithm. The new verification setup in this paper uses real 4G and 5G connectivity, a 5G New Radio together with a functional non-standalone (NSA) 5G core network and corresponding hardware infrastructure. We also present an upgraded profiling tool, to allow Network Address Translation traversal for edge devices.

In fact, the main unique contributions of this paper are: (i) a lightweight CoAP-based tool to collect cluster metrics, able to traverse Network Address Translation when UEs act as modems for devices in the network (i.e. tethering); (ii) an improved design of a Kubernetes compatible scheduler, still based on real-time application context information, but with a more aggressive scoring mechanism to prioritize the balance of resources usages all over the cluster. (iii) tests and measurements on a Kubernetes cluster deployed in a real 5G access and non-standalone core network. (iiii) an extended analysis of the Kubernetes default scheduler trough three categories of IIoT use cases.

Our solution reduces the overall scheduling and application startup time, without increasing resource degradation.

The paper is structured as follows: after putting our paper into the perspective of other related works (Section I-A); in Section II the concept of 5G, Edge Computing and orchestration in relation to IIoT are briefly presented. An extended discussion on IIoT use case simulations is in Appendix A. In Section III the shortcomings of the Kubernetes scheduler

Manuscript received March 10, 2020; revised March 30, 2020. Date of current version March 31, 2020. The associate editor prof. Nikola Rožić has been coordinating the review of this manuscript and approved it for publication.

The part of this paper was presented at the International Conference on Software, Telecommunications and Computer Networks (SoftCOM) 2019.

M. Ogbuachi and A. Reale are with the Department of Programming Languages and Compilers Eötvös Loránd University, Budapest, Hungary (e-mails: micogb@inf.elte.hu, anna.reale@inf.elte.hu). B. Kovács and P. Suskovics are with Ericsson Hungary R&D Budapest, Hungary (e-mails: peter.suskovics@ericsson.com, benedek.kovacs@ericsson.com

Digital Object Identifier (DOI): 10.24138/jcomss.v16i1.1027

are analyzed with a brief description of its algorithm. Our edge scheduler implementation is described in Section IV. We validate our solution with new tests in Section V. Conclusions and future objectives are in Section VI while a richer analysis of the default scheduler is in Appendix B.

A. Related Works

Only the sum of requested resources in each node of a cluster is taken into consideration by the default scheduler in Kubernetes. This is not effective enough when resource optimization should also account for potential sudden and drastic performance degradation. Despite that, there are not too many works focusing mainly on the improvement of the scheduling process in Kubernetes.

Specifying only limits for resource utilization is not enough to avert the risk of resource contention, as shown in [7]. The authors explored the problem and proposed as a result the software architecture for a scheduler which tries to avoid the issue by characterizing the "incoming" applications. Briefly, the scheduler makes an effort to put containers that are characterized by high resource usage in different host machines. The scheduling time also happened to get an improvement in speed that was around 20%, compared to the default scheduler, in a few test scenarios. Similarly to our approach, the authors tried to actualize a balanced distribution of tasks, but the overall speed improvement is not significantly strong and there was no dynamic input taken into consideration.

A close approach to ours in introducing better multi-objective scheduling in Kubernetes has been published in 2019 [8]. The authors formulated energy efficiency as a multi-objective optimization problem between maximal use of green energy, optimal performance with minimal interference, and overall energy minimization. The ILP problem is solved via Mosek Solver.

The orchestration concept based on Kubernetes has been extended to fog computing in [9]. Authors designed a set of labels for the default scheduler to addresses the application deployment challenges in fog set-ups: distribution, connectivity, availability, heterogeneity, and real-time.

Also [10] extends the basic K8s scheduler with labels: applications are classified depending on which resource they use more intensively – CPU, I/O disk , network bandwidth, or memory bandwidth. The score of a node at scheduling time is penalized if similar labels are already present on that node. This ensures that the resulting Kubernetes scheduler can balance the number of applications in each node while still minimizing the degradation caused by resource competition. The overall execution time of the experiment is about eight minutes. This value is only 20% better than the mean time of the Kubernetes scheduler. The total time is similar to the best case of the default scheduler and overall the scheduling process has a lower time variance.

Similar efforts to reduce resource degradation have been expressed in [11]. Authors define a Reference Net (a Petri net) to model performance and management of resources for Kubernetes, identifying different operational states associated with "pods", containers and their shared resources. Such a model may be potentially used to calculate interference generated from certain deployments.

A network-aware scheduling approach for container-based applications in Smart City deployments is proposed in [12]. The Kubernetes scheduler is configured to make use of nodes RTT labels to decide where they are suitable to deploy a specific service with the target location specified on the pod configuration file. After completion of the scheduling request, the available bandwidth is updated on the corresponding node label. The objective of this scheduler is not to reduce strains on nodes nor to reduce the scheduling time, but to enhance the deployment performance by choosing nodes closer to the target of the service. In fact, the overall time of scheduling is slightly increased compared to the default scheduler.

The monitoring mechanism in [13] takes both system resource utilization and application QoS metrics into account. However it is not lightweight since it is composed of a container-based cluster-monitoring tool for Kubernetes, a database designed to store time series data and a visualization application. The authors also provide a dynamically configurable resource provisioning algorithm for K8s. As mentioned in [8] these tools occupy a great number of resources in the cluster.

A notable example of a custom scheduling mechanism that can be installed as a plug-in into Kubernetes is the Poseidon-Firmament Scheduler [14], which tries to solve the scheduling problem by modeling a graph of the network flow, on which it runs its the optimization. It is therefore meant to solve the issue of optimal allocation only in regards to the network performance, which anyway proved to be good enough as an approach to make it 50%-80% faster than the Kubernetes default scheduler in the process of binding a pod to a node.

VM placement [15], such as cloud resource management [16], can be considered to be the "traditional ingredients" in the field of provisioning for resilient online services, which had a strong influence in the research related to scheduling and placement of containers. A very informative listing of several solutions for VM placement can be found in [17]. These solutions are categorized by the objective they want to fulfill, in terms of resource usage optimization, and the categorization underlines the fact that these solutions are specific for a single or very restrictive set of parameters. Our study, on the other hand, was conducted with a different mindset, aiming to design a solution that would allow the infrastructure owner to directly control how many parameters are taken into consideration for the system optimization. In [18] a thorough analysis of weight computation formulas, related to multi-objective optimizations, is presented, especially for tasks like dynamic and automated strategies for decision making.

The paper by Parest [19] shows several other methodologies to accomplish similar estimations, nonetheless, both this and the aforementioned work do not present approaches that make use of online scoring and dynamic weight attribution.

II. Background

In Industrial IoT (IIoT) scenarios such as collaborative robotics, remote robot controlling, flexible reconfiguration of

manufacturing pipeline , etc. introduce new challenges for the involved resources management functions.

It is likely in these scenarios that an Edge instance on the premises will host not only the software in charge of mobile connectivity, but also the orchestration logic.

The general role of an Orchestrator is to handle the applications life-cycles (actions like service binding, querying, copying, updating and deleting). Scheduling configurations are usually simplistic since there is seldomly distribution of the computational resources: quite often Virtual Machines with very similar baseline characteristics and hosted in the same datacenter. The devices that will compose the orchestrated cluster in our IIoT case scenarios are diverse in capabilities, resources, availability, life-time, supported temperatures and so on.

Cloud-native orchestrator solutions need to be adapted and customized towards this context. In the next section we will briefly explain how IIoT applications benefit from 5G and Edge computing infrastructures and what characteristics of current orchestration solution made us verge towards Kubernetes as a starting point for IIoT edge-native applications orchestration.

A. 5G and Edge Computing in Industrial IoT

Employing wireless systems for IIoT has been a challenge: the classical wireless technologies used on information sensing do not satisfy the requirements of ultra-reliable low-latency communication (URLLC), deep coverage, ability to scale-up deployments and efficient distributed operations for factory plants.

Applications of 5G communication technology in robotics are demonstrated in [20], especially how the URLLC capabilities of the 5G can facilitate a distributed robotics control system.

Authors in [21] list several use cases and technical requirements for the main three capabilities of 5G applied to IIoT [22]: Infrastructure retrofit, Mobile robots, Inbound logistics for manufacturing, flexible and modular assembly area, plug-and-produce, Massive wireless sensor network, and process monitoring.

As a further example, in fault detection and diagnostics (FDD) systems data transmission latency usually needs to be maintained at the scale of milliseconds, which can hardly be met through classical wireless networking technologies [23].

URLLC, is one of several different types of use cases supported by the 5G.

Many 5G URLLC designs and experiments have been conducted in this direction [24]–[26].

From the network architectural perspective, Edge Computing (and it's variations [27]: Fog computing [28], Mobile Cloud Computing (MCC), cloudlet computing, MEC [29]...) can help even further in reducing overall latency, by allowing computational capacity to sit right next to the non standalone 5GNR and the IIoT devices. According to the maturity of 5G standards and the implementation, we've chosen the 5G NSA deployment to validate our proposal.

B. Non-standalone 5G

5G standard for future networks has two versions: Non-Standalone 5G (NSA) and Standalone 5G. The specifications indeed provide a general technology framework that addresses different and, in some cases, conflicting 5G requirements and is forward compatible to accommodate future applications and use cases [30].

Non-Standalone 5G primarily focuses on enhanced mobile broadband (eMBB), where the 5G supported mobiles will use mm-Wave frequencies for increased data capacity but will use existing 4G infrastructure for voice communications.

C. Kubernetes as Orchestrator for IIoT

Among the prominent orchestrator solutions of this years we identified: Docker Swarm [31], Openstack [32] and Kubernetes [33].

Kubernetes (K8s) can be considered as an orchestrator since it is able to automate deployment, scaling, and management of containerized applications. In the cluster a master node instantiates services and deployments aggregating containers into Pods. Pods are indivisible units running at the worker nodes. Containers in a pod, thus, share resources, filesystem, kernel namespaces, and an IP address. [34]:

Docker Swarm and Kubernetes are good tools to prototype container run-times [35]. Although easy to deploy, Docker is not a scalable product solution for enterprise applications and large deployment like those that we envision in an IIoT context. OpenStack is not meant to be run on limited devices [36], which is also the scope of this work; thus, the choice fell on Kubernetes. The current market also confirms our choice, since Kubernetes retains the largest cut and is a *de facto* standard [37].

It is important to state that from March 2019, a stripped down version of Kubernetes for IoT and Edge application has been released: K3s [38]. Scheduling, network and cluster logic are kept the same and only the kubelet size is significantly reduced through a reorganized plugin structure and a less resource-intensive database (sqlite3). When faced with the choice between K3s and K8s, the latter was still preferred, considered the following:

1) The K3s project was fairly too recent and not production-ready, not enough information was available online;
2) The scheduler component was identical in both Kubernetes;
3) K3s does not allow multiple masters, so if the master goes down the whole cluster is lost;
4) a K3s cluster is not compatible with K8s, adding unnecessary complexity in the case of multilayered orchestration, to enable aggregated control of clusters located in distributed facilities.

III. KUBERNETES SCHEDULER

For our specific scenario, we found that the most lacking orchestration feature in K8s was related to the scheduling techniques. As expressed in Appendix B, the amount of time spent scheduling and rescheduling a pod may grow

significantly, with service disruptions up to a minute in case of node death. In the default K8s scheduler nodes are evaluated at scheduling time; when the user creates a new pod and assigns it to the Kubernetes cluster, the pod gets into the event stream (Object Store) with a "Pending" status. The scheduler watches constantly for this type of event and decides the most appropriate binding to a node.At first, a subset of *feasible schedules*, containing only those nodes that are satisfying the given constraints for the deployment are collected; Then the scheduler computes a subset called *viable schedules*, which ranks the selected nodesbased on scoring functions:

1) PodFitsResources: If the free amount of CPU and memory on a given node is enough
2) *NoDiskConflict*: if a pod can fit due to the volumes it requests
3) *NoVolumeZoneConflict*: checks possible zone restrictions.
4) *PodFitsHostPorts*: check if the needed port is free
5) *CheckNodeMemoryPressure* and *CheckNodeDiskPressure*: if a pod can be allocated on a node reporting memory pressure condition or disk pressure condition.
6) *MatchNodeSelector (Affinity/Anti-Affinity)*: By using node selectors (labels), it is possible to define that a given pod can only run on a particular set of nodes or that it cannot be allocated on a node that has already certain pods deployed (pod-anti-affinity).

The scheduler also uses "general-purpose" cloud computing scheduling criteria, called priorities: for example *ImageLocalityPriority* ranks according to the location of the requested pod container images.

Our own scheduler implementation is a hybrid between multi-step [39] and single-step scheduling. Multi-step scheduling is the approach of K8s default scheduler: each pod is considered independently to provide a local optimum for each allocation process. Overall this technique requires more maintenance processes, especially in case of rearrangements based on pod priorities, pod preemption and reassignment.

In our case we do not perform exactly Single-step scheduling since the optimum is calculated every batch of pods.

IV. Edge Cloud Scheduler Implementation

There was a possibility to instantiate the custom scheduler as a pod on the master node to profit from the virtual network built up by K8s. Our version does run in the master node of the cluster, however, it is outside of K8s control, to avoid delays and interference from the default scheduler. Hence we preferred to handle the communication separately and make use of an external asynchronous module, so that the scheduler itself can operate without an excessive computational overhead.

The status data fed by the nodes of the cluster are input to the scheduler for node score computation. The requirements of a Pod also influence the choice.

The scheduling mechanism is able to dynamically adapt to the changes, and its fair since it distributes workloads based on the actual node status.

The motivation behind our approach also relies on the observation that the existing scheduler does not fully satisfy the need to preserve the health of the cluster and its services. Sharing physical resources among containers might lead to a degradation in the performance of the applications running inside them [10]. Kubernetes resource reservation mechanism is only available for CPU and RAM. However other resources are shared such as bandwidth and network access [40].

In the following paragraphs, we will first describe our improved tool to collect information about network and cluster context (IV-A),finally the scheduler algorithm described (IV-B) : first the task prioritization and candidates selection are summarized, then the new score computation rationale is presented.

A. Monitoring Agents

The limited resources on our minion nodes required a stripped-down solution to monitor their resources. Kubernetes monitoring solutions mostly employ greater resources since they rely on multiple containers and more complex networking. In our solution monitoring agents were added to the minion nodes for reporting run-time data of the edge devices to the edge-cloud network management system. The agent uses CoAP (Constrained Application Protocol) [41] for the communication between client and servers. When gathering data, the master node acts as the client and edge devices act as the servers.Low message overhead, low latency and high efficiency in comparison [42] with other IoT communication protocols such as MQTT and DDS, made us verge toward this solution. CoAP follows a Client-Server model, our implementation takes advantage of an asynchronous function facility in python, *asyncio*, which facilitates execution of concurrent operations. The client sends multiple requests to all the connected servers and collects the responses, then exposes the collected data on localhost. This simplified service is always available for the custom scheduler to gather data simultaneously from all nodes.

Such a solution is completely independent of K8s, so it can be replaced and maintained without having to update the cluster, which is often an expensive and error prone task.

The client can choose what parameters to request, via command-line. The frequency of the collection of all parameters can be adjusted dynamically. On the edge devices, a server-like application handles GET request, and is able to measure the temperature of the CPU and its usage; total amount of free and used physical memory, and total amount of swap memory. Network parameters are: latency, jitter and packet loss. In the case of NAT, a *login* mode can be activated to initiate an ACK sequence from the device to the master. This will ensure that any farther communication can transverse NAT.

B. Customized Scheduler

To integrate to the default scheduler our customized version will fetch nodes and pods through Python Kubernetes APIs, then select pods with "phase=Pending" and matching in the *schedulerName* property.

As for our previous experiments [1] the'hybrid' single-step scheduling optimizes the pods globally, as a single-step

scheduler would do, but it categorizes the pods and arranges them locally.

The specification of resource requirements in the Pod influence the allocation. Memory and CPU and any other custom label can be easily specified for this phase, just like in the default scheduler.

The scheduler will however take also inputs data from the metric agents and its finals scores may be influenced by the infrastructure owner as he sees fit.

The list of pods to deploy will be ordered from the most resource-intensive pod to the least one. Pods with no requirements remain in an untouched order on a separate list.

Ordering by resource consumption is a powerful way to simplify the knapsack problem [43] towards a greedy solution.

Nodes are first classified based on their *Usability* for that specific scheduling time: Liveness (is it responding to ALIVE probes), CPU and Memory (are enough and available for the job).

To build the *priority list* nodes receive a score based on their run-time state.

In the investigation from [18] multiple automated placement decision algorithms are compared and ranked. Among those, our improved score computation solution is close to the Coefficient of Variation (CV) weighting method, combined with Multiplicative Exponent Weighting (MEW). This upgrade was done based on the observation that this combination was the one resulting with a greater influence towards highly volatile parameters.

The properties that are considered are formulated so that their minimizations correspond to an increased health for the system.

The new score formulation is:

$$score = \prod_{i=1}^{|P|} \left(\frac{p_{min}}{p_i} \right)^{-w_i} \tag{1}$$

For each parameter related to the context of the scheduling, $P := set\ of\ parameters$, the stream of values received from the devices is a time series data. The weighted scores will be $W := set\ of\ weights\ of\ the\ parameters$ with $|W| = |P|$; so that each p_i parameter data from a node device, is normalized by the minimum value it historically reached for that node; and with w_i the weight assigned to it.

For this multi-objective problem the weight aggregation strategy is kept as in our previous algorithm [1]:

$$w_i = D_i = \frac{\sigma_i^2}{\mu_{i,n}} \tag{2}$$

The weight is the index of dispersion of the values taken by the parameters of a node at run-time. The scheduler will use the weights to compute how compromised is the operational state of a certain node.Since the computation is based on time series, the longer the scheduler runs, the stronger the influence of the diverging parameters will be, ensuring the balance between the different objective functions.

Since the history of the parameters is linearly growing we simplify the complexityvia an*online update*, or recursive estimation of the mean, as shown below:

$$\mu_{i,n} = \mu_{i,n-1} + \frac{(x_n - \mu_{i,n-1})}{n} \tag{3}$$

$$\sigma_{i,n}^2 = \frac{\sigma_{i,n-1}^2 (n-1) + (x_n - \mu_{i,n-1})(x_n - \mu_{i,n})}{n} \tag{4}$$

V. Setup and Evaluation Criteria

The scheduler and the metrics component were implemented in two versions each. Two different Setups where tested. An Old Setup [1] within WiFi and a New Setup employing 2 kinds of cellular networks: 4G and non-standalone 5G.

A. Old Setup: Baseline Experiments and Preliminary Results

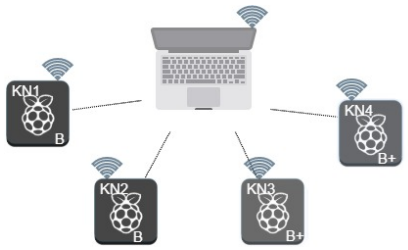

Fig. 1. Preliminary setup

The objective of the first testbed was to simulate an industrial setting where the devices composing the cluster would be heterogeneous and reflecting the ideas of Industry 4.0 [44].

The master node was an HP Elitebook 8560w, while the worker nodes Raspberry Pi minicomputers (models 3B and 3B+) as shown in Figure1. Limitations of this first trial were that: the responsiveness of the cluster was reduced by the WiFi connection. The network setup was over-simplistic since devices were in the same subnet and had static IPs.

TO simulate an intensive work, multiple batches of pods were deployed at the same time: first 2 replicas, then 10, 20, 30 and 40, The application was a small Python Flask web server. For higher numbers of simultaneous pods there was a risk that the cluster would start to resent both schedulers and nodes would shut down just because the requested resources were too high.

To compare the two schedulers we analyzed the pod distribution, the time to schedule and the delta of the temperature of the CPUs pre and post scheduling.

This last metric is an observational indicator for cluster health, since it is a parameter indicating the usage of the nodes not considered in the allocation by our scheduler.

Nodes overheating happened a few times with the default scheduler.

The average temperature and delta of both schedulers resulted similar, with custom scheduler being 64% faster in case of 40 pods deployments.

B. New Improved Setup

To further explore the IIoT use-case, the second cluster was built on top of a real 4G/5G cellular network. This end-to-end verification system is built up by using 5G NSA core, and an Edge network with Kubernetes cluster that operates across the wireless connectivity. The connectivity between core and edge is established via a high-performance 100G router. The core network provides 4G and 5G connectivity with the following main components, installed in a virtualized environment: Evolved packet gateway (EPG), 3GPP compliant Policy and Charging Rules Function (PCRF) [45] and Policy Control Function(PCF), Mobility Management Entity (MME), Home Subscriber Server (HSS), User Database compliant to 3GPP User Data Convergence standard [46]. The setup operates in NSA mode, which means the control plane uses 4G control functions, while the user plane is provided by 5G. The Edge network is distributed, the master node is instantiated in a VM close the core network, while the 4G, and 5G worker nodes are connected to the Kubernetes cluster via 4G and 5G connectivity, respectively. The worker nodes types are Raspberry Pi model 3B and 3B+, two of them can perform 4G attach to the test APN of the core via 4G USB modems (Huawei e3372) and the other two via 5G modem through USB tethering.

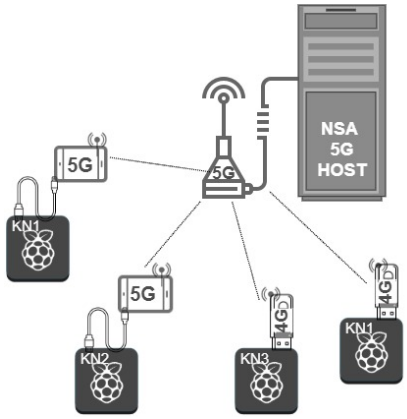

Fig. 2. Second setup

This configuration allows the nodes to have access to the internet (dashed line in Figure 3), for example to access a datacenter or to download container images needed for deployments.

Mobile connectivity, however, introduces in the cluster the issue of Network address translation (NAT) . The mobile nodes are not physically in the same subnet and cannot be pinged by the master on the server nor can they communicate with each other using their internal addresses. This is against basic requirements of a Kubernetes network [34]:

1) any two containers should be able to communicate without NAT
2) any node should communicate with all containers without NAT
3) the IP that a container sees itself as must be the same IP others can use to reach it

As shown in Figure 3 the K8s cluster will have to rely on a Container Network Interface plugin. Pods are connected to the node network namespace with a virtual Ethernet pair: two namespaces with an interface on each end (veth0 in the root node namespace, and eth0 within the pod). Pods in the same node are connected to each other and to the node's eth0 interface via a bridge: docker0. The mapping of virtual IPs to pod IPs within the cluster is coordinated by the kube-proxy process on each node. This process sets up iptables. In our cluster setup we used Flannel [47] as the plugin to configure the layer 3 IPv4 network fabric for Kubernetes.

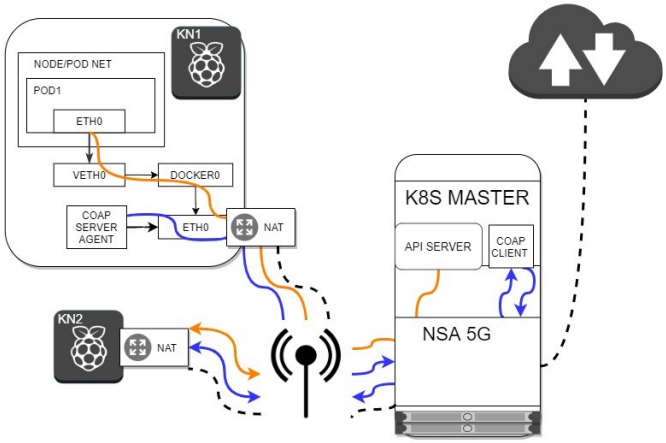

Fig. 3. Network Setup-2

Any deployment or service inside the cluster will not be influenced by NAT since it will follow this virtual network (orange line), however, our monitoring tool being outside K8s namespace, had to be adapted to overcome the NAT (blue line). Our approach consisted in having the Server side on the *Raspberry pis* to send an ACK as soon as the tool starts. Since only the minion nodes are behind the NAT, they can see and have access to the Master node. After the client side on the K8s node receives the ACK, his address will be resolved in the NAT tables of the modems attached to the Raspberry pi. From now on the VM hosting the master node will be able to act as a client as described in Section IV-A.

C. New Test Results

Compared to the previous implementation [1], the current custom scheduler applies an algorithm that makes it more sensitive to environment and application changes. This enabled us to extend the limits of nodes capabilities, which is especially important in the case of Edge Computing.

In the new test setup the number of pods allocated per node is less balanced, as expected from the algorithm of our scheduler (Table I). This results in a variation of the overall cluster temperature that is also less linear: nodes get the chance to cool down and "rest". When actuating the allocation of a significant number of pods, the custom scheduler becomes faster, not only at selecting nodes but also for what matters: the overall completion time of the deployments. In our test cases this improvement in performance showed itself starting

TABLE I
POD ALLOCATION PER NODE

	Allocated pods per node							
	Default scheduler				Custom scheduler			
Total # pods	kn1	kn2	kn3	kn4	kn1	kn2	kn3	kn4
2	1	0	1	0	0	1	1	0
10	2	3	2	3	2	3	3	2
20	5	5	5	5	3	8	6	3
30	7	8	8	7	5	8	10	7
40	10	10	10	10	14	10	8	8
50	13	12	13	12	11	16	14	9
60	-	-	-	-	13	20	15	12

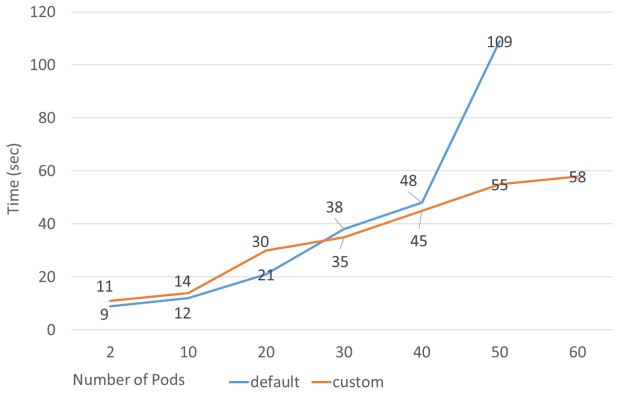

Fig. 4. Time of scheduling deployments

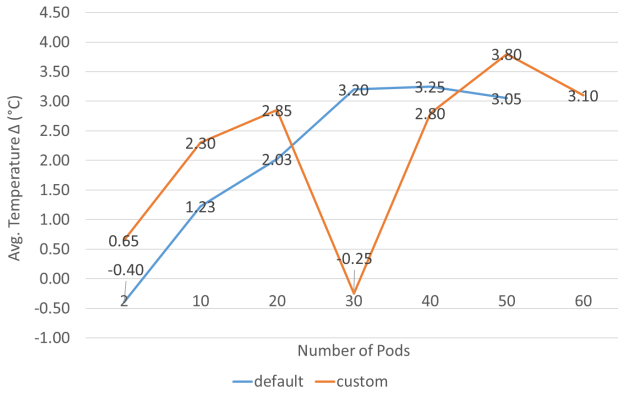

Fig. 5. Average cluster temperature deltas

from the deployment of 30 pods, as visible in Figure 4. This confirms that when the devices are under stress, our scheduler performs better by choosing less overwhelmed nodes. The result becomes even more interesting when considering the allocation of 50 pods onward. In this instance, the time spent by the default scheduler resulted being 109 seconds, against the 55 seconds taken by ours.

The main reason for this is the fact that, while the default scheduler prioritizes strongly the balance in the allocation of the pods (mostly by number), ours tries to preserve a safe operational state of the devices. The default scheduler then had issues optimizing the allocation when the Kubernetes core agent running in the worker nodes started reporting that the devices were in ”risky states”, but since each of them would sporadically report something similar (because a very similar number of pods was being allocated), it found itself having to adjust its decision several times per pod, before actually binding them to a node. All this added to the fact that some nodes probably weren’t capable of reporting themselves correctly to the master node, again due to pressure, as defined in point 5 of Section III.

As further proof for that, the custom scheduler managed to successfully allocate 60 pods on the Raspberry Pis, while the default one made the entire cluster crash (the devices overheated and stopped working), as visible in Table I.

VI. CONCLUSION AND FUTURE WORKS

In this work we improve our scheduler presented in [1]. the current custom scheduler applies an algorithm that makes it more sensitive to environmental and application changes. This enabled us to extend the limits of nodes capabilities, especially scheduling edge-native applications.

A fully functioning 5G and Edge computing network was build, complete with 5G radio and connectivity located in the proximity of the IoT devices. All the LTE virtualized functions are hosted at the same server rack as the Kubernetes cluster, as per the definition of Edge Computing.

Our solution is capable of adjusting jobs allocations over the nodes, balancing not only memory and CPU usage, but also multiple specific network and infrastructure parameters.

Points of improvement regard the speed and efficiency of the scheduling process in case of stressful deployments: starting from 30 pods onward, the application is ready to run earlier for the custom scheduler. At 50 pods the time of scheduling is cut in a half compared to the default scheduler. While the custom scheduler also manages to allocate up to 60 pods, the default one causes all the devices to overheat and shut down even before completing container creation. In the next works we plan to reduce the sampling of the nodes and network status. Each parameter should have a separate sampling rate, adapted according to the application dynamic behavior and previous footprint in the score calculation. The idea is to further shrink down the impact of the metric agents on the nodes and the network, intending to shorten the reaction time of the scheduler.

APPENDIX A
SIMULATING IIOT APPLICATIONS

To perform the tests three different use cases were produced, having the purpose of emulating different main category scenarios that could occur in an Industrial IoT setting.

The use cases are simulated by applications manipulating the 3D model of a car with high polygon count (190426), an object that could be used within the construction pipeline of an automotive industrial plant.

All the use cases represent execution scenarios that are run and combined during the scheduling trials. The purpose is to evaluate how the scheduling decisions are influenced and how some parameters of the nodes, related to their longevity, are being preserved.

1) CPU Load - 3D Model Rotation: This use case has been simulated by generating a lengthy CPU-bound set of operations, to create a load that could strongly influence the scheduling process: the rotation of the 3D model. Since it is a geometric transformation that consists completely of matrix multiplication.

2) Memory Load - Loading a Set 3D Models: The method to perform this simulation consisted of loading what would conceptually be considered as a 3D scene containing multiple copies of the car model used for this test.The number of cars that have been used in this test scenario was limited to 5, for a total memory consumption of 118.45MB. This choice was dictated by the necessity of creating a substantial memory load in an environment that was already constrained, without making a node unusable for scheduling already before the beginning of the tests.

3) Network Load - Streaming a 3D Model File: To simulate this use case, the 3D model file has been sent across the network on which the cluster is running, in order to generate some relevant network load. More in detail, the master node of the cluster acts as server for the worker nodes, which will stream the file from it. This streaming has been cycled to consume network space throughout the scheduling process, and the transmission has been set to happen in chunks of 1MB at a time.

The main reason for choosing the master node as the streaming server was its high capacity network interface (supporting Gigabit Ethernet), which makes it capable of supporting multiple streams of the aforementioned chunk size.

APPENDIX B
TESTING KUBERNETES DEFAULT SCHEDULER

The tests were conducted by performing the deployment of a Deployment object which is configured to issue 40 replicas of a small Python Flask webserver Pod to the cluster.

For the first phase this deployment was run and one or more nodes were “killed” before it completed, to see how the scheduler reacted to the change and how much time passed before it realized what happened. This phase was intended to be independent of the use cases.

For the second phase, the Deployment was created and deleted (to perform a new scheduling from scratch, each time) for every use case, in order to consider 7 total scenarios deriving from all the possible diverse combinations of the 3 use cases:

- CPU load only (1 node);
- Memory load only (1 node);
- Network load only (1 node);
- CPU + Memory load (2 nodes);
- CPU + Network load (2 nodes);
- Memory + Network load (2 nodes);
- CPU + Memory + Network load (3 nodes).

The first test was to measure the behavior of the default scheduler at the most normal circumstances, which means that all the nodes are available at the beginning of the process and no node died during the scheduling process. The “Time taken” row in the following tables represent the time needed to fully complete the scheduling of all 40 replicas.

TABLE II
BEHAVIOR OF THE DEFAULT SCHEDULER IN NORMAL CONDITIONS

Deployment test 1 - Default scheduler				
Nodes	knode1	knode2	knode3	knode4
Time taken	34s			
# Pods	10	10	10	10
CPU temperature	50.5 °C	50.5 °C	51.0 °C	54.5 °C
Memory usage	397MB	383MB	412MB	391MB
Network latency	1.83ms	3.54ms	3.95ms	4.47ms

As it is visible, from Table II, the default scheduler tries to put the same number of pods in each node. That’s its main way of performing load balancing, in the most general case. Thus, this type of result was expected.

The following cases saw the default scheduler of Kubernetes acting the same way even when one or more nodes stopped responding for a short period (both at the beginning and during the scheduling process). The pods assigned to that node would remain in a *Pending* state until the node started responding again, resulting only in a slight increase of the scheduling time. The results are shown in the tables below. The simulation of the dead node was done by disabling its network interface, making it impossible for the master node to communicate with it, in 2 steps: first for 10 seconds, then for 20 seconds. In the following table, *knode4* was ”killed” for the aforementioned amounts of time.

TABLE III
BEHAVIOR OF THE DEFAULT SCHEDULER - 1 NODE DIED FOR 10S

Deployment test 2 - Default scheduler				
Nodes	knode1	knode2	knode3	knode4
Time taken	42s			
# Pods	10	10	10	10
CPU temperature	50.5 °C	49.2 °C	51.2 °C	55.5 °C
Memory usage	389MB	381MB	412MB	396MB
Network latency	2.87ms	3.21ms	3.94ms	8.37ms

TABLE IV
BEHAVIOR OF THE DEFAULT SCHEDULER - 1 NODE DIED FOR 20S

Deployment test 3 - Default scheduler				
Nodes	knode1	knode2	knode3	knode4
Time taken	47s			
# Pods	10	10	10	10
CPU temperature	51.5 °C	51.5 °C	51.5 °C	57.5 °C
Memory usage	400MB	386MB	414MB	399MB
Network latency	3.27ms	2.41ms	2.91ms	5.36ms

The same test was performed by “killing” two nodes for the same two periods of time. The result was a slight increase to 56 seconds overall for the entire scheduling process in the case of the two nodes dying for 10 seconds, visible in Table V. The nodes that were disconnected were *knode3* and *knode4*. In the case of the two nodes getting disconnected for 20 seconds, a very similar situation occurred, with a further increase in scheduling time. It is important to notice that the core temperatures are quite stable across different scenarios.

ACKNOWLEDGMENT

The authors would like to thank Chinmay Gore, Márk László Mikecz and Armand Desztics for their support in the

Lab environment at the garage.

TABLE V
BEHAVIOR OF THE DEFAULT SCHEDULER - 2 NODES DIED FOR 10S

Deployment test 4 - Default scheduler				
Nodes	knode1	knode2	knode3	knode4
Time taken	56s			
# Pods	10	10	10	10
CPU temperature	50.5 °C	51.0 °C	52.1 °C	54.8 °C
Memory usage	402MB	388MB	418MB	399MB
Network latency	3.48ms	2.51ms	2.91ms	4.86ms

TABLE VI
BEHAVIOR OF THE DEFAULT SCHEDULER - 2 NODES DIED FOR 20S

Deployment test 5 - Default scheduler				
Nodes	knode1	knode2	knode3	knode4
Time taken	63s			
# Pods	10	10	10	10
CPU temperature	51.5 °C	52.1 °C	54.2 °C	58.0 °C
Memory usage	402MB	388MB	418MB	399MB
Network latency	2.15ms	3.27ms	2.84ms	4.21ms

REFERENCES

[1] M. Chima Ogbuachi, C. Gore, A. Reale, P. Suskovics, and B. Kovács, "Context-aware k8s scheduler for real time distributed 5g edge computing applications," in *2019 International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, Sep. 2019. doi: 10.23919/SOFTCOM.2019.8903766. ISSN 1847-358X pp. 1–6.

[2] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016. doi: 10.1109/JIOT.2016.2579198

[3] S. T. P. W. Péter Suskovics, Benedek Kovács, "Creating the next-generation edge-cloud ecosystem," *Ericsson Technology Review*, vol. 11, 2019.

[4] M. Satyanarayanan, G. Klas, M. Silva, and S. Mangiante, "The seminal role of edge-native applications," in *2019 IEEE International Conference on Edge Computing (EDGE)*. IEEE, 2019. doi: 10.1109/EDGE.2019.00022 pp. 33–40.

[5] K. Svensson, Boberg, "Distributed cloud – a key enabler of automotive and industry 4.0 use cases," *Ericsson Review*, 11 2018. [Online]. Available: https://www.ericsson.com/en/ericsson-technology-review/archive/2018/distributed-cloud

[6] 3GPP, "3gpp release 15 – the first full set of 5g standards," International Organization for Standardization, Standard 3GPP TR 21.915, 2018. [Online]. Available: https://www.3gpp.org/release-15

[7] V. Medel, C. Tolón, U. Arronategui, R. Tolosana-Calasanz, J. Á. Bañares, and O. F. Rana, "Client-side scheduling based on application characterization on kubernetes," in *Economics of Grids, Clouds, Systems, and Services*, C. Pham, J. Altmann, and J. Á. Bañares, Eds. Cham: Springer International Publishing, 2017. doi: 10.1007/978-3-319-68066-8_13. ISBN 978-3-319-68066-8 pp. 162–176.

[8] K. Kaur, S. Garg, G. Kaddoum, S. H. Ahmed, and M. Atiquzzaman, "Keids: Kubernetes based energy and interference driven scheduler for industrial iot in edge-cloud ecosystem," *IEEE Internet of Things Journal*, pp. 1–1, 2019. doi: 10.1109/JIOT.2019.2939534

[9] C. Wöbker, A. Seitz, H. Mueller, and B. Bruegge, "Fogernetes: Deployment and management of fog computing applications," in *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*, April 2018. doi: 10.1109/NOMS.2018.8406321. ISSN 2374-9709 pp. 1–7.

[10] V. Medel, C. Tolón, U. Arronategui, R. Tolosana-Calasanz, J. Á. Bañares, and O. F. Rana, "Client-side scheduling based on application characterization on kubernetes," in *International Conference on the Economics of Grids, Clouds, Systems, and Services*. Springer, 2017. doi: 10.1007/978-3-319-68066-8_13 pp. 162–176.

[11] V. Medel, O. Rana, J. A. Banares, and U. Arronategui, "Adaptive application scheduling under interference in kubernetes," in *2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing (UCC)*, Dec 2016. ISSN null pp. 426–427.

[12] J. Santos, T. Wauters, B. Volckaert, and F. De Turck, "Towards network-aware resource provisioning in kubernetes for fog computing applications," in *2019 IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 2019. doi: 10.1109/NETSOFT.2019.8806671 pp. 351–359.

[13] C. Chang, S. Yang, E. Yeh, P. Lin, and J. Jeng, "A kubernetes-based monitoring platform for dynamic cloud resource provisioning," in *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*, Dec 2017. doi: 10.1109/GLOCOM.2017.8254046. ISSN null pp. 1–6.

[14] Firmament, "Poseidon scheduler," https://github.com/kubernetes-sigs/poseidon.

[15] K. Péter, R. Anna, T. Melinda, and H. Zoltán, "Designing a decentralized container based fogcomputing framework for task distribution and management," *International Journal of Computers and Communications*, vol. 13, pp. 1–7, 2019.

[16] B. Jennings and R. Stadler, "Resource management in clouds: Survey and research challenges," *Journal of Network and Systems Management*, vol. 23, no. 3, pp. 567–619, 2015. doi: 10.1007/s10922-014-9307-7

[17] A. Laghrissi and T. Taleb, "A survey on the placement of virtual resources and virtual network functions," *"IEEE Communications Surveys & Tutorials"*, 2018. doi: 10.1109/COMST.2018.2884835

[18] M. Seufert, S. Lange, and M. Meixner, "Automated decision making based on pareto frontiers in the context of service placement in networks," in *2017 29th International Teletraffic Congress (ITC 29)*, vol. 1, Sep. 2017. doi: 10.23919/ITC.2017.8064350 pp. 143–151.

[19] B. Kovács, "Parameter estimation of dynamical systems," *PhD Thesis dissertation*, 2011.

[20] F. Voigtländer, A. Ramadan, J. Eichinger, C. Lenz, D. Pensky, and A. Knoll, "5g for robotics: Ultra-low latency control of distributed robotic systems," in *2017 International Symposium on Computer Science and Intelligent Controls (ISCSIC)*. IEEE, 2017. doi: 10.1109/ISCSIC.2017.27 pp. 69–72.

[21] M. Karrenbauer, S. Ludwig, H. Buhr, H. Klessig, A. Bernardy, H. Wu, C. Pallasch, A. Fellan, N. Hoffmann, V. Seelmann *et al.*, "Future industrial networking: from use cases to wireless technologies to a flexible system architecture," *at-Automatisierungstechnik*, vol. 67, no. 7, pp. 526–544, 2019. doi: 10.1515/auto-2018-0141

[22] P. Varga, J. Peto, A. Franko, D. Balla, D. Haja, F. Janky, G. Soos, D. Ficzere, M. Maliosz, and L. Toka, "5g support for industrial iot applications–challenges, solutions, and research gaps," *Sensors*, vol. 20, no. 3, p. 828, 2020. doi: 10.3390/s20030828

[23] P. Hu and J. Zhang, "5g enabled fault detection and diagnostics: How do we achieve efficiency?" *IEEE Internet of Things Journal*, 2020. doi: 10.1109/JIOT.2020.2965034

[24] X. Wu, M. Jiang, C. Zhao, L. Ma, and Y. Wei, "Low-rate pbrl-ldpc codes for urllc in 5g," *IEEE Wireless Communications Letters*, vol. 7, no. 5, pp. 800–803, 2018. doi: 10.1109/LWC.2018.2825988

[25] M. Sybis, K. Wesolowski, K. Jayasinghe, V. Venkatasubramanian, and V. Vukadinovic, "Channel coding for ultra-reliable low-latency communication in 5g systems," in *2016 IEEE 84th vehicular technology conference (VTC-Fall)*. IEEE, 2016. doi: 10.1109/VTCFall.2016.7880930 pp. 1–5.

[26] Y. Hu, M. C. Gursoy, and A. Schmeink, "Relaying-enabled ultra-reliable low-latency communications in 5g," *IEEE Network*, vol. 32, no. 2, pp. 62--68, 2018. doi: 10.1109/MNET.2018.1700252

[27] H. Elazhary, "Internet of things (iot), mobile cloud, cloudlet, mobile iot, iot cloud, fog, mobile edge, and edge emerging computing paradigms: Disambiguation and research directions," *Journal of Network and Computer Applications*, vol. 128, pp. 105–140, 2019. doi: 10.1016/j.jnca.2018.10.021

[28] C. Puliafito, E. Mingozzi, F. Longo, A. Puliafito, and O. Rana, "Fog computing for the internet of things: A survey," *ACM Transactions on Internet Technology (TOIT)*, vol. 19, no. 2, pp. 1–41, 2019. doi: 10.1145/3301443

[29] A. Neal, B. Naughton, C. Chan, N. Sprecher, and S. Abeta, "Mobile edge computing (mec); technical requirements," *ETSI, Sophia Antipolis, France, White Paper no. DGS/MEC-002*, 2016.

[30] M. Giordani, M. Polese, A. Roy, D. Castor, and M. Zorzi, "Standalone and non-standalone beam management for 3gpp nr at mmwaves," *IEEE Communications Magazine*, vol. 57, no. 4, pp. 123–129, 2019. doi: 10.1109/MCOM.2019.1800384

[31] Docker, "Swarm mode key concepts," https://docs.docker.com/engine/swarm/key-concepts.

[32] Google, "Understanding openstack," https://www.redhat.com/en/topics/openstack.

[33] Google, "K8s: Production-grade container orchestration," https://kubernetes.io.

[34] D. K. Rensin, "Kubernetes - scheduling the future at cloud scale," p. All, 2015. [Online]. Available: http://www.oreilly.com/webops-perf/free/kubernetes.csp

[35] E. Truyen, D. Van Landuyt, D. Preuveneers, B. Lagaisse, and W. Joosen, "A comprehensive feature comparison study of open-source container orchestration frameworks," *Applied Sciences*, vol. 9, no. 5, 2019. [Online]. Available: http://www.mdpi.com/2076-3417/9/5/931

[36] C. Dupont, R. Giaffreda, and L. Capra, "Edge computing in iot context: Horizontal and vertical linux container migration," in *2017 Global Internet of Things Summit (GIoTS)*, June 2017. doi: 10.1109/GIOTS.2017.8016218 pp. 1–4.

[37] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014. doi: 10.1109/MCC.2014.51

[38] "Rancher", "K3s: Lightweight Kubernetes," https://k3s.io/.

[39] A. I. Bucur and D. H. Epema, "Local versus global schedulers with processor co-allocation in multicluster systems," in *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 2002. doi: 10.1007/3-540-36180-4_10 pp. 184–204.

[40] V. Medel, O. Rana, J. Á. Bañares, and U. Arronategui, "Modelling performance & resource management in kubernetes," in *Proceedings of the 9th International Conference on Utility and Cloud Computing*, 2016. doi: 10.1145/2996890.3007869 pp. 257–262.

[41] Google, "CoAP — Constrained Application Protocol — Overview," https://coap.technology/.

[42] J. Dizdarević, F. Carpio, A. Jukan, and X. Masip-Bruin, "A survey of communication protocols for internet of things and related challenges of fog and cloud computing integration," *ACM Computing Surveys (CSUR)*, vol. 51, no. 6, p. 116, 2019. doi: 10.1145/3292674

[43] E. Horowitz and S. Sahni, "Computing partitions with applications to the knapsack problem," *J. ACM*, vol. 21, no. 2, pp. 277–292, Apr. 1974. doi: 10.1145/321812.321823. [Online]. Available: http://doi.acm.org/10.1145/321812.321823

[44] M. Brettel, N. Friederichsen, M. Keller, and N. Rosenberg, "How virtualization, decentralization and network building change the manufacturing landscape: An industry 4.0 perspective," *International Journal of Science, Engineering and Technology*, vol. 8, pp. 37–44, 08 2014. doi: 10.5281/zenodo.1336426

[45] 3GPP, *Policy and Charging Rules Function*, ser. TS. 3GPP, Sep. 2009, no. TS23.203, Rel-9 v0.4.0. [Online]. Available: http://www.3gpp.org/ftp/Specs/html-info/23203.htm

[46] 3GPP, *User Data Convergence (UDC); Technical realization and information flows*, ser. TS. 3GPP, Sep. 2009, no. TS23.335, Rel-9 v0.4.0. [Online]. Available: http://www.3gpp.org/ftp/Specs/html-info/23335.htm

[47] Flannel, "Flannel," https://github.com/coreos/flannel\#flannel.

Michael Chima Ogbuachi is a Ph.D. student at the Eötvös Loránd University, with a background in Computer Science and an EIT Digital double master degree (KTH, Stockholm - BME, Budapest) in Embedded Systems. His research interests are on methods and infrastructures for microservices and artificial intelligence.

Anna Reale is a Ph.D. student at the Eötvös Loránd University, with a background in Information Engineering and Computer Science. After her EIT Digital double master degree in Service Design Engineering, she joined an EIT Digital Industrial Ph.D. program In ELTE. Her Ph.D. topic being 5G Edge Computing, she works on code migration, computation offloading and partitioning frameworks.

Péter Suskovics Joined Ericsson in 2007, currently system architect of Cloud technologies and 5G. Holds an M.Sc. in computer science from the Budapest University of Technology and Economics. His current focus is edge computing, cloud native, performance and resiliency, he leads global research & development projects within these areas.

Benedek Kovacs Joined Ericsson in 2005, senior specialist on network service performance. Holds an M.Sc. in information engineering and a Ph.D. in mathematics from the Budapest University of Technology and Economics. Today he researches 5G networks and edge computing and coordinates corresponding global engineering projects.