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.2020.3045632, IEEE Access

IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

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

*Digital Object Identifier 10.1109/ACCESS.2017.DOI*

# From Cloud to Edge: Seamless Software Migration at the Era of the Web of Things

**CRISTIANO AGUZZI[1], LORENZO GIGLI[1], LUCA SCIULLO[1], ANGELO TROTTA[1] and MARCO DI FELICE[1,2]**

[1]Department of Computer Science and Engineering, University of Bologna, Italy
[2]Advanced Research Center on Electronic Systems "Ercole De Castro", University of Bologna, Italy

Corresponding author: Cristiano Aguzzi (e-mail: cristiano.aguzzi@unibo.it).

This work has been funded by INAIL within the BRIC/2018, ID=11 framework, project MAC4PRO ("Smart maintenance of industrial plants and civil structures via innovative monitoring technologies and prognostic approaches")

**ABSTRACT** The Web of Things (WoT) standard recently promoted by the W3C constitutes a promising approach to devise interoperable IoT systems able to cope with the heterogeneity of software platforms and devices. The WoT architecture envisages interconnected IoT scenarios characterized by a multitude of Web Things (WTs) that interact according to well-defined software interfaces; at the same time, it assumes static allocations of WTs to hosting devices, and it does not cope with the intrinsic dynamicity of IoT environments in terms of time-varying network and computational loads. In this paper, we extend the WoT paradigm for cloud-edge continuum deployments, hence supporting dynamic orchestration and mobility of WTs among the available computational resources. Differently from state-of-art Mobile Edge Computing (MEC) approaches, we heavily exploit the W3C WoT, and specifically its capability to standardize the software interfaces of the WTs, in order to propose the concept of a Migratable WoT (M-WoT), in which WTs are seamlessly allocated to hosts according to their dynamic interactions. Three main contributions are proposed in this paper. First, we describe the architecture of the M-WoT framework, by focusing on the stateful migration of WTs and on the management of the WT handoff process. Second, we rigorously formulate the WT allocation as a multi-objective optimization problem, and propose a graph-based heuristic. Third, we describe a container-based implementation of M-WoT and a twofold evaluation, through which we assess the performance of the proposed migration policy in a distributed edge computing setup and in a real-world IoT monitoring scenario.

**INDEX TERMS** Web of Things (WoT), Edge/Cloud computing, Service Migration, Software architecture, Performance Evaluation.

## I. INTRODUCTION

The impressive growth of the Internet of Things (IoT) in terms of devices connected and of data produced can be explained by the versatility of its paradigm, which applies to a wide range of different use-cases, from digital manufacturing to smart cities and environmental monitoring [1]. In such domains, service mobility has gained considerable interest for various purposes. On the one side, several large-scale IoT applications operate in dynamic environments: consequently, software solutions are requested to adapt to rapid changes in the bandwidth/computational resources, in the number of connected devices, and in service requirements. Several IoT platforms like [2] [3] provide such a layer of adaptation by supporting the seamless software mobility among the nodes of an edge-cloud continuum. On the other side, mobile IoT devices generating space/time-variant data streams are further pushing the research towards flexible computational architectures able to self-configure to meet the Quality of Service (QoS) for the user applications [4]. This is the case of Mobile Edge Computing (MEC) [5] architecture (and closely related concepts such as Cloudlet [6], Fog Computing [7], and Follow Me Cloud [8]) that aim at running processing tasks in the proximity of the data sources. A core component of MEC architectures is the capability of offloading the software services on the edge/fog servers closest to the current user position [5], often by means of container/Virtual Machines (VMs) mobility techniques [9] [10], and of migration policies driven by the physical mobility of the IoT devices [11].

Service migration is not the only open challenge in the



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.2020.3045632, IEEE Access



IoT landscape, which comprises an uncountable number of protocols, stacks, and cloud ecosystems. Indeed, most of the IoT environments are characterized by the heterogeneity of hardware and software components, as well as by the dynamicity of their interactions. Interoperability issues are estimated to reduce up to of 40% the potential revenues [12]. At the same time, novel business opportunities can swell by enabling different IoT systems to communicate together [12]. Although cloud ecosystems can mitigate some of the interoperability issues by means of Web technologies (i.e., REST APIs, JSON, Web Sockets, etc.), they are often based on silos architectures with implicit or explicit vendor lock-in [12] [13] [14]. Furthermore, such solutions employ a sensor-to-cloud approach where devices are managed through cloud-based connectivity, again at the expense of limited extensibility. The Web of Things (WoT) standard [15] by the W3C consortium represents a recent and promising solution to unlock the potential of the IoT by enabling interoperability across IoT platforms. The interoperability support is managed at the application level by defining a standard interface for IoT components (physical or virtual), known as the Thing Description (TD) [15], which formally states the Web Things' (WTs) capabilities or affordances. Despite the recent appearance, some interesting applications of the W3C WoT have been proposed so far on different IoT domains [16] [17] [18] [19] [20]. At the same time, the WoT reference implementation [21] does not support mobility of software components among cloud/edge nodes, since the runtime environment of a WT (called *Servient* in [21]) must be statically deployed on a device.

In this paper, we address the two IoT issues previously mentioned (i.e., service migration and service interoperability) from a WoT perspective: more specifically, we aim at extending the WoT potential for dynamic IoT environments by supporting dynamic orchestration and mobility of WTs among the available computational resources of the full IoT spectrum (edge/fog/cloud nodes). The WT migration offers novel opportunities compared to existing software mobility approaches in the MEC literature e.g. [5], [22]. Indeed, since the interactions among the WTs are described through uniform software interfaces (i.e., the TDs), fine-grained and adaptive allocation policies can be engineered; such policies can migrate groups of WTs to meet system-wide QoS requirements by taking into account the real-world network and computational load conditions, and with much lower implementation complexity for the service monitoring than other ad-hoc solutions. At the same time, the mobility of a WT from one node to another might impact the operations of other WTs that were using it. Therefore, novel solutions must be deployed to manage the WT handoff and to guarantee system consistency. This paper addresses research questions related to both WT *migration mechanisms* and WT *migration policies*, i.e.:

- How to enable the seamless migration of a WT between two nodes?
- How to optimize the performance of a WoT deployment by orchestrating the WT allocations on a cloud-edge continuum?

To address the issues above, we propose the Migratable Web of Things (M-WoT), a novel architectural framework supporting the dynamic allocation of W3C WTs to the available computational nodes. Specifically, we investigate how to enable the *stateful* migration of WTs by managing the handoff procedure on WT consumers. At the same time, we envisage the presence of a WoT Orchestrator service, which is in charge of monitoring the interactions among the WTs, and of computing the optimal allocation of WTs to nodes, based on high-level policies (e.g. data locality maximization, latency minimization, etc). More in detail, three main contributions are provided by this study:

- We discuss the advantages of WT migration mechanisms on two selected IoT use-cases, and then the components of the M-WoT software architecture.
- We formulate the WT allocation as a multi-objective optimization problem. Then, we propose a centralized heuristic that aims at balancing the inter-host communication load (generated by the interactions among WTs) and the computational load of each host.
- We validate the M-WoT operations through two testbeds. First, we evaluate the performance of different allocation policies on edge computing scenarios where we vary the number of WTs and the interactions among them. Second, we investigate the effectiveness of the M-WoT framework on a generic IoT monitoring scenario where real-time diagnostic services are dynamically migrated from cloud to edge nodes based on context conditions.

The evaluation analysis demonstrates that the proposed heuristic can balance the inter-host communication and the computational load in an effective manner when compared to greedy policies. Moreover, in the IoT monitoring use-case, the M-WoT solution is able to effectively reduce the diagnostic latency compared to a state-of-art, no-migrate approach.

The remainder of this paper is structured as follows. Section II reviews the IoT service migration approaches and the W3C WoT architecture. Section III highlights the novelties of the M-WoT framework and its suitability on selected IoT use-cases. Section IV describes the M-WoT architecture and its enabling components. Section V discusses the WoT deployment as a multi-goal optimization problem and proposes a graph-based heuristic for the allocation of WTs to nodes. Section VI sketches the actual M-WoT implementation. The experimental results are presented in Section VII. Section VIII draws the conclusion and discusses the future works.

## II. RELATED WORKS

At the best of our knowledge, the problem of dynamic allocation and live migration of WTs can be considered relatively new in the literature on WoT systems. At the same time,



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.2020.3045632, IEEE Access



there are plenty of scientific papers addressing the migration of software services between edge/cloud nodes in order to support the physical mobility of the IoT devices. For this reason, we split the related works in two Sections. First, in Section II-A, we briefly review the W3C Web of Things (WoT) architecture, motivated by the newness of the standard and by the need of introducing the terminology used along the paper; also, we discuss the (few) tools and applications developed so far. Then, in Section II-B we review architectures and enabling technologies for service migration in the IoT, by mainly focusing on Mobile Edge Computing (MEC) approaches.

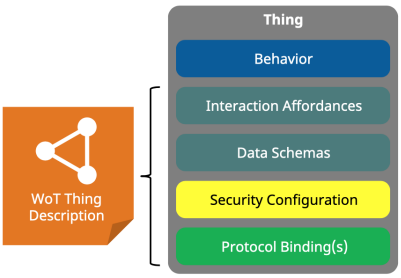



**FIGURE 1.** The W3C WoT architecture proposed in [15].

### A. W3C WEB OF THINGS

The W3C WoT group started its activities in 2015 with the aim of defining a reference set of standards enabling interoperability across different IoT systems [15]. The core of the proposal is the definition of the Web Thing (WT), which represents the abstraction of a physical or a virtual entity whose metadata and interfaces can be formally described by a WoT Thing Description (TD) [15]. The architecture of a WT includes four building blocks of interest: the *Interaction Affordances*, the *Security Configuration*, the *Protocol Bindings*, and the *Behaviour*, as depicted in Figure 1. The first three blocks are included in the TD: the latter can be defined as a sequence of standardized, machine-understandable metadata allowing consumers to discover and interpret the capabilities of a WT in order to interact with it. More in details:

- The *Interaction Affordances* (or simply Affordances in the following) provide an abstract model of how consumers can interact with the WT, in terms of properties (i.e. the state variables of the WT), actions (i.e. commands that can be invoked on the WT) and events (i.e. notifications sent by the WT).
- The *Protocol Bindings* define the mapping between the abstract Affordances and the network strategies (e.g. the protocols) that can be used to interact with the WT.
- The *Security Configuration* defines the access control mechanisms of the Affordances.

The TD can be encoded by means of the JSON-LD language, hence embedding also semantic description [23]. Finally, the *Behaviour* is the implementation of the WT, including the Affordances, e.g. the code of its actions. All the blocks above are executed within a software runtime named *Servient*, which can indifferently act as a Server or as a Client. In the first case, the Servient is said to host and *expose* Things, i.e. it creates a run-time object serving the requests towards the hosted WT, like accessing the exposed properties, actions and events. In the second case, the Servient is said to *consume* Things, i.e. it processes the TD, generates a run-time representation called Consumed Thing, and makes it available to those clients that are interacting with the remote WT.

Due to the recent appearance of the standard, the literature on W3C WoT is still scarce and limited to few applications and supporting tools. The mapping of real-world IoT devices to W3C WTs have been explored in [16], [17], [18] and [19], respectively for the case of mobile phones, automotive industry and wireless sensor networks. Specifically, in the latter, the authors demonstrate how to deploy WoT-based interoperable sensing applications able to manage heterogeneous sensors equipped with three different wireless access technologies (Wi-Fi, BLE and Zigbee). Regarding the tools, beside the ones implementing the W3C WoT Servient on different programming languages (e.g. `Javascript` in [21], `Python` in [24]), we cite the WoT Store platform [20] that supports seamless management of WTs and mash-up applications able to consume multiple, heterogeneous WTs at the same time.

### B. IOT SERVICE MIGRATION

A multitude of solutions has been proposed to enable the seamless service migration among nodes of a distributed system. We can classify the existing approaches into two broad categories: static migration vs dynamic migration techniques. In the first case, software migration is used as a synonym for software modernization, i.e. the process of adapting the capabilities of an existing system in order to be deployed on a new operating environment; the reader can refer to [25] for an exhaustive survey on the migration of legacy systems toward cloud-based software. The second case (i.e. dynamic migration) refers to the process of offloading the run-time execution of software services from one node to another. We focus on the dynamic case since it is more relevant for the scope of the paper. For the IoT domain, we can further distinguish between user-induced and mobility-induced migration approaches. The first case includes several different studies on how to enable mobile applications to seamlessly migrate between nodes during their normal operation [22]. The final goal is to offer the best Quality of Experience (QoE) to users while they switch from one device to another. To this aim, in [26], the authors describe the TRAMP middleware for the fine-grained mobility of multimedia applications; the migration decision is manually defined by the users. In mobility-induced approaches, software migration is achieved by ensuring that the data management/processing is always occurring as close as possible to the current device location. Such a conceptual model is generally denoted as Mobile Edge Computing (MEC) [5], although it presents several overlaps with other state-of-art architectures, such as Cloudlet [6],



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.2020.3045632, IEEE Access



Follow Me Cloud (FMC) [8], and Fog Computing (FC) [7]. The latter has numerous different definitions: in this paper we refer to the proposal in [27], which defines the FC as a "resource layer that fits between the edge devices and the cloud data centers, with features that may resemble either". A detailed illustration of service migration techniques and strategies can be found in [5]; here, the unique challenges of MEC compared to live migration for data centers and to handover management in cellular networks are highlighted. Similarly, in [28], the authors propose the concept of Companion Fog Computing (CFC), a software architecture composed of distributed layers, one running on the mobile device, and another on a fog server; the latter is dynamically allocated to nodes of a fog infrastructure in order to minimize the distance from the current device location. Similarly, the study in [29] proposes a cloudlet-based networking architecture including a cooperative algorithm for workload mobility among the nodes of the cluster. Generally speaking, MEC-related platforms must address two main issues: (*i*) how to define the service migration strategy, by taking into account the current resource utilization of the nodes as well as the QoS of the IoT application; (*ii*) how to implement the software mobility, by also handling the migration of the execution state. Regarding the first issue (migration policy), most of QoS-aware service migration policies focuses on delay as the principal performance indicator [30] and relies on multi-dimensional Markov Decision Process (MDP) models to describe the system evolution (i.e. the device mobility and consequential service mobility actions) over time (e.g. [11]). Since mobility patterns might be difficult to collect in advance, an increasing number of studies is investigating the application of Machine Learning (ML) techniques for the computation of the optimal migration policy; an example can be found in [3], where the usage of Deep Reinforcement Learning (DRL) technique is proved to maximize the users' reward, defined as the difference between the QoS and the migration cost. Among the non-delay oriented studies, we cite the self-organizing service management platform for smart-city proposed in [31], wherein the ETX (Expected Transmission Count) metric is used to determine the optimal allocation of IoT services to the fog nodes. Regarding the second issue (i.e. software mobility), Virtual Machines (VMs) and containers are the most investigated techniques to implement stateless or stateful service migration. Proactive migration of VMs according to predicted device mobility is considered in [9]; moreover, in order to reduce the network overhead induced by the VM transfer, a container synthesis technique is applied, allowing a fog node to quickly resume the VM execution by applying deltas over a base image. The possibility to perform horizontal (roaming) and vertical (offloading) migration of IoT functions based on Docker containers is demonstrated in [10]. From a performance perspective, the container-based implementation is often considered more suitable for the virtualization at the network edge than the VM-based [32]. This is confirmed by several experimental studies, including [33] that investigates the implementation of Docker-based virtualization mechanisms for IoT data management and demonstrates that the energy impact on single-board computers is negligible. An alternative to the usage of VM/containers is constituted by the migration of active code: to this purpose, the ThingMigrate framework [34] enables the migration of active `JavaScript` processes between different machines by employing injection mechanisms to track the local state of each function. With respect to the studies cited so far, the WT migration addressed in this paper can be considered a special instance of dynamic, agent-based [35] migration; at the same time, it presents novel opportunities as well as novel technical challenges which are discussed in detail in the Section below.

## III. M-WOT: PRELIMINARY DEFINITIONS AND MOTIVATIONS

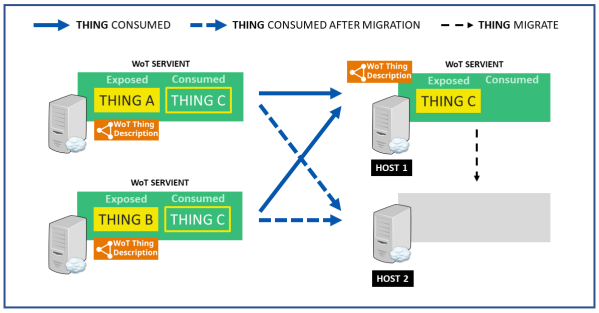



**FIGURE 2.** The M-WoT migration environment.

We consider a distributed scenario composed of a set of computing nodes distributed in the full stack of the IoT spectrum (from edge to cloud), as depicted in Figure 2; each node is W3C WoT enabled, i.e. it can host one or more Servients (i.e. the run-time environment of the W3C WoT architecture), and each Servient contains one single WT in running state. We define the WT migration as the *capability of dynamically offloading a WT between different nodes, by stopping the execution on the source node and respawning it on the destination node*. The migration process is assumed *stateful*, i.e. the internal state of a WT and its TD should be moved and updated together with the code. In particular, all the current values of its *Properties* and the information describing the current computational context of the WT should be considered as part of its *state* and hence migrated. Compared to classical migration approaches (VM/container/agent based) previously reviewed, the WT migration presents unique advantages as well as novel research challenges to be addressed:

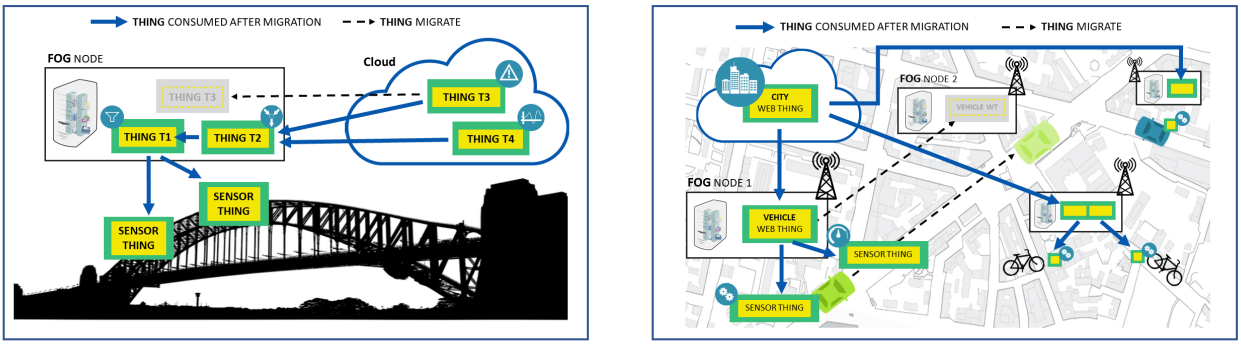
- *Challenge: Thing handoff management*. The W3C WoT allows seamless interactions among heterogeneous software through the operations of WT consuming; if a WT migrates to a different node, all the other WTs that were consuming it must be notified in order to update their Consumed objects and point to the new TD address. The



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.2020.3045632, IEEE Access






(a) (b)

**FIGURE 3.** Two possible M-WoT use-cases: the data processing service migration (Figure 3(a)) and the Digital Twin migration (Figure 3(b)).

case is depicted in Figure 2, where both WTs A and B are consuming WT C; the latter is migrated from Host 1 to Host 2 at some future instant. As a result, a proper signaling procedure must be employed in order to inform WTs A and B of when the activation of WT C at Host 2 has been completed, so that they could consume again the TD of WT C. Also, the migration process introduces an handoff interval, during which WT C might not be able to process remote invocations from WTs A and B; the duration of such handoff is clearly a critical parameter affecting the system performance.

- *Advantage: Support to Edge-cloud continuum.* Although our implementation is based on the reference W3C WoT run-time [21], Servients might have been designed with minimal requirements in terms of CPU/memory in order to be executed also on edge servers or even on the extreme edge (i.e. small devices, micro-controllers etc.). As a result, an edge-cloud computing continuum can be devised by allowing WoT software to be seamlessly deployed and dynamically moved over all the nodes of the continuum, in order to guarantee system goals such as delay minimization, workload balancing, network traffic reduction, privacy maximization.
- *Advantage: Support to Group migrations.* As followup of the previous point, a WoT Migration framework could support the mobility of groups of software components (rather than of a single service like it occurs in MEC approaches [5]) as a consequence of the active data dependencies (i.e. interactions) among the WTs, beside of the physical mobility of the IoT devices. Indeed, each WT exposes its Affordances through the TD in a standardized way; as a result, it is possible to build a real-time dependency graph among all the WTs of a distributed WoT system (as further detailed in Section V) and consequently envisage allocation policies that determine group migrations of interacting subsets of WTs in order to maximize the data locality. Clearly, group-based migration policies could be deployed also on top of other micro-services architectures; however, for the case of M-WoT, this feature could be supported in a general, protocol-agnostic way since the interactions among the WTs occur according to a standardized interface, and hence they could be easily accounted through the M-WoT monitoring layer described in Section IV-C.

Figures 3(a) and 3(b) show two possible use-cases of the WoT migration, related to slightly different conceptual models of WTs: the data processing service migration (Figure 3(a)) and the Digital Twin migration (Figure 3(b)). More specifically, Figure 3(a) depicts a Structural Health Monitoring (SHM) application based on IoT/WoT technologies [36] [37], as proposed among others by the MAC4PRO project [38]. We assume that the monitoring system can work in two modes: *Normal* and *Critical*, denoting two different QoS requirements for the risk detection. On the extreme edge there are the sensors (e.g. accelerometers) monitoring the vibrations of the building over time. The sensor data is made available through the Sensor Web Things (SWT) providing functionalities of data querying, and device status querying and updating. The sensor data processing is handled by migratable WTs $T_1$, $T_2$, $T_3$ and $T_4$ that implement respectively the functionalities of data fusion, data cleaning, data alerting, data forecasting. In Normal mode, $T_1$, $T_2$ are executed on a shelter/fog node in proximity of the monitored structure, while $T_3$ and $T_4$ are hosted on the remote cloud; this introduces some network latency in detecting anomalous/dangerous situations (computed by $T_3$) but at the same time it minimizes the load on fog nodes. At one point of the system execution, we assume that consecutive data anomalies are detected on the row data ($T_2$), and hence the monitoring system switches its mode from *Normal* to *Critic*; this action might also request an higher degree of responsiveness for the diagnostic system. In the M-WoT environment, the mode change can be automatically handled by migrating the $T_3$ service from the cloud to fog nodes (or vice-versa when the mode switches again to *Normal*), without any manual need of configuration, and without introducing any explicit signaling mechanism among the involved WTs (i.e. $T_2$ and $T_3$).

Figure 3(b) depicts the second M-WoT use-case where the



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.2020.3045632, IEEE Access



migration involves WoT digital twins. The latter is defined in the W3C standard as a *virtual representation of a device or a group of devices that resides on a cloud or edge node (...) they can model a single device, or they can aggregate multiple devices in a virtual representation of the combined devices*" [15]. To this purpose, we consider a WoT application for the automotive industry like the one proposed in [17]; a WT is associated to each in-vehicle component in order to enable seamless access and interaction to car signals. Similarly to [17], the Sensor Web Things (SWTs) are in charge of acquiring the data from the hardware of the vehicle. In addition, we assume the presence of a Vehicle Web Thing (VWT), defined as the digital twin of the vehicle as a whole; the VWT is the unique point of access to a subset of the SWTs properties/actions/events, but it also exposes new Affordances derived from the processing and fusion of multiple sensor data, e.g. for real-time vehicle diagnostic. Due to the energy overhead, the VWT is hosted externally to the vehicle, on fog nodes owned by the municipality. While the vehicle moves within the scenario, its VWT is dynamically spawned on the closest fog node, similarly to the MEC applications [28] [30], although here the physical mobility of a device induces the mobility of a WT digital twin. In addition, we conceive a city-wide scenario with many and heterogeneous VMTs, associated to different vehicle types (e.g. cars, bikes, buses, etc); the VMTs are in turn consumed by cloud-based City Web Things (CWTs) that provide advanced mobility-related services, such as smart parking, traffic monitoring, multimodal routing, etc. We highlight that the number of VMTs can be highly dynamic over time, i.e. new Things might be created or disposed, as an effect of the ground mobility; similarly the computational load needed for the execution of the VMTs and CWTs might vary over time. In our M-WoT environment, the VMTs are dynamically allocated among the cloud/fog nodes as they appear in the system; moreover, multi-goal load-balancing policies can be used, i.e. to minimize the distance from the data source (i.e. the vehicle) while maximizing the utilization of the computational resources of the fog/cloud nodes.

## IV. M-WOT: ARCHITECTURE

The M-WoT software architecture is depicted in Figure 4. We assume a set of W3C WoT Servients, deployed on different nodes; each Servient hosts exactly one WT. Differently from a legacy W3 WoT deployment, which is assumed static, the M-WoT enables WT mobility between different nodes. To this aim, the M-WoT features two novel components, respectively the Orchestrator and the Thing Directory; these modules do not migrate and can be deployed either on the edge (if the computational requirements are met) or on cloud servers. In addition, a Monitoring Layer has been added to the Servient's stack. In the following, we detail the internal structure of the three modules, while in Section IV-D we clarify the modules' operations when a WT migration process occurs.

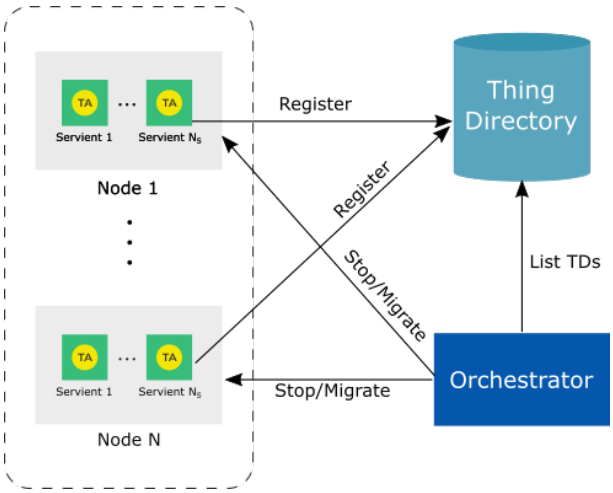



**FIGURE 4.** The M-WoT software block diagram and main interactions.

### A. THING DIRECTORY

The Thing Directory (TDir) serves as registry of the M-WoT resources, i.e. of the active Thing Descriptors (TDs). More in details, we assume two types of TDs, one associated to WTs, and one to Servients; the latter describes the capabilities of the run-time environment, and it is used to enable the functionalities of the Monitoring Layer described in Section IV-C. Once activated, each Servient registers its TD and the TD of the hosted WT on the TDir. The TDir itself plays two main roles. First, it serves as discovery service, i.e. when queried by clients, it returns the list of TDs meeting the query parameters; as a result, the Orchestrator module can be aware of the list of Servients currently available in the WoT scenario. Second, it supports generic push notifications towards WTs/Servients once specific system events are detected, for instance a WT handoff completion. To this purpose, let us assume that WT $T_1$ has been consumed by $T_2$, which is periodically accessing one of its properties. In case $T_1$ is migrated on a different node, the actual data-pipeline is broken unless $T_2$ is notified about the mobility event and the new service location. The notification process is illustrated in the sequence diagram of Fig. 7, discussed later in Section IV-D. Alternatively, a polling mechanism might be employed (involving $T_1$ and TDir in our example). However, this approach might introduce significant network overhead with consequent bandwidth wastage. Therefore it has not been considered in our solution.

### B. WT ORCHESTRATOR

The Orchestrator constitutes the core component of the M-WoT architecture. As explained before, it exploits the TDir to retrieve the list of active Servients (i.e. of their TDs). Then, it periodically queries each Servient through its WoT interface in order to collect live statistics, like the utilization of the CPUs and the network traffic generated by the WT interactions. Based on the received metric values and on the optimization policy in use, the Orchestrator determines the



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.2020.3045632, IEEE Access



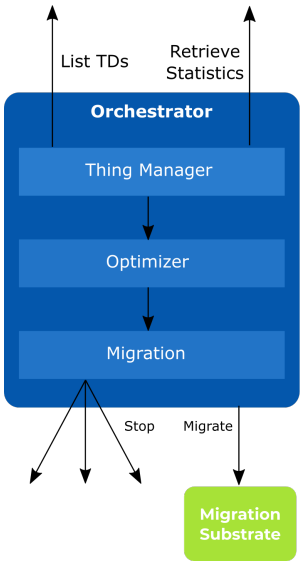



**FIGURE 5.** The internal structure of the Orchestrator.

proper allocation of WTs/Servients to nodes. The allocation plan is then transferred to an underlying layer (external to M-WoT), generically called here *Migration Substrate* which is in charge of implementing the physical software mobility between the source and destination nodes. The steps above are continuously executed by the Orchestrator during the system lifetime; as a result, the dynamicity of the IoT/WoT environment concerning the WT creation/disposal, network bandwidth variation, policy update at run-time, is fully supported. Moreover, in order to favour the platform extensibility, the structure of the Orchestrator has been split into the three main submodules of Figure 5, reflecting the internal data pipeline:

1) *Thing Manager*: it periodically polls data from the TDir to manage the list of the active Servients/WTs and their TDs. The list is used to gather periodic reports from each Servient.
2) *Optimizer*: it runs the WT/Servient allocation policy. At the current stage of implementation, the module hosts the graph-based optimization algorithm defined in Section V and the other greedy policies evaluated in Section VII; however, we remark that any user-defined policy implementing the interface towards the upper (i.e. the Thing Manager) and lower (i.e. the Migration) layers can be installed and used.
3) *Migration*: it receives the deployment plan from the Optimizer, and it implements the WT handoff events. First, it stops the execution of the WTs to migrate at their actual nodes; then, through specific connectors, it issues actions towards the Migration Substrate to enable the physical transfer of the Servients (and of the hosted WTs) from the source to the destination nodes.

The M-WoT architecture does not depend on any specific software mobility technology. Instead, we have introduced an abstraction layer - called the Migration Substrate- which can employ any state-of-art solution (via proper migration connectors), such as Docker containers, VMs, or Javascript processes [5] [34]. Those connectors will actuate the Optimizer output plan received as input. Concretely, the current implementation relies on Docker Swarm as a default migration connector, as better detailed in Section VI.

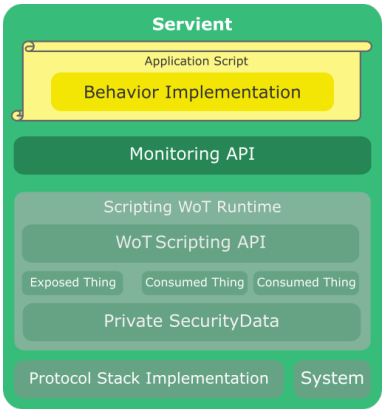



**FIGURE 6.** The M-WoT Servient internal structure. The new Monitoring API module is highlighted in solid green.

### C. M-WOT SERVIENT

Finally, the M-WoT framework introduces light modifications to the Servient runtime [21] in order to feed the Optimizer with real-time data about system performance. More specifically, a Monitoring API layer has been introduced between the WT application and the Scripting WoT runtime as depicted in Figure 6. The layer is in charge of intercepting the invocations to the Scripting API and of generating periodic Thing Reports (`TRs`). The latter can be considered a snapshot of the current Servient/WT execution, and it contains the metrics' values (both for the Servient and WT) required by the Optimizer; in the Appendix (Section IX) we report a fragment of the `TR` structure in use. The Monitoring layer exposes all the data collected through a proper Affordance action, which has been added to the Servient TD; by invoking it, the Orchestrator can issue a new request of `TR` generation to the Servient.

### D. MIGRATION EXAMPLE

To summarize the operations of the three components presented so far, we provide an example of WT migration process. We consider two WTs/Servients, respectively $T_A/S_A$ and $T_B/S_B$ (with $T_A$ running on $S_A$ and $T_B$ on $S_B$), hosted on nodes $N_1$ and $N_2$. We also assume that $T_B$ has consumed $T_A$ and it is periodically reading some of its properties. At time instant $t$, the Thing Manager queries $S_A$ and $S_B$ in order to collect the `TRs`; this is implemented by consuming the TDs of the Servients and issuing a `retrieveReport` command (details in Section VI). Then, the Optimizer is executed; a new allocation is produced where $T_A$ must be



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.2020.3045632, IEEE Access



moved to $N_2$. The sequence of operations performing the migration of $T_A$ from $N_1$ to $N_2$ are shown in Figure 7. First, the current execution of $T_A$ is stopped: this is performed by the Orchestrator (and more specifically by the Migration submodule) by invoking the `stop` action on $S_A$ which, in sequence, stops the WT application, cleans the system resources, retrieves the application data context (i.e. the current state) and returns it. Hence, the application context of $T_A$ is stored as metadata inside the TDir for later use. Next, the Orchestrator (through a proper Connector) issues a request to the Migration Substrate (e.g. Docker Swarm) in order to move $T_A$/$S_A$ to the destination node ($N_2$). After $S_A$ has been respawned, it register its new TD (with the updated network addresses of its Affordances) in the TDir. Consequently, it queries the TDir to retrieve the $T_A$'s context; the latter is deserialized and injected as a global object inside the $T_A$'s application script. Finally, $T_A$ starts the initialization process and exposes itself by triggering the registration of its TD on the TDir. At this point, $T_A$ resumes in the same state of when it has been stopped and it is considered fully migrated. The TDir pushes a notification to $T_B$ regarding the handoff process; $T_B$ retrieves the new TD of $T_A$ from the TDir and consumes it again in order to point to the updated service location. Finally, $T_B$ restarts interacting with $T_A$ and accessing its affordances.

## V. M-WOT: MIGRATION POLICY

In the following, we formally characterize the operations of the Optimizer as a multi-objective optimization problem. For the purpose of this study, we consider a twofold optimization process which takes into account the load-balancing issue (i.e. how much each host[1] is loaded), and the network communication overhead (i.e. how much data is exchanged among hosts). The optimization problem is formally defined in Section V-A. Next, a graph-based heuristic is proposed in Section V-B; its computational complexity is calculated in Section V-C. Table 1 reports the list of variables introduced in Section V-A.

### A. PROBLEM FORMULATION

Regardless of the target use-case, we consider a generic WoT deployment with $N_{WT}$ active WTs. The system evolves over ordered time-slots $T = \{t_0, t_1, \dots\}$; each slot has a duration of $t_{\text{slot}}$ seconds and is equal to the interval between consecutive executions of the migration policy. Let $WT = \{wt_1, wt_2, \ ..., wt_{N_{WT}}\}$ be the set of WTs, which can be heterogeneous in terms of data model (e.g. the Affordances). Without loss of generality, let $A_i = \{a_i^1, a_i^2, ..., a_i^{N_{A_i}}\}$ be the Affordances exposed by $wt_i$ in its TD; each Affordance can represent a property, an action or an event. The set $A_i$ is assumed static, i.e. $wt_i$ cannot update its TD at run-time (e.g. by defining new properties). Let $H$ be the set of hosting nodes, with $H = \{h_1, h_2, ..., h_{N_H}\}$ and assumed heterogeneous in terms of hardware. Indeed, each node may have a different computational power; without loss of generality, this is modeled through a generic computational power index $\gamma(h_l), \forall h_l \in H$ which abstracts from the hardware details, and is defined as the maximum number of Things that can be executed on that host. The allocation of WTs/Servients[2] to hosts is defined by the policy function $P : WT \times T \rightarrow H$; for each WT $wt_i$, the value $P(wt_i, t_k) = h_m$ specifies the machine (i.e. $h_m$) which is hosting it at time slot $t_k$. Based on the output of the allocation policy, the set $PT_{m,k} \subseteq WT$ denotes the list of WTs that are hosted by host $h_m$ at time-slot $t_k$, i.e.: $PT_{m,k} = \{wt_i \in WT |\ P(wt_i, t_k) = h_m\}$. According to the W3C WoT architecture surveyed in Section II, each WT $wt_i$ can interact with another WT $wt_j$ by first consuming it. This is modeled by assuming that, at each time-slot $t_k$, $wt_i$ can issue a list of requests $R_{i,j,k}$=$\{r^1_{i,j,k}, r^2_{i,j,k}, ...\}$ on the consumed $wt_j$; each request $r^y_{i,j,k}$ refers to an Affordance of $wt_j$, and it consists in: a property reading/writing, action invoking or event processing. It is worth highlighting that the notation above assumes that the same Affordance $a^x_i$ might be activated multiple times by $wt_i$ during the same time-slot, although they are considered different requests (e.g. WT $wt_i$ reads twice the same property $a^x_j$ on WT $wt_j$ during time-slot $t_k$). The implementation of each request $r^y_{i,j,k}$ involves some data exchange between WTs $wt_i$ and $wt_j$; let $B(r^y_{i,j,k})$ be the data exchanged (in bytes) between the two WTs, including both the eventual parameters passed from $wt_i$ to $wt_j$ as well as the eventual return values from $wt_j$ to $wt_i$. The $B(r^y_{i,j,k})$ value is included in the `TR` message, which is periodically sent by each WT to the Optimizer as previously described in Section IV. We denote with $B(i,j,k) = \sum B(r^y_{i,j,k})\ \forall r^y_{i,j,k} \in R_{i,j,k}$ the total communication load occurring between WTs $wt_i$ and $wt_j$ at time slot $t_k$. Clearly, $B(i,j,k) = 0$ whether $wt_i$ is not consuming $wt_j$ at time $t_k$, or no interaction occurs among them (i.e. $R_{i,j,k} = \emptyset$).

The goal of the Optimizer is to determine the policy which computes - at each time-slot $t_k$- the optimal trade-off between computational resource utilization (i.e. load balancing over the hosts) and data locality (i.e. how much data is transferred among the hosts). To this purpose, we define the Network Overhead ($NO$) metric as the total inter-host communication load (in bytes) occurring due to interactions among WTs hosted by different nodes. More formally:

$$NO(t_k) = \sum_{wt_i \in WT, wt_j \in WT, P(wt_i,t_k) \neq P(wt_j,t_k)} B(i,j,k) \tag{1}$$

We clarify that the $NO(t_k)$ metric above quantifies the *end-to-end*, application-layer, communication traffic between nodes of the M-WoT cluster, generated by the interactions among different WTs; it does not include the network-layer overhead (e.g. caused by multi-hop message forwarding among the routers) since the M-WoT framework is imple-

[1]The terms hosts and nodes are used interchangeably in this paper.

[2]For ease of disposition, we refer to WT migration in the following by meaning also the migration of the Servient hosting it. We do not model the Servient in the theoretical framework, since we assume that each Servient hosts exactly one WT.



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.2020.3045632, IEEE Access



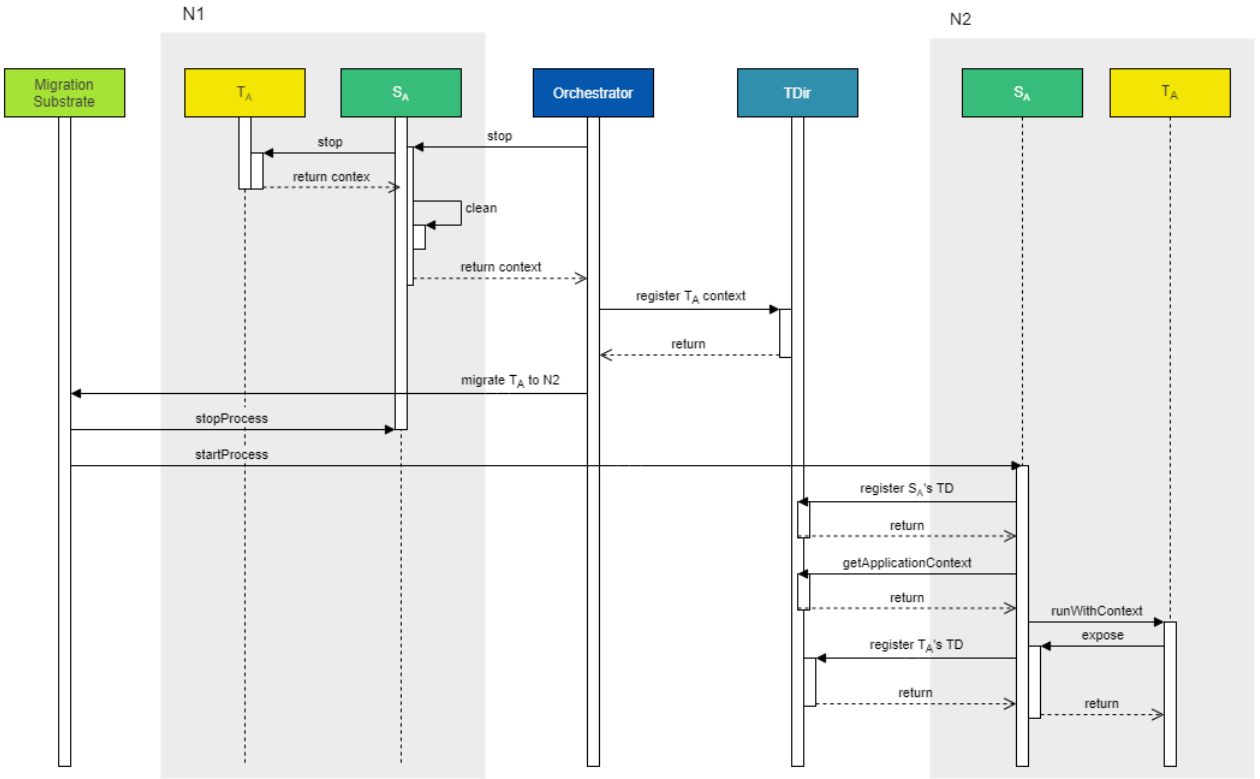



**FIGURE 7.** Sequence diagram of a WT migration event.

| Parameter | Description |
|---|---|
| $t_{\text{slot}}$ | Slot Length (interval between consecutive migration policy executions) |
| $T = \{t_0, t_1, ...t_k\}$ | Temporal evolution as sequence of fixed-length time slots |
| $H$ | Set of hosting nodes composing the M-WoT cluster |
| $N_H = \|H\|$ | Number of hosting nodes |
| $WT$ | Set of Web Things (WTs) |
| $N_W = \|WT\|$ | Number of Web Things (WTs) |
| $\gamma(h_l)$ | Computational power of hosting node $h_l$ |
| $P(wt_i, t_k)$ | Allocation function, returning the hosting node of WT $wt_i$ at time $t_k$ |
| $p^{t_k}_{wt_i,h_m}$ | Binary variable defining the allocation of the WTs over time depending on $P(wt_i, t_k)$ |
| $PT_{m,k}$ | Set of WTs hosted by node $h_m$ at time $wt_i$ |
| $A_i = \{a_i^1, a_i^2, ..., a_i^{N_{A_i}}\}$ | List of Affordances exposed by WT $wt_i$ |
| $r^y_{i,j,k} \in R_{i,j,k}$ | Affordance request issued by $wt_i$ toward $wt_j$ at time $t_k$ |
| $R_{i,j,k}$ | Cumulative list of Affordance requests issued by $wt_i$ towards $wt_j$ at time $t_k$ |
| $B(r^y_{i,j,k})$ | Traffic load (in bytes) between WTs $wt_i$ and $wt_j$ to execute request $r^y_{i,j,k}$ |
| $B(i,j,k)$ | Total traffic load (in bytes) exchanged between $wt_i$ and $wt_j$ at time $t_k$ |
| $NO(t_k)$ | Total inter-host communication at time $t_k$ in the M-WoT cluster |
| $L(h_m, t_k)$ | Load Ratio of host $h_m$ at time $t_k$: number of WTs hosted, normalized over $\gamma(h_m)$ |
| $HF(t_k)$ | Difference between most loaded and unloaded hosting nodes (loads expressed in terms of $L(h_m, t_k)$) |
| $\Delta$ | User-defined constraint on the $HF(t_k)$ value |

**TABLE 1.** List of variables and parameters introduced in Section V-A.

mented at application layer and the knowledge of the topology of the underlying network infrastructure is not assumed. Similarly, we introduce the Host Fairness ($HF$) metric defined as the difference between the most loaded and most unloaded host of the cluster, i.e.:

$$HF(t_k) = max_{h_m \in H} L(h_m, t_k) - min_{h_m \in H} L(h_m, t_k) \quad (2)$$

here, $L(h_m, t_k)$ defines the computational load ratio of $h_m$ at time-slot $t_k$, and it is related to the number of WTs hosted by it over its computational power, i.e.:

$$L(h_m, t_k) = \frac{|PT_{m,k}|}{\gamma(h_m)} \quad (3)$$



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.2020.3045632, IEEE Access



Let $p^{t_k}_{wt_i,h_m}$ be the binary variable indicating the WT allocation, defined $\forall t_k \in T$, $\forall wt_i \in WT$, and $\forall h_m \in H$ as follows:

$$p^{t_k}_{wt_i,h_m} = \begin{cases} 1 & \text{if } P(wt_i, t_k) = h_m \\ 0 & \text{otherwise} \end{cases} \quad (4)$$

Through the NO and HF metric introduced above, the migration problem can be formally defined as follows:

$$\min_{p^{t_k}_{wt_i,h_m}} \quad NO(t_k) \quad (5)$$

$$\text{s.t.} \quad L(h_m, t_k) \leq 1 \quad \forall h_m \in H \quad (6)$$

$$HF(t_k) \leq \Delta \quad (7)$$

Constraint 6 ensures that the allocation on each host does not exceed the computational capabilities of that host ($\gamma(h_m)$). In Constraint 7, $\Delta$ is a user-defined parameter, which quantifies the trade-off previously mentioned. It is easy to notice that the $HF$ and $NO$ metrics are tightly coupled: minimizing the network load can be achieved by a policy which allocates all the WTs to the same host. However, this constitutes the worst case for the load fairness. Hence, two extreme scenarios are possible:

- The system goal is to minimize the data exchanged over the network, regardless of the service latency; this might the case of a edge-cloud IoT scenario, where the stakeholder is interested in minimizing the amount of data transferred toward a remote infrastructure for privacy reasons. In this case, $\Delta = \infty$.
- The system goal is to minimize the service latency, by avoiding the presence of performance bottlenecks, i.e. overloaded hosts, while still mitigating the amount of inter-host communications. In this case, $\Delta = 1$.

All the intermediate situations are modeled through a proper tuning of the $\Delta$ parameter, which is assumed as input of the optimization problem.

### B. PROPOSED HEURISTIC

We propose a graph-based heuristic that ensures the constraint 7, while relaxing the constraint 6 and addressing the goal function (Equation 5) through a greedy approach. The solution relies on the construction of a WT dependency graph $G(V, E, W, L)$ which models the interactions among the WTs:

- $V$ is the set of vertexes; each vertex represents a WT, hence $V = WT$ and $v_i = wt_i \; \forall wt_i \in WT$.
- $E$ is the set of edges; each edge $e_l(v_i, v_j)$ connects two vertexes $v_i, v_j \in V$ and models the interaction between the two WTs. More specifically, there exists the edge $e_l(v_i, v_j)$ only if $B(i, j, k) > 0$ or $B(j, i, k) > 0$.
- $W : E \rightarrow \mathbb{R}$ is a weight function, assigning a cost to each edge $e_l(v_i, v_j) \in E$. Here, the value $W(e_l(v_i, v_j))$ quantifies the total data exchanged among WTs, in case $wt_i$ is consuming $wt_j$ or vice versa, i.e. $W(e_l(v_i, v_j)) = B(i, j, k) + B(j, i, k)$.
- $L : V \rightarrow \mathbb{R}$ is a load function, assigning a cost to each vertex $v \in V$. If we assume to know the CPU load ($C(r)$) induced by each request received by $wt_j$, then $L(v_j)$ can be defined in a fine-grained way as $L(v_j) = \sum_{wt_i \in WT} \sum_{r \in R_{i,j,k}} C(r)$. In this paper, we do not assume such knowledge, hence we generally set $L(v_i) = 1 \; \forall v_i \in V$, i.e. all WTs are assumed to produce the same load, while the total load of host $h_m$ (denoted as $L(h_m)$ in the following) is simply the number of WTs hosted.

The graph $G$ is built and continuously updated by the Optimizer by processing the `TR` messages received by each Servient. At the beginning of each time-slot (e.g. $t_k$), the Optimizer visits the graph and allocates the WTs to hosts according to the policy output (i.e. the $PT(h_i, t_k)$ values); since the policy is computed once for each slot, we omit the temporal notation (i.e. the $t_k$) in the rest of this Section.

The rationale of the proposed policy is the following. First, we compute the set of connected components on the dependency graph $G$. By construction, each component contains a closed set of interacting WTs; hence, the network overhead occurring among different graph components is equal to zero. The load of each component is defined as the sum of loads of its WTs; next, graph components are ordered based on their load values, and assigned to hosts in a round-robin way. In case the constraint 7 (load fairness) is satisfied, the algorithm stops its execution. Otherwise, we break the connected components computed so far (hence, introducing some network overhead at each iteration) by iteratively migrating one WT from the most loaded host to the most unused one, till the constraint 7 is satisfied. The migrated WT $wt_i$ is selected in a greedy way as the one which minimizes the network overhead, computed as the difference between: (*i*) the new overhead generated when detaching $wt_i$ from the source host and (*ii*) the performance gain on the destination host, caused by the fact that $wt_i$ has become a local service on that host.

Algorithm 1 shows the pseudo-code of the proposed heuristic. First, we build the dependency graph $G(V, E, W, L)$ and we compute its set of connected components, denoted as $GC$ at line 1. The load of each component $G_i$ (i.e. $L(G_i)$) is estimated as the sum of the loads of its vertices (line 3). Then, we order the set $GC$ based on the load values, and the set of hosts $H$ based on the computational power of it, represented by the $\gamma$ metric. To this purpose, at lines 5-6, the function `Sort` (not reported here) is sorting a set passed as first argument in descending order according to the metric values given by the second argument. The loop at lines 8-13 assigns sub-graphs to hosts in a round robin, by also updating the load for each host as the load of its vertexes/WTs (line 11). Next, we check whether constraint 7 is satisfied through the `CheckBalanced` function (lines 30-39), which also returns the hosts associated to the highest and lowest load values, respectively $h_{max}$ and $h_{min}$. If the load difference is lower than the user-threshold $\Delta$, than the current allocation is returned. Vice-versa, a greedy mechanism is implemented through the loop at lines 15-27; here, at each iteration, a



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.2020.3045632, IEEE Access



**Algorithm 1:** The graph-based heuristic

**Input:** Dependency graph $G(V, E, W, L)$, Time-slot $t_k$
**Output:** Allocation sets $PT(h_m, t_k) \forall h_m \in H$

1 $GC = \{G_1, G_2, ..., G_{N_C}\} \leftarrow$ `GetGomponent`$(G)$
2 **forall** $G_i \in GC$ **do**
3 $\quad L(G_i) \leftarrow \sum_{v \in G_i} L(v_i)$
4 **end**
5 $GC \leftarrow$ `Sort`$(GC, L)$
6 $H \leftarrow$ `Sort`$(H, \gamma)$
7 $cont \leftarrow 0$
8 **while** $GC \neq \emptyset$ **do**
9 $\quad G_h \leftarrow$ `Head`$(GC)$
10 $\quad PT(cont, t_k) \leftarrow PT(cont, t_k) \bigcup G_h$
11 $\quad L(h_{cont}) \leftarrow L(h_{cont}) + L(G_h)$
12 $\quad cont \leftarrow (cont + 1)\%N_H$
13 **end**
14 $\langle balanced, h_{min}, h_{max} \rangle \leftarrow$ `CheckBalanced`$(H, \Delta)$
15 **while** $balanced ==$ `false` **do**
16 $\quad$ **forall** $v_i \in PT(h_{max}, t_k)$ **do**
17 $\quad\quad loss \leftarrow$ `TotInteractions`$(v_i, PT(h_{max}, t_k))$
18 $\quad\quad gain \leftarrow$ `TotInteractions`$(v_i, PT(h_{min}, t_k))$
19 $\quad\quad overhead(v_i) = loss - gain$
20 $\quad$ **end**
21 $\quad v_s \leftarrow$ `argmin`$(overhead(v_i))\ \forall v_i \in PT(h_{max}, t_k)$
22 $\quad PT(h_{min}, t_k)) \leftarrow PT(h_{min}, t_k) \bigcup \{v_s\}$
23 $\quad L(h_{min}) \leftarrow L(h_{min}) + L(v_s)$
24 $\quad PT(h_{max}, t_k)) \leftarrow PT(h_{max}, t_k) \setminus \{v_s\}$
25 $\quad L(h_{max}) \leftarrow L(h_{max}) - L(v_s)$
26 $\quad \langle balanced, h_{min}, h_{max} \rangle \leftarrow$ `CheckBalanced`$(H, \Delta)$
27 **end**
28 **return** $PT$
29
30 **Function** `CheckBalanced`$(H, \Delta)$**:**
31 $\quad h_{min} \leftarrow$ `argmin`$(L(h_i))\ \forall h_i \in H$
32 $\quad h_{max} \leftarrow$ `argmax`$(L(h_i))\ \forall h_i \in H$
33 $\quad n_{iter} \leftarrow n_{iter} + 1$
34 $\quad$ **if** $L(h_{max}) - L(h_{min}) \leq \Delta$ **then**
35 $\quad\quad balanced \leftarrow$ `true`
36 $\quad$ **else**
37 $\quad\quad balanced \leftarrow$ `false`
38 $\quad$ **end**
39 $\quad$ **return** $balanced, h_{min}, h_{max}$
40
41 **Function** `TotInteractions`$(v_s, S)$**:**
42 $\quad interactions \leftarrow 0$
43 $\quad$ **forall** $v_j \in S$ **do**
44 $\quad\quad interactions \leftarrow interactions + W(e(v_i, v_j))$
45 $\quad$ **end**
46 $\quad$ **return** interactions

candidate WT $v_s$ is migrated from $h_{max}$ to $h_{min}$ (lines 22-25) (by consequently updating the per-host load information) and the load balance condition is evaluated again at line 26. The WT/vertex to migrate ($v_s$) is selected as the one that minimizes the overhead function at line 21. The latter takes into account: (*i*) the total amount of network communications (in bytes) between $v_s$ and any other WT hosted by $h_{max}$, which will now become inter-host communications and hence will constitute a network overhead after the WT migration (the value is stored within the $loss$ variable at line 17); (*ii*) the total amount of network communications (in bytes) between $v_s$ and any other WT hosted by $h_{min}$, which will now occur locally (intra-host communication) and hence will reduce the network overhead (the value is stored within the $loss$ variable at line 18). The computation of gain/loss values is performed through the helper function `TotInteractions` (lines 41-46) that returns the total number of interactions occurring between a target vertext/thing ($v_s$) and a set of vertexes ($S$) provided as inputs, over the dependency graph $G$.

### C. COMPUTATIONAL COMPLEXITY

The computational complexity is expressed in terms of $N_W$ (number of WTs) and $N_H$ (number of nodes) for the worst case scenario. At line 1 of Algorithm 1, the connected components of graph $G$ are computed; this operation is completed in time O($N_W$) through a DFS graph visit. Then, from line 8 to line 13, the connected components are assigned to the computational nodes; again this is performed in O($N_W$). The complexity of the balancing loop (from line 15 to line 26) depends on the $\Delta$ value and on the $L$ function definition. We assume that all hosts are homogeneous ($\gamma(h_m)$=1 $\forall h_m \in H$), hence $L(h_m, t_k) = PT_{m,k}$. The assumption is compliant with the experimental analysis presented in Section VII. By considering a totally unbalanced allocation of WTs to nodes, the loop is executed for $N_W - \Delta$ times; the internal loop (lines 16-20) has complexity of $O(\frac{N_W}{N_H})^2$ since we visit each WT hosted by the most used node, and for each WT we compute the total $NO$ with the WTs hosted on the most unused node. Finally, the `CheckBalance` function loops over the $N_H$ set hence it has complexity of $O(N_H)$. Putting all together, the complexity of Algorithm 1 is $O(N_W)$ in case the load balancing procedure is not executed (e.g. $\Delta = \infty$). Vice versa, it is dominated by the loop of lines 14-26, and it has complexity equal to $\frac{O(N_W^2)}{O(N_H^2)} + O(N_W) \cdot O(N_H)$. Since we expect that $N_W << N_H$, the overall complexity of Algorithm 1 is $\sim O(N_W^2)$.

## VI. M-WOT: IMPLEMENTATION

We detail here the implementation of the architecture components presented in Section IV. Our solution extends the Thingweb node-wot [21] framework, the official reference implementation of the W3C WoT Working Group, to which we added specific primitives in order to support the WT migration process.

### A. THING DIRECTORY & ORCHESTRATOR

The TDir is implemented as a dedicated (non-migratable) WT, which is hosted by a WoT Servient exposing a specific API for managing TDs and contexts. Among the most important interaction Affordances we cite: the `registerThing` action that takes a TD as input, and makes it globally available to the other M-WoT components; the `getThingById` and `listThings` actions, which respectively return one or more TDs based on the id or on a semantic filter; the



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.2020.3045632, IEEE Access



`getContextById` action, which returns the context associated to a WT, and the `thingRegistered` event, which is triggered each time a WT registers itself to the TDir, and causes its TD to be broadcasted to all the subscribers. The Orchestrator is implemented as a `Node.js` application written in `TypeScript` and using the Nest[3] (v6) framework in *standalone application* mode. The Orchestrator includes several modules working in synergy, and corresponding to the three components presented in Section IV-B:

- *Thing Manager*: it provides a `TasksManager` capable of executing generic tasks at a specific schedule; the functionality is implemented by the `@nestjs/schedule` package, which in turn uses the `node-cron`[4] package. Among the others, we cite the `collectReports` task that periodically retrieves the list of active WTs through the TDir and invokes the `retrieveReport` action on each one of them to get the corresponding `TR`s.
- *Optimizer*: it provides the data structures representing the current status of the M-WoT deployment. In particular, it records the live metrics of WTs (i.e. interactions with other WTs) and the list of the hosting nodes. Moreover, it provides the `Policy` abstract class, with a `getAllocation` method that returns the planned allocation of WTs to nodes (i.e. the $PT(h_m, t_k)$ sets of Algorithm 1). Any new policy installed in the Optimizer must implement the method above.

### B. WOT SERVIENT

The default *node-wot* tool [21] has been extended in two directions: (*i*) the script run time has been proxied with a monitoring module, and (*ii*) the default CLI implementation has been modified to handle WT state injection and retrieval.

#### 1) Monitor APIs

The Monitoring API is a collection of `Typescript` classes and functions collecting the data needed by the Optimizer. More specifically, the Monitoring API intercepts any invocation of the WTs to the underlying WoT scripting functions, and updates the number of activations of each property/action/event as well as the total time of completion. Then, it stores such data inside the `TR`, whose structure is reported in Section 1. The main fields of the `TR` include: the `id` of the WT being monitored, the `hostID` of the node hosting the WT/Servient, the `serviceID` used to map the WT to the corresponding docker swarm service, the average CPU and memory utilization of the node, and the Interaction List. The latter contains statistics related to the interaction with each consumed WT, and more specifically the number of times a specific Affordance has been activated, and the latency involved in the request-response.

[3]https://nestjs.com/

[4]https://github.com/kelektiv/node-cron

#### 2) Context migration

In case of active WT, the M-WoT framework supports the migration of its context, i.e. all the information characterizing the internal *state* and including: `Global variables` of the Thing Application, the `Properties values` and the current `State` of eventual external libraries in use. Before migration can start, all the possible running operations should be interrupted and the context must be collected. This has been implemented by adding the `stop` method to the TD of the WT, which disables all its Affordances in order to avoid a possible state change during the context saving. After that, it collects the WT `Context` and returns it to the Servient; the context is then stored on the TDir as described in Section IV. After the new Servient has been deployed, and before running the migrated WT, it makes a request to the TDir (by using the `Thing ID`) for retrieving the context. The latter is then passed to the WT to be loaded, hence restoring the state at the time of migration. For sake of simplicity, and to ease the programmers' tasks, we automatized the process of adding the auxiliary functionalities inside the WT behaviour. More in details, the methods for stopping the WT Affordances and returning the context are automatically injected into the code of the WT application by the Servient before exposing it. The Servient searches for a specific comment in the script (*/*INIT*/*) to understand whether and where the M-WoT code should be inserted. The only operation required to the programmer in order to make a WT migration-enabled is to add such comment to the application code.

## VII. EVALUATION

In this Section, we test the performance of the M-WoT framework via a twofold experimental evaluation. First, in Section VII-A we compare different migration policies, including multiple variants of the graph-based heuristic presented in Section V, on ad-hoc edge scenarios. Then, in Section VII-B we investigate the effectiveness of WT migration mechanisms on the edge-cloud continuum. More in detail, we evaluate a concrete IoT structural monitoring application inspired by one of the use cases presented in Section III (see Figure3(a)). The characteristics and parameters of each scenario are discussed separately in Sections VII-A and VII-B.

### A. POLICY ANALYSIS

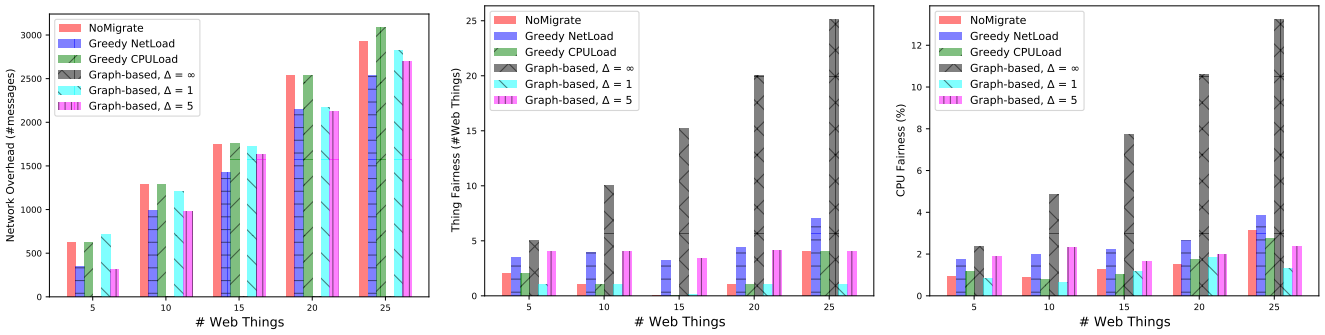
We consider a distributed setup composed of three edge servers (i.e., $N_H = 3$), physically located at the DISI/ARCES data centers of the University of Bologna, and connected through an Ethernet LAN, at one hop distance one from each other. Specifically, two servers are equipped with 4-core 2 GHz CPUs and 4 Gb of RAM, while the third server is equipped with an Intel Xeon E5440 processor with 32 Gb of RAM. Moreover, the Orchestrator and the TDir have been installed on a different node within the same data center. Therefore, in total, the experimental setup is composed of 4 nodes, three of which constitute the M-WoT deployment space, and can be used to host the WTs. On this space, we deployed $N_{WT}$ Servients, each hosting exactly one WT; at



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

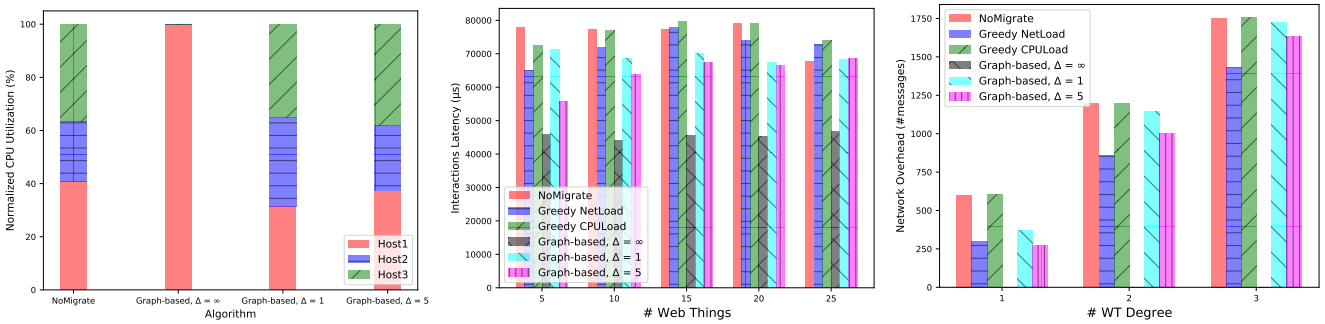
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3045632, IEEE Access






(a) (b) (c)

**FIGURE 8.** The $NO$, $TF$ and $CF$ metrics for the six policies when varying the number of active WTs are shown respectively in Figures 8(a), 8(b) and 8(c).




(a) (b) (c)

**FIGURE 9.** The average utilization of each computational node is shown in Figure 9(a). The $IL$ metric when when varying the number of active WTs is shown in Figure 9(b). The $NO$ metric as a function of the WT degree is reported in Figure 9(c).

the startup, the Servients are randomly allocated over the available $N_H$ nodes. The WT interactions are modeled as follows. We abstract from the physical meaning of the WT and the correspondence to specific real-world applications since the focus is on the assessment of the migration operations and on the evaluation of the policies' performance. Hence, each WT exposes exactly one action in its TD (e.g. `test`), which computes a sequence of trigonometric operations (mainly $tan$ and $atan$) in order to generate some CPU load. Each WT (e.g. $wt_i$) consumes exactly other $N_C$ WTs, chosen randomly among the $N_{WT}$ available. On each consumed thing $wt_j$, $wt_i$ issues a request for the `test` action every 1.5 seconds. In order to automatically apply the test configurations on each WTs, we implemented a *Mashup* application, i.e. a WoT client that is in charge of consuming the WTs and of passing them the proper setup (e.g. the list of WTs to consume). Every $t_f$=45 seconds, the Orchestrator collects the Thing Reports (`TR`) produced by each Servient; every 190 seconds, a new WT allocation is computed by the Optimizer according to the current policy, and implemented through proper WT migrations among the edge servers. The latter is also the duration of one time slot (i.e. $t_{slot}$=190 seconds), in accordance with the problem formulation presented in Section V-A. The setting of $t_f$ and $t_{slot}$ parameters allows the Optimizer to collect at least three reports from each WT and hence to estimate the WT interactions before computing a new allocation of WTs to nodes. The performance analysis is based on the following metrics:

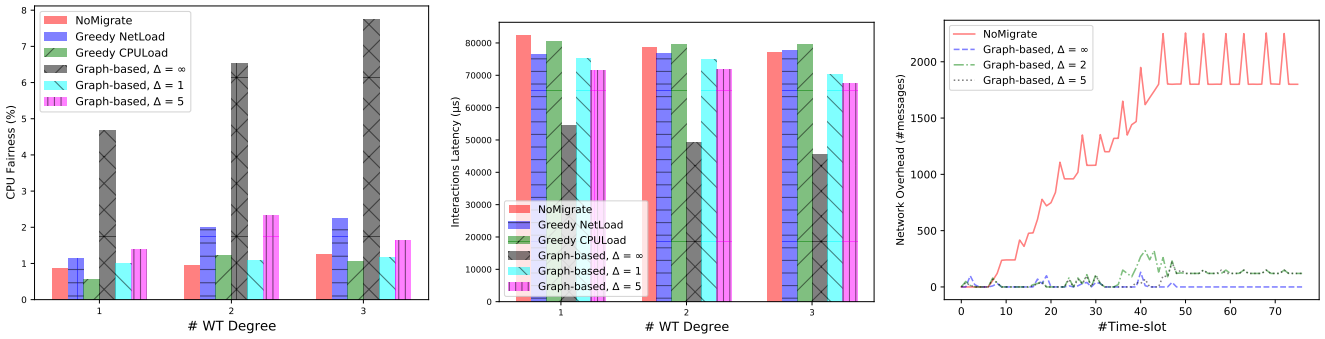
- *Network Overhead* ($NO$): this is the performance index defined by Equation 1 and quantifying the amount of inter-host network communications produced by remote WT interactions. Differently from the theoretical model, we compute the $NO$ in terms of number of interactions rather than of bytes, since all the WT interactions refer to the same affordance (i.e. the `test` action); this is the equivalent to set $B(i,j,k)$=1 in Equation 1, $\forall wt_i, wt_j \in WT, t_k \in T$.
- *CPU Fairness* ($CF$): this is the performance index defined by Equation 2 and quantifying the fairness unbalance in terms of max-min difference of the average CPU occupation loads among the $N_H$ nodes of the cluster. We set $\gamma(h_l) = 1, \forall h_l \in H$.
- *Thing Fairness* ($TF$): this is similar to the $CF$ metric, however the fairness unbalance is expressed in terms of number of WTs hosted respectively by the most loaded and unloaded node (rather than of average CPU values).
- *Interaction Latency* ($IL$): this is the average latency required to perform a WT action invocation issued by an external WT; more explicitly, this is the average time lapsed from when $wt_i$ issues a `test` action on $wt_j$



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.2020.3045632, IEEE Access






(a) (b) (c)

**FIGURE 10.** The $CF$ and $IL$ metrics when varying the WT degree are shown respectively in Figures 10(a) and 10(b). The $NO$ over time-slots in a dynamic WoT deployment where the number of WTs is varied over time is reported in Figure 10(c).

to when the corresponding reply is received. Hence, it takes into account both the processing delay and the network delay in case $wt_i$ and $wt_j$ are executed on different nodes of the cluster.

We compared the following policies:

- *NoMigrate*: this is the state-of-art WoT solution, i.e. the WTs are statically deployed on nodes and they are not migrated during the whole system lifetime.
- *Greedy NetLoad*: this is a greedy policy which aims at minimizing the $NO$ metric. At each time slot, it selects the WT generating the highest NO, and moves it towards the same node of the consumer WT.
- *Greedy CPULoad*: this is a greedy policy which aims at minimizing the $CF$ metric. At each time slot, it selects the edge node of the cluster associated with the highest average CPU load, detaches one WT and moves it towards the node with the lowest CPU load.
- *Graph-based,* $\Delta = \infty$: this is the WT dependency-graph policy presented in Section V; we set $\Delta = \infty$, hence the policy aims exclusively at minimizing the $NO$ metric, while no load-balancing action is executed (i.e. lines 16-26 of Algorithm 1 are skipped).
- *Graph-based,* $\Delta$*=5*: this is again the policy of Section V, where the balance parameter is put into action. The policy computes a minimal $NO$ solution ensuring that $TF$ metric cannot exceed the $\Delta$ threshold equal to 5.
- *Graph-based,* $\Delta$*=1*: this is similar to the previous policy, however we set the maximum balancing of the WT allocations over the nodes of the cluster.

For each configuration, we ran 10 repetitions, and then averaged the metric values; on each repetition, a random initial allocation of WTs to nodes, and random dependencies among the WTs are considered.

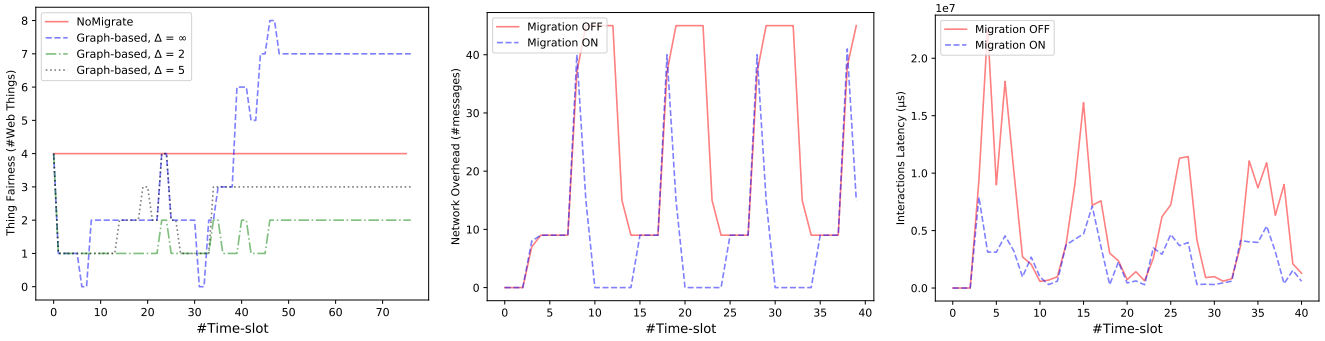
Figure 8(a), 8(b), 8(c) and 9(a) show the metrics previously introduced when varying the policy in use and the $N_{WT}$ configuration, i.e. the number of WTs in the scenario. The $N_C$ value is fixed and equal to 3, i.e. each WT consumes exactly 3 peers, randomly selected. From the $NO$ values of Figure 8(a), we can notice that the amount of inter-host communications increases with the number of active WTs, as expected. At the same time, the *Graph-based* and the *NetLoad* policies are more effective than the *NoMigrate* and the *CPULoad* since they both aim at allocating interacting WTs on the same node; the $NO$ performance gain of the *Graph-based* policy can be tuned through the $\Delta$ parameter. For $\Delta = \infty$, the $NO$ is always zero, since the WT dependency graph is likely connected (this is also due to $N_C$=3); as a result, all the WTs are moved to the same edge node, as better highlighted below. For $\Delta = 1$ and $\Delta = 5$, the *Graph-based* policy introduces some $NO$ due to the load-balancing constraint, but still lower than the *NoMigrate*, hence it is preferable to a random allocation. The load-balancing capabilities of the six policies are investigated in Figure 8(b) which shows the $TF$ metric as a function of the number of WTs; for the *Graph-based* with $\Delta = \infty$, the $TF$ is always equal to the number of WTs in the scenario, since all the WTs are allocated to the same node. Vice versa, we can notice that, for $\Delta = 1$ and $\Delta = 5$, the $TF$ value is always lower than the required threshold, demonstrating the effectiveness of the load-balancing mechanism. The fairness in terms of WTs translates into a better utilization of computational resources, as investigated in Figure 8(c). Here, the $CF$ metric is shown for the six policies; we can notice that the *Graph-based* heuristic with $\Delta = \infty$ and $\Delta = 1$ are respectively the worst and optimal cases, once again demonstrating the versatility of our approach. By comparing Figures 8(a) and 8(c), we can also appreciate that the *Graph-based* policies (with $\Delta \neq \infty$) are able to achieve a better trade-off between $NO$ and $CF$ metrics when compared to the two Greedy policies; based on the system requirements (i.e. data locality or resource utilization), the administrator can achieve the wanted performance trade-off by properly tuning the $\Delta$ parameter, whose optimal setting is clearly scenario-dependant. Figure 9(a) provides additional insights on the WT allocation, by showing, for the *Graph-based* policies and different values of $\Delta$, the average CPU utilization of each node of the cluster (denoted by the colors on each bar); the CPU values are normalized between 0 and 100%. It is easy to notice that lower values of $\Delta$ correspond to more balanced utilization of the computational resources



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.2020.3045632, IEEE Access






(a) (b) (c)

**FIGURE 11.** The $TF$ over time-slots in a dynamic WoT deployment where the number of WTs is varied over time is reported in Figure 11(a). The $NO$ over time in the IoT monitoring use-case is shown in Figure 11(b); the processing latency for the same scenario is reported in Figure 11(c).

of the cluster, while for $\Delta = \infty$ only one node is used. Finally, Figure 9(b) shows the $IL$ metric for the six policies; we highlight that the latency is not taken into account in the optimization framework of Section V, although delay-aware policy can be designed and installed in the Optimizer as future work. Nevertheless, the *Graph-based* with $\Delta = \infty$ overcomes the other competitors for all the configurations of WTs; this is due to the reduction of communication latency since all the WT interactions occur locally on the same node. In Figures 10(a), 10(b), 10(c) we expand the evaluation by considering the impact of different WT interaction amounts on the system performance. More specifically, we consider a fixed number of WTs ($N_{WT}$=15), while on the x-axis we vary the WT degree ($N_C$), i.e. the number of peers consumed by each WT, again selected in a random way. Figure 10(a) depicts the $NO$ metric for the six policies; as expected, the amount of inter-host communication increases with the $N_C$ values on the x-axis. The only exception is the *Graph-based* with $\Delta = \infty$: similarly to the previous analysis, the $NO$ is zero since interacting WTs are allocated to separate nodes, however more than one connected component is found on the dependency graph for $N_C$=1 and $N_C$=2. As a result, the $CF$ metric of the *Graph-based* with $\Delta = \infty$ shows the increasing trend of Figure 10(a); for $N_C$=1 and $N_C$=2, a more balanced allocation is achieved since the graph components are allocated to different nodes, while for $N_C$=3 the graph is fully connected hence the whole workload is allocated to the same node. Comparing 9(c) and 10(a), we can appreciate again how the *Graph-based* policies (with $\Delta \neq \infty$ ) are able to capture a better $NO$-$CF$ tradeoff than the *NoMigrate* and greedy policies. This translates into a relevant performance gain of the *Graph-based* policies for the $IL$ metric in Figure 10(b); for $N_C$=1, the latency reduction provided by the *Graph-based* policy over the *NoMigrate* is up to 37% with $\Delta = \infty$, 13% with $\Delta = 5$.

In the analysis presented so far we considered WoT scenarios where the number of WTs is fixed at startup, hence the WT discovery process can be considered static over time. In Figures 10(c) and 11(a) we analyze the performance of M-WoT on a dynamic environment where the number of active WTs (and hence the amount of traffic and computational loads) is varying over time. More specifically, we setup the system with $N_{WT}$=0. Every 360 seconds, a new WT is created and added to the scenario; each WT consumes exactly one peer ($N_C$=1). Figure 10(c) shows the $NO$ metric over system evolution, expressed in time-slots; we remind that each time-slot corresponds to the execution of the Optimizer policy, and this event occurs every 190 seconds. It is easy to notice that the $NO$ metric increases significantly over time for the *NoMigrate* policy as a consequence of the creation of new WTs, and hence of the additional inter-host communication introduced in the system; vice versa, the *Graph-based* policies are able to adapt the WT allocation so that the $NO$ minimization goal is continuously met. The adaptiveness of M-WoT to network load conditions is further demonstrated by Figure 11(a) which shows the $TF$ metric over time slot; for the case of *Graph-based* with $\Delta = \infty$, the $TF$ increases over time as a consequence of the fact that -by adding new WTs in the system- larger connected components could be created and migrated to the same node. Vice versa, the *Graph-based* policies with $\Delta = 5$ and $\Delta = 2$ dynamically allocate the WTs so that the load-balancing constraint (reflected by the $\Delta$ value) is continuously satisfied.

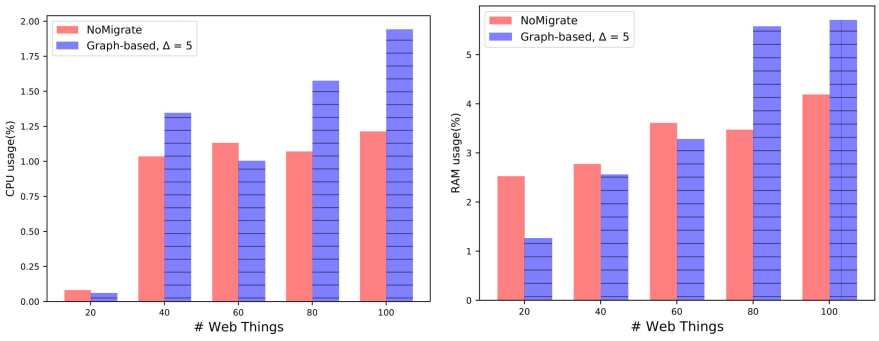
Finally, we evaluated the scalability of the proposed solution by monitoring the CPU and RAM consumption on the Orchestrator and Thing Directory node. Figures 12(a) and 12(b) show our findings. The results were obtained by sampling the container metrics every second, and then averaging the results for different number of deployed WTs. It is possible to notice that the consumption grows linearly but it is pretty negligible even with 100 WTs. Also, the overhead introduced by the *Graph-based* policy is only slightly higher than a *NoMigrate* policy, although M-WoT must execute the WT allocation procedure and the handoff procedure detailed in Section IV-D. Clearly, despite such positive results, the centralized Orchestrator might still become a performance bottleneck in large-scale WoT deployments; to address the issue, we can envisage the usage of a federated network of



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.2020.3045632, IEEE Access






(a) (b)

**FIGURE 12.** CPU load (Figure 12(a)) and RAM consumption (Figure 12(b)) of the Orchestrator for different numbers of deployed WTs.

Orchestrators, each controlling a specific region of nodes. Such distributed M-WoT framework would require proper data replication, load-balancing and gossiping mechanisms, which we plan to investigate as future works.

### B. USE-CASE ANALYSIS

We consider an IoT monitoring application, which mimics the operations of the SHM use-case presented in Section III. Specifically, we assume that a W3C WoT system has been designed to acquire and process the IoT data of a smart building. The WoT system involves three WTs:

- A *Sensing* WT, which performs data acquisition from an IoT sensor device (e.g. an accelerometer) through a Serial connection. More specifically, we assume that the *Sensing* WT can run in two modes, which differ from the sensor query frequency ($qf$), respectively the *Normal* mode (with 1 sample every 5 seconds) and *Warning* mode (with 1 sample every second); the mode switch (i.e. from Normal to Warning and vice versa) occurs when the last consecutive three readings are higher or lower than a static threshold; in other words, the granularity of the monitoring system is adjusted according to the detection of possible data anomalies.
- A *Processing* WT, which continuously receives the real-time measurements from the *Sensing* and applies a statistical method (i.e. the ARIMA regression) to forecast the next sensor values.
- A *Reporting* WT, which produces a notification (e.g. an alarm) based on the output of the *Processing* WT.

We abstract from the specific physical meaning of the IoT sensing values, while we focus on the capabilities of the WoT system to minimize the latency of processing specially in *Warning* mode, i.e. the time from when the data is acquired to when the forecast value is produced in output. We consider an initial setup with two nodes ($N_C$=2), respectively an edge server (connected to the IoT sensor device) and a remote cloud server on the Internet. Two scenarios are configured and compared in the evaluation analysis:

- *Migration OFF*. This represents the state-of-art WoT environment, where the WT migration is not enabled. The *Sensing* and *Reporting* WTs are deployed on the edge node, while the *Processing* WT is deployed on the cloud due to its higher computational power.
- *Migration ON*. This corresponds to the M-WoT environment, where the *Processing* WT is configured as migratable, i.e. it can be dynamically moved on the edge or on the cloud node based on the actual sensing mode. To this purpose, we deployed in the Optimizer a scenario-specific policy which checks the number of interactions between the *Sensing* and *Processing* WTs at each time-slot; in case such value is higher than a threshold (set equal to the $sf$ configuration in Normal Mode), the Optimizer realizes that the *Sensing* WT is working in Warning mode, and hence it migrates the *Processing* WT on the edge node, i.e. closer to the acquisition in order to minimize the communication latency. Otherwise, the *Processing* WT is allocated to the cloud node.

In the test-bed, the *Sensing* WT starts in *Normal* mode for 5 seconds, than it switches to *Warning* mode for 1 second, then again it repeats the same sequence for other two times. Figure 11(b) shows the $NO$ metric over the time-slots; for the *Migration OFF* configuration, the $NO$ value at each slot is equal to the number of messages exchanged by the *Sensing* and *Processing* WTs, since they are hosted by different nodes. The peaks correspond to intervals where the *Sensing* WT switches to the *Warning* mode. It is interesting to notice that: (*i*) the *Migration ON* configuration follows the same curve of the *Migration OFF* when the inter-host communication load is below a threshold; (*ii*) the $NO$ of the *Migration ON* is zero in correspondence of *Warning* periods, since the the *Processing* WT is migrated to the edge node, and hence all the communication occurs locally. Such action impacts the utilization of computational resources on the cloud/edge nodes as well as the processing latency. We report only the latter in Figure 11(c). We can notice the effectiveness of the M-WoT framework in terms of latency reduction for the *Migration ON*, which is more evident during the *Warning* periods since the edge-cloud communication delay is canceled.



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.2020.3045632, IEEE Access



## VIII. CONCLUSION

The W3C WoT constitutes a recent and promising approach to devise large-scale IoT systems composed of multiple, heterogeneous interacting components. The actual standard addresses both physical and virtual Web Things (WTs) however it assumes a static allocation of WTs to computational nodes, hence introducing potential performance bottlenecks in dynamic IoT scenarios characterized by time-varying WT interactions. In this paper, we aim at overcoming such issues by proposing M-WoT, a novel software framework supporting live migration and dynamic allocation of WTs among the computational nodes of an edge-cloud continuum. The proposed solution leverages the presence of uniform and well-defined WT interfaces (i.e. the TDs) in order to support stateful migration mechanisms of WTs as well as to enable dynamic group allocation of WTs to the nodes of the continuum. We proposed and implemented a centralized M-WoT architecture with novel software components for the WT software mobility, the WT handoff management, the context management, the Servient monitoring and the WoT deployment optimization. Regarding the latter, we addressed the problem of jointly maximizing the WoT data locality while balancing the workload on the computational resources; to this aim, a centralized heuristic computing a user-controllable trade-off between data locality and workload fairness has been proposed. Finally, we validated the performance gain of the proposed policy and the effectiveness of the overall M-WoT framework through two testbeds characterized by static/dynamic WT densities and traffic loads. Being a pioneeristic study on service mobility within W3C WoT environments, there is room for several extensions concerning the software architecture, the policy definition and the evaluation analysis. Indeed, the actual centralized architecture might suffer of single point-of-failure and scalability issues in large-scale environments; to this aim, a straightforward solution would be to employ multiple, distributed Orchestrator entities, each managing a subset of the available WTs and addressing collaborative tasks such as the Servient discovery and the context replication. Similarly, additional metrics can be gathered at the Servients' Monitoring API layer (e.g. the network bandwidth), and consequently novel multi-goal policies can be defined in the M-WoT Optimizer; given the high number of parameters potentially affecting the performance of WoT deployments, we are interested on the application of Machine Learning techniques (and mainly Deep Reiforcement Learning approaches) for seamless, adaptive deployment of WTs on distributed WoT environments. Finally, we plan to further test the effectiveness of WT migration mechanisms on real-world SHM scenarios of the MAC4PRO project [38], such as the monitoring of large civil and industrial structures (e.g. bridges and pressurized vessels); here, we will investigate the possibility to dynamically reconfigure the workloads of edge/cloud nodes based on the Quality of Service (QoS) requirements of the monitoring system and the real-time data streams among the WTs.

## APPENDIX

In the following, we include a subset of the Thing Report (`TR`) produced by the Monitoring API described in Section VI. The data-structure has been coded in `TypeScript` language.

```
export interface Report{
    id:string;
    hostID:string;
    serviceID:string;
    nodeStats: NodeStats;
    interactions:
    ↪  SerializableMap<string,InteractionReport>;
}

export interface NodeStats{
    cpu:Number,
    memory:BigInt
}

export interface InteractionReport{
    id:string;
    reconsumeCounts:number;
    url?:Url;
    propertyReports:
    ↪  SerializableMap<string,AffordanceReport>;
    actionReports:
    ↪  SerializableMap<string,AffordanceReport>;

    eventReports:
    ↪  SerializableMap<string,AffordanceReport>;
    summaryReport: AffordanceReport;
}

export interface AffordanceReport{
    calledTimes: number;
    computationalCost: bigint;
}
```

Listing 1: Report interface definition

## ACKNOWLEDGMENTS

This work has been funded by INAIL within the BRIC/2018, ID=11 framework, project MAC4PRO ("Smart maintenance of industrial plants and civil structures via innovative monitoring technologies and prognostic approaches").

## REFERENCES

[1] H. Xu, W. Yu, D. Griffith, and N. Golmie, "A survey on industrial internet of things: A cyber-physical systems perspective," IEEE Access, vol. 6, pp. 78238–78259, 2018.

[2] F. Jalali, T. Lynar, O. J. Smith, R. R. Kolluri, C. V. Hardgrove, N. Waywood, and F. Suits, "Dynamic Edge Fabric EnvironmenT: Seamless and Automatic Switching among Resources at the Edge of IoT Network and Cloud," Proceedings of the IEEE International Conference on Edge Computing (EDGE 2019), pp. 77–86, 2019.

[3] C. Zhang and Z. Zheng, "Task migration for mobile edge computing using deep reinforcement learning," Future Generation Computer Systems, vol. 96, pp. 111–118, 2019.

[4] X. Sun and N. Ansari, "EdgeIoT: Mobile Edge Computing for the Internet of Things," IEEE Communications Magazine, vol. 54, no. 12, pp. 22–29, 2016.

[5] S. Wang, J. Xu, N. Zhang, and Y. Liu, "A Survey on Service Migration in Mobile Edge Computing," IEEE Access, vol. 6, pp. 23511–23528, 2018.

[6] K. Ha, Y. Abe, Z. Chen, W. Hu, B. Amos, P. Pillai, and M. Satyanarayanan, "Adaptive vm handoff across cloudlets," Technical Report CMU-CS-15-113, 2015.

[7] C. Puliafito, E. Mingozzi, F. Longo, A. Puliafito, and O. Rana, "Fog computing for the internet of things: A survey," ACM Transaction on Internet Technologies, vol. 19, Apr. 2019.



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.2020.3045632, IEEE Access



[8] T. Taleb, A. Ksentini, and P. A. Frangoudis, "Follow-me cloud: When cloud services follow mobile users," IEEE Transactions on Cloud Computing, vol. 7, no. 2, pp. 369–382, 2019.
[9] P. Bellavista, A. Zanni, and M. Solimando, "A migration-enhanced edge computing support for mobile devices in hostile environments," in Proceedings of the 13th International Wireless Communications and Mobile Computing Conference (IEEE IWCMC 2017), pp. 957–962, 2017.
[10] C. Dupont, R. Giaffreda, and L. Capra, "Edge computing in IoT context: Horizontal and vertical Linux container migration," Proceedings of the Global Internet of Things Summit (GIoTS 2017), pp. 2–5, 2017.
[11] S. Wang, R. Urgaonkar, T. He, M. Zafer, K. Chan, and K. K. Leung, "Mobility-induced service migration in mobile micro-clouds," in Proceedings of the 2014 IEEE Military Communications Conference, pp. 835–840, 2014.
[12] "The internet of things: Mapping the value beyond the hype," McKInsey Global Institute, 2015.
[13] P. P. Ray, "A survey of iot cloud platforms," Future Computing and Informatics Journal, vol. 1, no. 1, pp. 35 – 46, 2016.
[14] P. Desai, A. Sheth, and P. Anantharam, "Semantic gateway as a service architecture for iot interoperability," 10 2014.
[15] "Wot reference architecture (w3c candidate recommendation 16 may 2019). http://www.w3.org/tr/wot-architecture/."
[16] Y. Ji, K. Ok, and W. S. Choi, "Web of things based iot standard interworking test case: Demo abstract," in Proceedings of the 5th Conference on Systems for Built Environments, BuildSys '18, p. 182–183, 2018.
[17] B. Klotz, S. K. Datta, D. Wilms, R. Troncy, and C. Bonnet, "A car as a semantic web thing: Motivation and demonstration," Proceedings of the 2018 Global Internet of Things Summit (GIoTS), pp. 1–6, 2018.
[18] V. Charpenay and S. Käbisch, "On modeling the physical world as a collection of things: The w3c thing description ontology," Proc. of the European Semantic Web Conference (ESWC), pp. 599–615, 2020.
[19] L. Sciullo, A. Trotta, L. Gigli, and M. Di Felice, "Deploying w3c web of things-based interoperable mash-up applications for industry 4.0: A testbed," in Wired/Wireless Internet Communications, pp. 3–14, Springer International Publishing, 2019.
[20] L. Sciullo, L. Gigli, A. Trotta, and M. D. Felice, "Wot store: Managing resources and applications on the web of things," Internet of Things, vol. 9, 2020.
[21] "Eclipse thingweb node-wot. https://github.com/eclipse/thingweb.node-wot."
[22] P. Yu, X. Ma, J. Cao, and J. Lu, "Application mobility in pervasive computing: A survey," Pervasive and Mobile Computing, vol. 9, no. 1, pp. 2 – 17, 2013. Special Section: Pervasive Sustainability.
[23] "Json-ld, json for linking data. https://json-ld.org."
[24] A. García Mangas and F. J. Suárez Alonso, "Wotpy: A framework for web of things applications," Computer Communications, vol. 147, pp. 235 – 251, 2019.
[25] P. Jamshidi, A. Ahmad, and C. Pahl, "Cloud migration research: A systematic review," IEEE Transactions on Cloud Computing, vol. 1, pp. 142 – 157, 02 2014.
[26] H. V. Hansen, V. Goebel, and T. Plagemann, "Tramp real-time application mobility platform," IEEE Transactions on Mobile Computing, vol. 16, no. 11, pp. 3236–3249, 2017.
[27] P. Varshney and Y. Simmhan, "Demystifying fog computing: Characterizing architectures, applications and abstractions," 2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC), May 2017.
[28] C. Puliafito, E. Mingozzi, C. Vallati, F. Longo, and G. Merlino, "Companion fog computing: supporting things mobility through container migration at the edge," Proceedings of the 2018 IEEE International Conference on Smart Computing, (IEEE SMARTCOMP 2018), pp. 97–105, 2018.
[29] H. Jin, S. Yan, C. Zhao, and D. Liang, "Pmc2o: Mobile cloudlet networking and performance analysis based on computation offloading," Ad Hoc Networks, vol. 58, pp. 86 – 98, 2017. Hybrid Wireless Ad Hoc Networks.
[30] H. Abdah, J. P. Barraca, and R. L. Aguiar, "QoS-aware service continuity in the virtualized edge," IEEE Access, vol. 7, pp. 51570–51588, 2019.
[31] K. Kientopf, S. Raza, S. Lansing, and M. Güneş, "Service management platform to support service migrations for IoT smart city applications," Proceedings of the IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (IEEE PIMRC 2018), vol. 2017-Octob, pp. 1–5, 2018.
[32] F. Ramalho and A. Neto, "Virtualization at the network edge: A performance comparison," in Proc. of the IEEE 17th International Symposium on A World of Wireless, Mobile and Multimedia Networks (IEEE WoWMoM 2016), pp. 1–6, 2016.
[33] R. Morabito and N. Beijar, "Enabling Data Processing at the Network Edge through Lightweight Virtualization Technologies," Proceedings of the IEEE International Conference on Sensing, Communication and Networking, SECON Workshops 2016, no. 607728, pp. 1–6, 2016.
[34] K. Jung, J. Gascon-Samson, and K. Pattabiraman, "Demo: ThingsMigrate - Platform-independent live-migration of javascript processes," Proceedings of the 2018 3rd ACM/IEEE Symposium on Edge Computing (SEC 2018), pp. 356–358, 2018.
[35] F. Bellifemine, G. Caire, A. Poggi, and G. Rimassa, "Jade: A software framework for developing multi-agent applications. lessons learned," Information and Software Technology, vol. 50, pp. 10–21, 01 2008.
[36] L. Alonso, J. Barbarán, J. Chen, M. Díaz, L. Llopis, and B. Rubio, "Middleware and communication technologies for structural health monitoring of critical infrastructures: A survey," Computer Standards and Interfaces, vol. 56, no. March 2017, pp. 83–100, 2018.
[37] C. J. A. Tokognon, B. Gao, G. Y. Tian, and Y. Yan, "Structural Health Monitoring Framework Based on Internet of Things: A Survey," IEEE Internet of Things Journal, vol. 4, no. 3, pp. 619–635, 2017.
[38] "Bric 2018 inail mac4pro project, https://site.unibo.it/mac4pro/it," 2019.

CRISTIANO AGUZZI received his Master Degree (summa cum Laude) in Computer Engineering in 2017 from the University of Bologna, Italy. He is currently pursuing the Ph.D. programme in Engineering and Information Technology for Structural Health and Environmental Monitoring and Risk Management (EIT4SEMM) at the University of Bologna. Furthermore, he is an invited expert in the W3C Web of Things Working group, where he is actively contributing to the Scripting API and Discovery specification documents. His research interests are semantic technologies, web of things, internet of things protocols, system software engineering and software dependability.

LORENZO GIGLI received his Master Degree (summa cum Laude) in Computer Science in 2019, from the University of Bologna, Italy. Currently he is a Research Fellow at the Department of Computer Science and Engineering (DISI), University of Bologna, Italy, working on the MAC4PRO project. He is part of the PeRvasive IoT SysteMs (PRISM) Research Laboratory directed by Prof. Marco Di Felice and co-founder of Modal Nodes, Anyprint S.r.l. and the research and development group of Epoca S.r.l. His field of study includes IoT / WoT technologies, distributed systems, containers and cloud architectures.

LUCA SCIULLO received his Master Degree (summa cum Laude) in Computer Science in 2017, from the University of Bologna, Italy. Currently, he is a Ph.D. candidate and a research fellow in Computer Science and Engineering at the University of Bologna, Italy. He was a Visiting Researcher at the Huawei European Research Center of Munich, Germany. He is part of the IoT Prism Lab directed by Prof.Marco Di Felice and Prof.Luciano Bononi. His research interests include wireless systems and protocols for emergency scenarios, wireless sensor networks, IoT systems and Web of Things.



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.2020.3045632, IEEE Access



ANGELO TROTTA is a Research Associate with the Department of Computer Science and Engineering, University of Bologna. He received his PhD. degree in computer science and engineering in 2017 from University of Bologna, Bologna, Italy. He was a Visiting Researcher with the Heudiasyc Laboratory, Sorbonne Universities, UTC, Compigne, France, and with Genesys-Laboratory, Northeastern University, Boston, MA, USA. He is co-founder of 'AI for People', an international association whose aim is to use the Artificial Intelligent technology for the social good. His current research includes nature inspired algorithms for self-organizing multirobots wireless systems, reinforcement learning and IoT.

MARCO DI FELICE is an Associate Professor of Computer Science with the University of Bologna, where he is the co-director of the PeRvasive IoT SysteMs (PRISM) laboratory. He received the Laurea (summa cum laude) and Ph.D. degrees in computer science from the University of Bologna, in 2004 and 2008, respectively. He was a Visiting Researcher with the Georgia Institute of Technology, Atlanta, GA, USA and with Northeastern University, Boston, MA, USA. He authored more than 100 papers on wireless and mobile systems, achieving three best paper awards for his scientific production. He is Associate Editor of the Elsevier Ad Hoc Networks journal. His research interests include self-organizing wireless networks, unmanned aerial systems, IoT, and context-aware computing.



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