Evaluating a Digital Twin of an IoT Resource Slice: An Emulation Study using the ELIoT Platform

Fabrizio Granelli, *Senior Member, IEEE*, Riccardo Capraro, Michela Lorandi, and Paolo Casari, *Senior Member, IEEE*

***Abstract*—This paper proposes a realistic setup for a digital twin of an IoT deployment, which makes it possible to measure the impact of IoT networking and computing slices on the physical resources of edge computing hosts. The proposed setup relies on a configurable IoT system, capable of emulating the behavior of a LWM2M broker, along with a large number of devices, both real and emulated. Through our digital twin, we can measure IoT resource utilization and its variation over time, quantifying utilization peaks that may occur at transients, when devices attach to or leave the network.**

***Index Terms*—Network slicing; Internet of Things; digital twin; emulation; virtualization; slice sizing**

I. Introduction and Related Work

Fifth-generation (5G) network slicing is a prominent strategy to make the same network infrastructure available to multiple virtual mobile network operators (VMNOs). Slicing works by virtualizing network resources and allotting a portion of them (called the "slice" in 3GPP jargon, or a logically isolated network partition (LINP) in ITU-T jargon) to a VMNO or a general third-party, such that a service-level agreement (SLA) between the VMNO and the physical network operator is honored at all times [1].

The key enabling technologies to realize slicing are software-defined networking (SDN), network function virtualization (NFV) and multi-access edge computing (MEC). SDN enables the management of network resources through policies defined and updated in software. NFV detaches network functions (e.g., authentication, resource allocation, etc.) from the physical hardware that provides each function, and makes it possible to deploy a function on any sufficiently powerful platform, typically through virtual machines or containers. MEC provides computing power close to the edge of the network. This makes it the ideal resource to deploy virtual network functions (VNFs) that implement low-latency services, or that pre-process data before forwarding them for remote cloud storage [2], [3].

The onset of machine-type communications (MTC) and Internet of things (IoT) network deployments poses special challenges to network operators. IoT networks can be composed of hundred, or even thousands of devices, whose compound traffic can be sizeable and challenging to manage. This is especially true if IoT devices share network resources with other applications with diverse requirements, such as ultra-reliable low-latency communication (URLLC) or broadband streaming applications.

For the above reasons, network operators need actionable data on networking and computation requirements of IoT deployments. This way, they can prepare to modulate allotted resources and support IoT traffic transients without impacting other applications, and without degrading the service offered by IoT applications themselves.

There is considerable interest in the use of MEC resources to deliver IoT services. For example, Husain *et al.* propose an architecture to deploy differentiated IoT services in edge cloud servers [4]. The authors argue that home IoT services can be delivered using mainstream network configurations, whereas massive (e.g., smart city-related [5]) and ultra-low latency (e.g., industrial [2]) IoT services require to virtualize network access and server-side computation and move them to the edge cloud. Theodorou and Xezonaki also consider a massive smart city deployment, and propose the virtualization of multiple IoT gateways through the NFV MANO framework, in order to flexibly allocate computational capacity to different applications [6]. The IoT-oriented architecture described by Kapassa *et al.* also provisions dynamic 5G network resource slices to IoT applications and services [7].

With an eye towards massive IoT deployments, Lekshmi *et al.* advocate slicing as a key 5G network feature, that would enable a flexible allocation of network resources over time [8]. To reduce the requirements of VNFs composing a slice, Ouedraogo *et al.* propose "flyweight" network functions, which reduce the storage and memory footprint of a VNF and its deployment overhead, at the price of losing isolation among VNFs deployed on the same physical machines [9].

Several contributions focus on slicing in long range wide area network (LoRaWAN) IoT deployments. In this case, slice resources are modulated by changing LoRa's transmission parameters in order to achieve the best QoS and agent experience. For example, Dawalibi *et al.* employ game theory to achieve the above [10], whereas Messaoud *et al.* design a deep learning approach to automatically tune LoRa's spreading factor and transmission power [11].

To assess the effectiveness of slicing for an IoT deployment, key aspects are modeling and estimating the amount of networking and computation resources that such deployment requires. While some of the above works model IoT requirements at least by distinguishing, e.g., low-latency vs. delay-tolerant applications, a clear evaluation of the transient and

Manuscript received April 15, 2021; revised June 4, 2021; accepted July 11, 2021. This work was supported in part by the Italian Ministry for University and Research under the initiative "Departments of Excellence" (Law 232/2016). The associate editor coordinating the review of this paper and approving it for publication was N. Passas. *(Corresponding author: P. Casari)*

All authors are with the Department of Information Engineering and Computer Science (DISI), University of Trento, 38123 Povo (TN), Italy (email: paolo.casari@unitn.it).

Digital Object Identifier XX.XXXX/XXXXXXXXXXX

long-run requirements of IoT slices is still missing in the literature.

In this letter, we intend to fill this gap by introducing the concept of *digital twin* [12] of the IoT service, and by using such digital twin to estimate the amount of networking and computing resources an IoT service requires during typical work cycles. The digital twin is built by exploiting the emulated IoT platform (ELIoT) [13] framework. ELIoT is part of a recent trend [13]–[15] that emulates network deployments using realistic virtualization technologies, as would be employed in production networking and computing systems. For our study, this enables the emulation of large IoT deployments while concentrating on network-side management aspects.

ELIoT provides the implementation of well known protocols for IoT and the deployment of a lightweight machine-to-machine (LWM2M) server, along with sample devices represented in LWM2M/IP for smart objects (IPSO) formats. Yet, for a digital twin to be an effective tool, it needs to be automatically deployed and configured in an edge server orchestration scenario. In this work, we deploy the digital twin in a realistic edge server configuration including typical toolchain elements such as Docker, Kubernetes, Helm and Grafana. Using such environment, the network manager can realistically analyze real-time as well as "what-if" scenario implementations to predict requirements on edge-side computation, network communications as well as system memory for different types of IoT deployments. Our system also enables researchers to easily build realistic performance and resource consumption datasets as a basis to train machine learning (ML)-based and artificial intelligence (AI)-based prediction and adaptation schemes, including automated methods to optimize resource allocation to the corresponding IoT slices.

The letter is organized as follows: in Section II we detail our evaluation setup; we then describe our results in Section III, provide a summary discussion in Section IV and draw concluding remarks in Section V.

II. Evaluation Setup

A digital twin needs to represent and closely replicate the behavior of its "real-life" counterpart. Therefore, for the case of an IoT service, we decided to use an emulator that can generate realistic traffic, e.g., between IoT devices and the IoT broker. To achieve this goal, we extended the original ELIoT project by deploying the system in Kubernetes (K8s), and by adding monitoring capabilities via Prometheus [16] and Grafana [17]. We containerize ELIoT components using Docker and run it on a local K8s cluster using Kubernetes in Docker (KIND) [18]. It is worth mentioning that this digital twin architecture also enables the integration of real sensors via TCP/IP connections. However, for the purpose of this work, we decided to run only emulated sensors, which enables the easier scaling of device numbers and traffic characteristics.

We elect to use K8s as it makes it possible to orchestrate containerized applications: it automates the deployment and connectivity of an application by specifying all required parameters in configuration files. Moreover, KIND provides a set of scripts written in the Go language to run a K8s cluster within a Docker container. The scripts implement cluster creation and deletion. In addition to this, KIND exposes a command-line interface for cluster management.

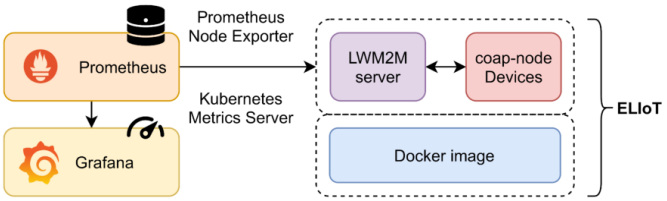

Fig. 1. Monitoring system architecture and integration with ELIoT

Fig. 2. Snapshot of our Grafana dashboard.

Prometheus is a monitoring system based on a time series database, which collects metrics from specified targets at predefined intervals. It also provides a query language that automates the evaluation of rule expressions and the retrieval of specific metrics. Prometheus easily integrates with Grafana, a tool that exposes dashboards encompassing time series graphs and charts for the visualization of metrics and other related information including, e.g., CPU and memory utilization, container startup time, etc.

To extract the information about K8s resources we employ the Metrics Server, whereas for machine utilization data we resort to the Prometheus Node Exporter. K8s's Metrics Server is a source of container resource metrics used that K8s leverages for autoscaling purposes. The Metrics Server enables the collection of resource metrics from kubelet agents, and exposes such metrics in the K8s API. The Prometheus Node Exporter, instead, makes it possible to retrieve hardware and OS metrics.

Our Grafana dashboard, developed starting from an established dashboard for Docker [19], allows us to monitor the system at runtime, and to execute retrospective analysis. The dashboard keeps track of system statistics such as the number of running containers, the network traffic, as well as memory and CPU utilization. We persist data on disk and log timestamps of test execution steps. Moreover, the dashboard allows us to easily export information collected during our tests, in order to post-process them. This data can also be directly queried using Prometheus. The architecture of the monitoring system is summarized in Fig. 1. In Fig. 2 we provide a visual example of a Grafana dashboard. Our setup involves open-source components, which makes the results presented in this paper fully reproducible. To favor this, we provide our code and configurations, along with full instructions for the installation of our framework, at our GitHub repository (https://github.com/ricCap/ELIoT).

TABLE I
RESOURCE REQUIREMENTS PER EMULATED IOT DEVICE TYPE

	Device type		
Resource	`Low`	`Medium`	`High`
RAM (idle)	70 MByte	70 MByte	70 MByte
Network traffic out	65 Byte/s	125 Byte/s	1.2 kByte/s
Network traffic in	49 Byte/s	180 Byte/s	923 Byte/s
Data generation period	2 s	700 ms	100 ms

TABLE II
EMULATION RESULTS PER APPLICATION TYPE

	Application type		
Metric	APP1	APP2	APP3
RAM (LWM2M server)	1.6 GByte	1.4 GByte	4.5 GByte
RAM (LWM2M server + dev.)	14 GByte	9.4 GByte	9.3 GByte
RAM (Monitoring)	15 GByte	9.3 GByte	5 GByte
LWM2M input network traffic	13 kByte/s	13 kByte/s	61 kByte/s

We run the ecosystem on a dedicated workstation with an AMD Ryzen Threadripper 3990X CPU (offering 64 hyperthreaded cores), 128 GByte of RAM, and a 512 GByte storage unit. We test our system by deploying the monitoring infrastructure on a K8s cluster using KIND, and subsequently test different ELIoT deployments; we oversee the tests using our dashboard, and finally evaluate the results by extracting data from Prometheus and Grafana.

In the following, we will also refer to the ELIoT LWM2M server using the common term "broker." For our evaluation, we implement three types of sensors (`Low`, `Medium`, `High`) with three different data generation patterns, as summarized in Table I. We observe the devices using the Observe/Notify mechanism implemented by the broker. In ELIoT, this results in a symmetric exchange of messages between the devices and the broker. We then combine the three types of devices into applications that simulate real world scenarios: **APP1**: an "environmental monitoring" application, with 200 `Low`-type devices; **APP2**: a "road monitoring" application, encompassing 100 `Medium`-type devices **APP3**: an "intensive monitoring" application, featuring 50 `High`-type devices.

In the charts and tables that follow, we only consider one-way traffic for the broker; different applications that interact with the broker may have different communication patterns.

III. RESULTS

A. System evaluation

We start with Table II, which provides a summary of the metrics obtained from the digital twin for the three application types defined above. For this evaluation, we obtain the metrics' values at exactly 5 minutes after all components are deployed: at this time, the management operations of the system are completed, and the related values have stabilized. Because of this, we obtained negligibly different values in two subsequent tests: the results in Table II are the averages of these two runs.

The results focus on the amount of RAM required for each application and on the amount of network traffic at the input of the LWM2M server. We observe that the server RAM allocation is reasonably limited to less than 2 GBytes for both APP1 and APP2. Only for APP3 does the RAM requirement increase to 4.5 GBytes. This is due to the caching of API calls, which are much more frequent for APP3 than for the other two applications.

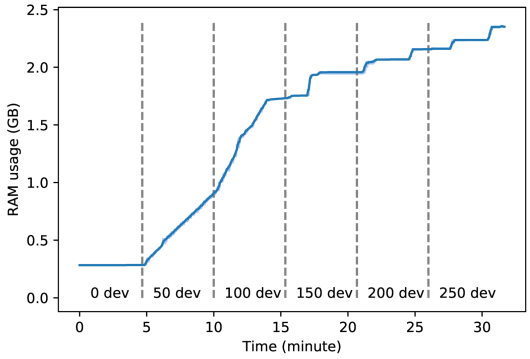

Fig. 3. Broker RAM usage for an increasing number of `Medium` devices.

Table II also reports the amount of RAM required to emulate the devices and to support data collection in the monitoring system. Emulating the 200 `Low` devices of APP1 requires a total of about 14 GBytes, whereas APP2 and APP3 require only slightly more than 9 GBytes. As the three applications feature 200, 100, and 50 devices, respectively, the monitoring system requires correspondingly less memory resources: 15 GBytes, 9.3 GBytes, and 5 GBytes, respectively. The input network traffic seen at the LWM2M server also depends on the application. As each of the 100 `Medium` devices of APP2 generates 125 Bytes/s, the LWM2M server sees about 13 kBytes/s of traffic. Similarly, APP3 generates about 60 kBytes/s of traffic.

We further inspect the RAM and network utilization requirements by considering the amount of system resources that the broker seizes as a function of the number of deployed devices. Fig. 3 shows the RAM usage as an average of five different runs (solid blue line). A light-blue band conveys the minimum and maximum RAM usage values throughout all five runs. For this test, we deploy 50 devices simultaneously every 5 minutes. The average usage increases from a minimum of about 300 MBytes up to 2.5 GBytes when 250 devices are deployed. The RAM increase is linear at the beginning, whereas further batches of 50 devices lead to step-like memory increases that occur with some delay with respect to our instantiation command. This delay is due to the time it takes to complete device spawning, for the devices to register with the broker, and for monitoring to start in the broker when all devices have completed their startup and registration process.

In Fig. 4, we show how increasing the number of deployed devices affects the network traffic. Again, the solid blue line conveys the average traffic over time, computed as an average over five runs, whereas the light-blue band conveys the minimum and maximum values measured in our tests. Here, devices appear in batches of 50, about every 5 minutes. We observe an initial traffic spike as devices bootstrap and set up a connection with the LWM2M server, after which the network traffic stabilizes to the expected level of 125 Bytes/s times the

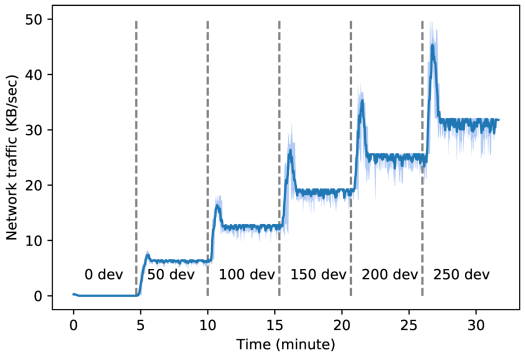

Fig. 4. Broker network traffic for an increasing number of `Medium` devices.

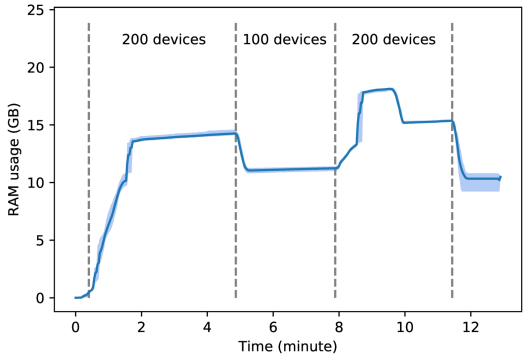

Fig. 5. ELIoT (broker + devices) RAM usage during transition phases.

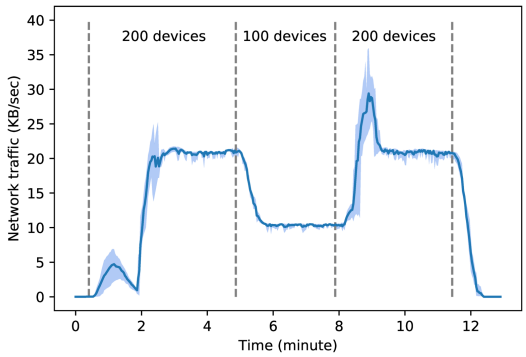

Fig. 6. ELIoT (broker + devices) network traffic during transition phases.

number of deployed `Medium`-type devices. The minimum and maximum values remain very close to the average, as different executions with the same storyboard yield very coherent results.

We now put the system under stress through a different setup. Here, we schedule the deployment of 200 `Low`-type devices at the start of the emulation, after all the monitoring containers are running. This ensures that the time required to pull containers images does not affect the simulation statistics.

Fig. 5 shows the RAM utilization over time, including both the average over five executions and the minimum/maximum span. We observe that the RAM requirements tend to increase rapidly and peak at about 15 GBytes on average after spawning 200 devices. We then decrease the number of devices from 200 to 100 (whereby the RAM usage decreases to about 11 GBytes) and then increase them back to 200 devices, showing that the amount of RAM required by the system follows the number of scheduled devices accurately. After the latter operation, the amount of allocated RAM increases to about 18 GBytes due to API call caching. We finally remove all devices from the emulation system after slightly more than 11 minutes from the start. In line with previous results, the system's performance is predictable in steady-state conditions. The data show a slightly greater variation in the device deployment and removal phases, where system metrics change faster over time, and each specific realization may yield a different resource allocation, depending on the way the system schedules the components of the emulated IoT deployment. The min-max interval confirms this statement.

We conclude our evaluation with Fig. 6, where we provide the evolution of the network traffic over time in the same conditions of Fig. 5. In line with Fig. 3, we observe that increasing the number of deployed devices implies a larger network traffic burden. For the two periods where 200 devices are deployed, the amount of traffic coherently settles around 20 kByte/s, and decreases to slightly more than 10 kByte/s when we transition to 100 devices. As before, the min-max interval shows that all executions behave coherently, and that only when we start deploying or removing devices does traffic greater more statistical variability.

B. The case of a real IoT localization network

We now use our framework to implement a digital twin of the IoT localization testbed deployed at the Department of Information Engineering and Computer Science of the University of Trento, Italy. The testbed has been recently extended to a total of 130 ceiling-mounted stations spanning two floors over two different buildings. Each station embeds different low-power wireless platforms offering communication options, e.g., via ultra-wideband (UWB), bluetooth low energy (BLE), as well as IEEE 802.15.4.

Here, we consider the UWB-based localization protocol TALLA [20], where ceiling-mounted nodes act as anchors, and mobile UWB tags initiate the localization process. The latter is centralized: a broker collects messages from the anchors and computes the location of each tag that sends a localization request. Each testbed node reports to the broker of the localization service in one of three cases: (i) when it receives a localization request from a UWB tag; (ii) when it broadcasts a synchronization message to neighboring anchors; or (iii) when it receives such a message from another anchor. UWB tags wishing to be localized send messages four times per second. Synchronization messages are sent three times per second by 25% of the anchors. Each of the messages sent to the broker in any of the above cases has a size of 120 Bytes. For additional testbed details, as well as for the design of the TALLA location system, we refer the reader to [20].

Fig. 7 shows the result of the IoT localization testbed's digital twin, including the total network traffic (top panel) and the broker RAM usage (bottom). We tested the system with 10 and 20 users trying to localize themselves. We observe that more users relate to a higher network traffic to transmit localization data. The difference between 10 and 20 users is minimal, because the IoT nodes exchange at least synchronization messages, which always generate traffic for

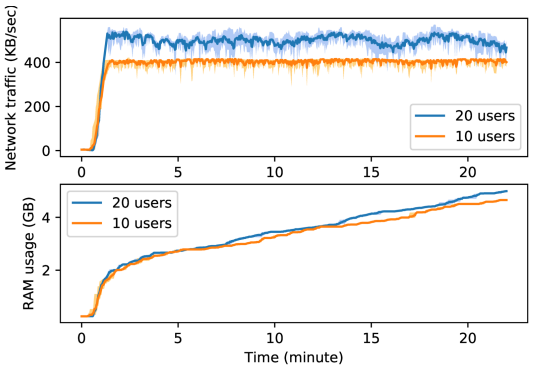

Fig. 7. Results for the digital twin of an IoT localization network. (Top) Total system (broker + devices) network traffic. (Bottom) Broker RAM usage.

the broker. A similar observation holds for the RAM usage at the broker, which increases over time due to the caching of API calls, in line with previous observations.

IV. Discussion

Our results suggest that the most impacted system resource is the RAM. This is mainly due to the number of emulated devices, as the broker caches all API calls related to device communication and data reporting until the end of the emulation. Therefore, applications with more devices and with higher data generation rates fill RAM faster over time. Fig. 3 supports this claim: in fact, the required amount of RAM increases by about 1 GByte when we scale up to 100 devices, and by 2.25 GBytes when the system transitions from 100 to 200 devices. Fig. 5 also shows how decreasing the number of devices frees the memory allocated to the devices, but has no effect on the API calls stored in the broker.

We also remark that the system can scale to simulate a larger number of devices. We made tests with 500 devices: by configuring Docker and K8s to guarantee that no resource allocation limits interfere with the emulation, our workstation can handle such deployments. We can support up to 1000 devices, mostly due to the limits of Docker bridge.

Finally, our setup has a monitoring overhead, as it is likely the case in any production deployment. In our configuration, at least 50% of the RAM allocated to monitoring tools stores system data that was unnecessary in this evaluation. Such configuration can be tuned to extract only the information displayed in the dashboard. Moreover, running Grafana is not mandatory during the emulation.

V. Conclusions

In this paper, we set up a digital twin to analyze and forecast the behavior of an IoT computing and networking slice. Our system supports different kinds of applications, which may encompass different types of IoT devices. The digital twin emulates IoT network components through the ELIoT framework, deployed in Kubernetes by using Docker and KIND, and monitored through Prometheus and Grafana.

Deploying the digital twin makes it possible to measure the amount of network and RAM resources to be allocated to different IoT applications. We observed that the measured resource usage is very coherent across different runs, and tends to exhibit greater variability only during transition phases. This makes our digital twin ideal to size resource provisioning prior to physical deployment, to perform "what-if" studies, as well as to inform quality management policies that uphold application requirements, possibly supported by machine learning algorithms.

Acknowledgments

The authors would like to thank Valentina Odorizzi for her help with the evaluation of the ELIoT emulator, as well as Davide Vecchia and Gian Pietro Picco for their help with traffic measurements in DISI's testbed.

References

[1] V. Sciancalepore, F. Cirillo, and X. Costa-Perez, "Slice as a Service (SlaaS) optimal IoT slice resources orchestration," in *Proc. IEEE GLOBECOM*, Singapore, 2017, pp. 1–7.

[2] A. E. Kalør, R. Guillaume, J. J. Nielsen, A. Mueller, and P. Popovski, "Network slicing in industry 4.0 applications: Abstraction methods and end-to-end analysis," *IEEE Trans. Ind. Informat.*, vol. 14, no. 12, pp. 5419–5427, 2018.

[3] J. Martín-Pérez, L. Cominardi, C. J. Bernardos, A. de la Oliva, and A. Azcorra, "Modeling mobile edge computing deployments for low latency multimedia services," *IEEE Trans. Broadcast.*, vol. 65, no. 2, pp. 464–474, 2019.

[4] S. Husain, A. Kunz, A. Prasad, K. Samdanis, and J. Song, "Mobile edge computing with network resource slicing for Internet-of-Things," in *Proc. IEEE WF-IoT*, Singapore, 2018, pp. 1–6.

[5] F. Zhou, P. Yu, L. Feng, X. Qiu, Z. Wang, L. Meng, M. Kadoch, L. Gong, and X. Yao, "Automatic network slicing for IoT in smart city," *IEEE Wireless Commun.*, pp. 2–9, 2020, in press.

[6] V. Theodorou and M. E. Xezonaki, "Network slicing for multi-tenant edge processing over shared IoT infrastructure," in *Proc. IEEE NetSoft*, 2020, pp. 8–14.

[7] E. Kapassa, M. Touloupou, P. Stavrianos, and D. Kyriazis, "Dynamic 5G slices for IoT applications with diverse requirements," in *Proc. IoTSMS*, Valencia, Spain, 2018, pp. 195–199.

[8] S. S. Lekshmi, M. S. Anjana, B. B. Nair, D. Raj, and S. Ponnekanti, "Framework for generic design of massive IoT slice in 5G," in *Proc. WiSPNET*, Chennai, India, 2019, pp. 523–529.

[9] C. A. Ouedraogo, S. Medjiah, C. Chassot, and J. Aguilar, "Flyweight network functions for network slicing in IoT," in *Proc. SaCoNeT*, El Oued, Algeria, 2018, pp. 31–36.

[10] S. Dawaliby, A. Bradai, and Y. Pousset, "Distributed network slicing in large scale IoT based on coalitional multi-game theory," *IEEE Trans. Netw. Service Manag.*, vol. 16, no. 4, pp. 1567–1580, 2019.

[11] S. Messaoud, A. Bradai, O. Ben Ahmed, P. Quang, M. Atri, and M. S. Hossain, "Deep federated Q-learning-based network slicing for industrial IoT," *IEEE Trans. Ind. Informat.*, pp. 1–1, 2020, early access.

[12] R. Minerva, G. M. Lee, and N. Crespi, "Digital twin in the IoT context: A survey on technical features, scenarios, and architectural models," *Proceedings of the IEEE*, vol. 108, no. 10, pp. 1785–1824, 2020.

[13] A. Mäkinen, J. Jiménez, and R. Morabito, "ELIoT: Design of an emulated IoT platform," in *Proc. IEEE PIMRC*, Montreal, Canada, 2017, pp. 1–7.

[14] N. Ly-Trong, C. Dang-Le-Bao, and Q. Le-Trung, "Towards a large-scale IoT emulation testbed based on container technology," in *Proc. IEEE ICCE*, 2018, pp. 63–68.

[15] T. Kuroiwa, Y. Aoyama, and N. Kushiro, "Automatic testing environment for virtual network embedded systems," in *Proc. IEEE ICCE*, 2020, pp. 1–3.

[16] "Prometheus: monitoring system and time series database," accessed: June 2021. [Online]. Available: https://prometheus.io/

[17] "Grafana: the open observability platform," accessed: June 2021. [Online]. Available: https://grafana.com/

[18] "Kind: running local kubernetes clusters using Docker container nodes," accessed: June 2021. [Online]. Available: https://kind.sigs.k8s.io/

[19] *Docker monitoring dashboard*, accessed: Jun. 2021. [Online]. Available: https://grafana.com/grafana/dashboards/893

[20] D. Vecchia, P. Corbalán, T. Istomin, and G. P. Picco, "TALLA: Large-scale TDoA localization with ultra-wideband radios," in *Proc. IPIN*, 2019, pp. 1–8.