WoT Micro Servient: Bringing the W3C Web of Things to Resource Constrained Edge Devices

WoT Micro Servient: Bringing the W3C Web of Things to Resource Constrained Edge Devices

Luca Sciullo*, Ivan Dimitry Ribeiro Zyrianoff*, Angelo Trotta*, Marco Di Felice*†,
* Department of Computer Science and Engineering, University of Bologna, Italy
† Advanced Research Center on Electronic Systems "Ercole De Castro", University of Bologna, Italy
Emails: {luca.sciullo, ivandimitry.ribeiro, angelo.trotta5, marco.difelice3}@unibo.it

***Abstract*—The chaotic growth of the Internet of Things (IoT) determined a fragmented landscape with a huge number of devices, technologies and platforms available on the market, and consequential issues of interoperability on many system deployments. The recent W3C Web of Things (WoT) standards aimed to ease the deployment of heterogeneous systems by introducing uniform and well-defined software interfaces among the systems' components. Although the WoT reference architecture is generic and agnostic to the target devices, its widespread adoption depends on the availability of specific tools named Servients, which enable the run-time operations of WoT applications. In this paper we aim at contributing to the adoption of the W3C WoT standards by presenting WoT Micro-Servient (WMS), a framework for bringing the WoT paradigm to the extreme edge of an IoT environment. Through WMS, developers can design, compile and install WoT applications on micro-controllers and embedded systems with constrained hardware capabilities. We describe the architecture and functionalities of the tool, and demonstrate its effectiveness in terms of reduced latency and energy consumption compared to the state-of-art proxy-based solution enabled by *Node-wot*, i.e. the official implementation of W3C WoT. Finally, we discuss a real-world application related to smart home, where WMS is used to enable a WoT-based remote monitoring and control of indoor plants, by enabling seamless integration between micro-controllers and mobile devices.**

***Index Terms*—Internet of Things, W3C Web of Things, embedded systems, performance evaluation**

I. Introduction

The last decade has been characterized by the exponential growth of the number of devices and technologies suitable for the Internet of Things (IoT) [1]. The reduction of costs and the miniaturization of electric components have led to the proliferation of micro-controllers able to sense the environment, to offload the data through wireless connectivity and/or to process it locally on the edge. Such devices can be considered as the backbone of IoT applications in several contexts, like domotics, smart agriculture, structural health monitoring, just to cite a few [2]. While the increasing number of such applications often translates in possible improvement of several aspects of our lives and society, it also highlights the severe technical issues posed by the lack of common standards and interfaces. From a different perspective, the interoperability can be considered a remunerative research challenge [3]: a recent report from McKinsey quantifies in 40% the additional IoT value that can be unlocked when achieving full interoperability among heterogeneous IoT systems [4]. Researchers have been addressing the interoperability problem by using different approaches [5], from the definition of middleware or custom frameworks at the network layers [6] [7], to the proposal of semantic approaches to achieve interoperability among heterogeneous devices and platforms at the data and application layer [8] [9]. Among others, the Web of Things (WoT) has gained considerable attention thanks to the popularity and well-known unifying nature of the Web [10]. In WoT approaches, Things are treated as Web resources and all the interactions towards and between them are mapped over the well-known Representational State Transfer (REST) pattern. However, due to the lack of a reference architecture, several different WoT proposals have been presented in the last years [11] [12], hence leading to a new stalemate. A radical change occurred in 2015, when the W3C Consortium started its activities with the ambitious goal to define a reference architecture for the Web Things and to achieve interoperability across IoT platforms and domains. One of the pillar of the standards is constituted by the concept of Web Thing [13] as the abstraction of a physical or a virtual entity whose metadata and interfaces are described according to a reference and well-defined structure, called Thing Description. This is instantiated as a software object by a program called Servient, which allows Web Things to exist as network resources and to be used by consumer applications. In addition, the W3C community gave concreteness to the WoT architecture by releasing the *Node-wot* framework [14] that implements a reference WoT Servient, and allows programmers to build WoT applications in Javascript. Despite the freshness of the standard, several WoT deployments have been proposed in the literature based on the *Node-wot* framework, from smart building [15] to automotive industry [16]. At the same time, we believe that the success of the W3C WoT initiative strongly depends on its widespread usage, and hence on the availability of a capillar software ecosystem of supporting tools. Unfortunately, the *Node-wot* tool, written on top of the NodeJS framework, cannot be executed on constrained IoT devices with limited energy lifetime and computational resources. This represents an important obstacle for the diffusion of the W3C WoT standard, since it limits the possibility to natively connect to the WoT a high percentage of edge IoT devices. Despite the possibility to adopt other architectural patterns (e.g. based on proxy and *System APIs* mechanisms [17]), the usage of native WoT solutions for embedded systems would definitively improve the performance of the entire WoT deployment, both

in terms of energy consumption and responsiveness.

In this paper we attempt to fill such gap by providing three main contributions:

- First, we design and implement a novel WoT-compliant *Servient* software -called WoT MicroServient (WMS) in the following- for resource-constrained edge IoT devices. Through the WMS tool, users can create and deploy a Web Thing which can be executed on a micro-controller in a native manner, so that other WoT-compliant applications can consume and interact with it (e.g., by reading a property value or invoking an action). We released the code of WMS as an open-source project, to allow third party usage by the W3C WoT community.
- Second, we demonstrate the effectiveness of the WMS tool through an IoT testbed characterized by the presence of heterogeneous, low-power devices. We compare its performance against a proxy-based solution enabled by the *Node-wot* framework, and quantify the improvements of the native WoT Servient for embedded systems in terms of reduced network latency and increased battery lifetime.
- Third, we illustrate a concrete use case of WMS in a smart-home scenario, by providing evidence of the new possibilities offered by the native WoT Servient for embedded systems, such as the ease of integration with mobile devices. More precisely, we design a *smart pot* system, through which the vital parameters of a plant can be monitored by a Mobile APP (both for Android and iOS) as well as asynchronous events related to environmental change.

The rest of this paper is structured as follows: in Section II we provide a brief review of the W3C WoT standards and its components, and we discuss other similar works on this field. Section III introduces the WMS architecture, while its implementation is described in Section IV. Section V provides a performance evaluation of WMS on a small IoT testbed, while a possible use-case scenario is presented in Section VI. Conclusions and future works are discussed in Section VII.

II. Related Work

The Internet of Things (IoT) is characterized by tens of different technologies, protocols, and architectures that create barriers for the interconnection of heterogeneous systems. The lack of interoperability represents one of the hardest challenge in the IoT landscape, and for this reason several attempts were made in the past year to counter this problem. Among others, the W3C Web of Things (WoT) represents a promising and recent approach for tackling the fragmentation in the IoT landscape [13]: starting from 2015, several universities and companies joined the W3C working group with the common goal of defining a set of standards to ease the deployment of IoT scenarios, regardless of the implementation details. Indeed, the W3C WoT architecture can be deployed on all the interesting use-cases for the IoT, like Industry 4.0, smart-home, smart agriculture, just to name a few.

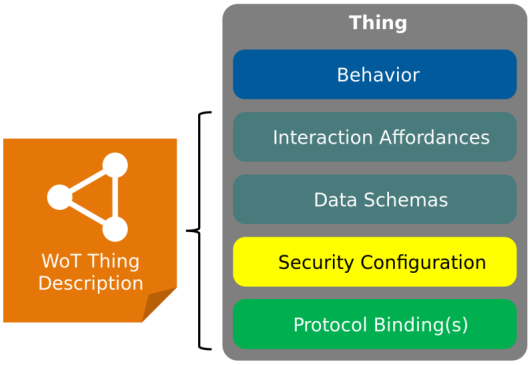


Fig. 1. The Thing Architecture [13]

The W3C WoT architecture is mainly based on four components:

- **Web Thing**: a Web Thing, most of the times simply called *Thing*, represents "an abstraction of a physical or a virtual entity whose metadata and interfaces are described by a WoT Thing Description, whereas a virtual entity is the composition of one or more Things." [13]. A Thing can be a device, a logical component of a device, a local hardware component, or even a logical entity such as a location (e.g., room or building).
- **WoT Thing Description (TD)**: the Thing Description [18] is the structured data that defines a Web Thing, like its metadata, together with its Interaction Affordances and links to other Things. By default, the TD is serialized by using the JSON-LD language and is structured according to the Property-Action-Event (PAE) paradigm: a Property represents an internal state variable of the Thing, each command that can be invoked on the Thing is mapped to an action, while each notification fired by the Thing is an event.
- **WoT Scripting API**: the WoT Scripting API [19] is an optional building block that describes an application programming interface (API) for the WoT interface, i.e, to allow scripts to discover, activate Things and to expose Things according to the *WoT Interactions* specified by the script itself. The APIs strictly adheres to the TD format, although it is possible to implement more abstract APIs on the top.
- **WoT Binding Templates**: the WoT Binding Templates [20] are a set of metadata that allows a TD to use communication protocols across different standards. More in detail, they enable a Consumer - technically an application client - to interact with Things that implement different communication strategies. For example, a Thing could implement a Machine-to-Machine (M2M) communication through CoAP with TLS security mechanism and CBOR as Content Type.

By the architectural perspective shown in Figure 1, a Thing is composed of 5 different layers: (*i*) the behaviour, (*ii*) the interaction affordances, (*iii*) the data schemas, (*iv*) the security

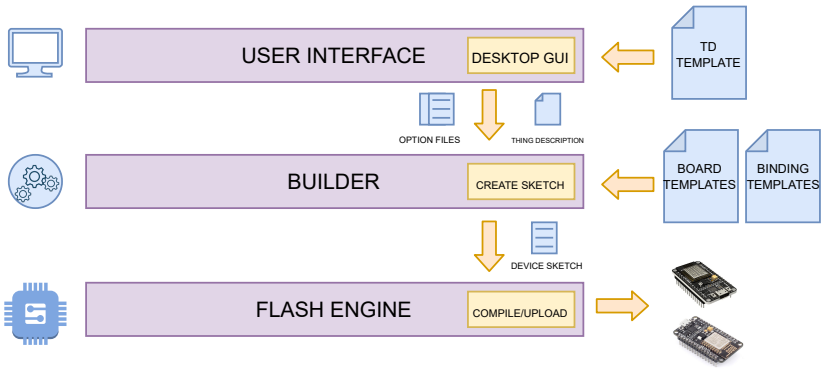


Fig. 2. The three-layer WMS architecture

configuration, and finally (*v*) the protocol bindings. The first layer defines the behaviour of a Thing and the handlers for the interaction affordances. These are described in the second layer, providing a model that specifies how *consumers* should interact with the Thing through abstract operations - hence without any reference to the specific protocol - and following the PAE (Properties, Actions, Events) paradigm previously introduced. The third layer explicitly details the *Information Model*, i.e., the payload structure of data exchanged between Thing and *consumer*. The fourth layer represents the access control mechanism to the Thing's affordances and, together with the last layer, it is in charge of mapping each interaction affordance to a concrete network protocol to be used, like HTTPS or CoAPS. All these architectural blocks are implemented in a software object called *Servient*. The *Servient* plays a crucial role in the WoT, since it represents the runtime instantiation of the Thing, i.e. it allows turning a Thing Description into a software object. As the word suggests, the Servient can act both as client and server. In the first case, a Servient is a *consumer* that hence interacts with an *Exposed Thing*, i.e., a Thing that has been instantiated by someone else and that is ready to be queried. On the contrary, in the second case the Servient is responsible to *expose* the Thing, hence enabling the possibility for a *consumer* to interact with that Thing. *Node-wot* [14] is the most used implementation for the Servient and it maintained directly by the W3C working group. It strictly adheres to the W3C standards and it is also considered the reference WoT Scripting APIs. The framework is written in *Typescript* and runs within NodeJS, despite there exists also a version for browsers. As a consequence of the increase in popularity of the W3C WoT initiative, additional Servient implementations for different programming languages have been proposed. In particular, we cite the *WotPy* [21] tool, an experimental implementation of a W3C WoT Runtime and the W3C WoT Scripting API in Python. In particular, the goal of the authors is to implement the W3C *WoT runtime* with complete support for all interaction verbs in all application layer protocols referenced by the specifications. A Java-based implementation of the WoT Servient is provided by the Smart Networks for Urban Participation (SANE) project [22]. The framework is intended to be used within its own stack or through a CLI interface. Another interesting initiative is represented by the work presented in [23], where authors propose a complete framework for generating custom WoT *Servients* starting from the Thing Descriptions. Nevertheless, despite the final code is written in C++, there is no explicit intention to target micro-controllers devices like in our study. Besides the *Servient* implementations, and as a consequence of the fact that W3C WoT is gaining attention in the research community, several other works related to the W3C WoT have been presented in the last few years. These studies can be classified into two broad categories: studies investigating the pro and cons of the W3C WoT standard (providing as well possible extensions/improvements), and studies describing new services enabled by the W3C WoT. As example of the first category, we cite the study in [24], where authors focused on the role models and lifecycles in IoT and its impact on the Thing Description (TD), since the latter is intended to be a static object while Things can change during their lifetime with respect to their physical and/or software components and specifications. Another example is represented by the work proposed in [25], where authors provide an accurate analysis about the security risks related to the TD modeling. In the second category, researches take advantage of the W3C WoT to propose new kind of interoperable systems or applications built upon Web Things. This is the case of [15], where authors propose a Building Energy Management System (BEMS) to enable the universal integration of both private and public systems through the W3C WoT, or the case of [16] where authors present the Vehicle Signal ontology (VSSo) - based on SOSA/SSN Observations and Actuation - to be used in conjunction with the W3C WoT for semantically describing a smart vehicle.

III. WMS Software Architecture

The WMS is a standalone framework that supports the deployment of W3C Web Things on edge IoT devices, i.e it allows a Thing TD to be natively exposed by a micro-controller device. The latter must be provided with an IP-based Internet connection (e.g. over Ethernet or Wi-Fi). As a result, other W3C WoT applications can consume the Thing and interacts

with the edge device through the standard WoT interface. We believe that this solution provides two main advantages. First, it brings the interoperability layer at the edge layer, hence it can reduce the deployment complexity on scenarios characterized by the presence of heterogeneous sensing devices using different protocols and software interfaces: in this case, a single WoT Mashup application is able to consume the IoT edge devices hiding the heterogeneity of network strategies through the WoT binding templates mechanism introduced in Section II. Second, our solution removes the need of ad-hoc gateways/bridges to integrate the IoT edge devices into the W3C WoT ecosystem; this again reduces the programming effort and it can also improve the system performance as better investigated in Section V. At the same time, while the WMS allows to expose WoT-compliant Things, hence serving as a WoT Servient in an effective manner, the tool itself cannot be precisely mapped to the *Servient* definition provided in Section II. Indeed, due to the limited hardware capabilities of the micro-controllers, WMS supports only a limited subset of the functionalities of a reference WoT *Servient* (e.g. the one available within the *Node-wot* tool [14]). At the same time, it acts also as a prototypal IDE framework to design, build and install Thing applications on embedded systems.
More in details, the WMS tool enables the user to implement the following operative flow:

1) *Device configuration*. The user is requested to insert all the extra-Thing information through the WMS GUI, like for instance the credentials for the Internet connection (e.g. Wi-Fi credentials). Similarly, he can add the extra pieces of code required by the specific embedded system in use, like for instance instantiating specific global variables needed by the onboard sensors.
2) *Protocol configuration*. The user can choose the *protocol bindings* among the ones available by the WMS (e.g. HTTP or Web sockets), as better explained in Section IV. The implementation of the *protocol bindings* is automatically added to the user code by the WMS tool.
3) *TD configuration*. The user can create the Thing Description thanks to the forms made available by the WMS GUI; in addition, he must provide the code implementing the *interaction affordances* handlers, which are application-dependent and hence cannot be generated automatically by our tool (the same also occurs with the *Node-wot* framework).
4) *Compilation and upload*. Finally, the user can select the target IoT *board* among those currenly supported by WMS and the right serial port; in addition, through the GUI, he can start the *compilation* and *upload* phase, which installs the WoT application (*Servient* plus Thing behaviour) on the physical device.

Based on the above description, we can highlight the peculiarities of the WMS project with respect to state-of-art WoT Servients like those described in [14] [21]. Those latter have been designed by leveraging on the full software stack available in a modern operating system. On the contrary, the WMS has been designed to adhere with the specifications of the WoT standard while coping with the limited hardware/software resources of embedded systems and micro-controllers. Indeed, WMS supports multiple different *protocol bindings* to expose a Thing in order to be consumed by external WoT-compliant applications; as a result, a Thing can be provided with multiple endpoints, which is a key requirement of the W3C WoT. Nevertheless, because of the limited memory of most of embedded systems, each *Servient* can host exactly one single Thing Application. We also remark that the current implementation of WMS supports only the *expose* functionality, but not the *consume* one, i.e. it does not allow the embedded Thing to consume external WoT services; we identified such feature as a future work, although we believe that it does not limit the potential of the current tool since -in most IoT devices- the micro-controllers expose basic functionalities such as transmit sensor measurements (sensors) or perform simple actions (actuators).
The WMS architecture is composed of three layers as depicted in Figure 2. The top layer is the *WMS GUI*, i.e., the software component easing the creation of WoT applications in a intuitive and guided manner. More in detail, through the GUI, an user can easily configure all the options required by the WoT application. The options can be divided into two macro-categories: the *auxiliary options* and the *structural options*. The first options include the network settings needed to connect the board to the Internet and the extra pieces of code required by the final sketch to properly work on the target micro-controller. The structural options define the *binding templates* to be used by the Thing and the definition of the *properties*, *actions*, and *events*, hence generating the TD. For this task, the GUI also takes advantage of predefined *Thing Description template* made available by the WMS tool; such template can be considered the skeleton of every W3C-complaint Thing since it contains the required metadata that every TD must contain according to the W3C WoT standard [18].
The outputs of the GUI layer (options and the self-generated TD) are passed as input to the *WMS Builder* layer. This is in charge of merging the TD-code with extra-code that implements the Servient functionalities. The added code includes the *Board Templates*, that are templates designed on purpose to address the specific board's capabilities, for instance for automatically *importing* the libraries needed by the onboard sensors or calling the function needed for the connection to the access point and the *Binding Templates* libraries, which implement the specific network strategies selected by the user at the previous layer.
Finally, once the final sketch of the WoT application has been assembled by the WMS, it is compiled and transferred to the physical device through the facitilities of the *Flash Engine* layer. This latter invokes CLI utilities that are specific of the IoT boards (e.g. Espressif ESP32) selected by the user through the GUI. At present, WMS supports selected Espressif boards (e.g. ESP32, ESP8266); we plan to extend the range of

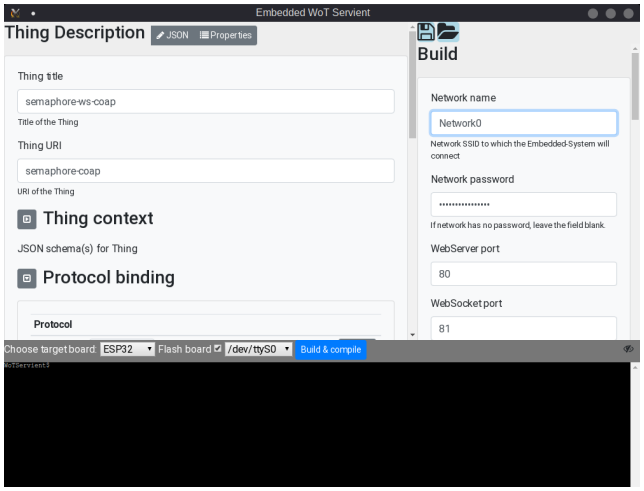


Fig. 3. The WMS GUI

supported boards in order to increase the tool popularity and usage from the W3C WoT community.

IV. WMS Implementation

We briefly discuss here the main technologies and solutions used for the implementation of the three layers of the WMS architecture presented in Section III. The code is released as an open-source project and it is available on Github[1]. The GUI is written in Javascript using the Electron framework[2], hence adopting HTML and CSS for the rendering of the application components, and it runs over *NodeJs V.10.21.x*. Furthermore, the *json-editor*[3] library was used for building the Thing Description according to the form compiled by the user in the application. As shown in the screenshot of Figure 3, the GUI is divided into three main panels, respectively one for the insertion of the *auxiliary options*, one for the insertion of the *structural options* - data and options required to build the Thing Description - and a terminal for the interaction with the flash engine.

The WMS builder is written in Python v.3.6.x and uses the *arduino-cli* tool[4] for compiling and uploading the sketch on the IoT device. The *board templates* are written in *Jinja*, a templating language widely used by Web programmers.

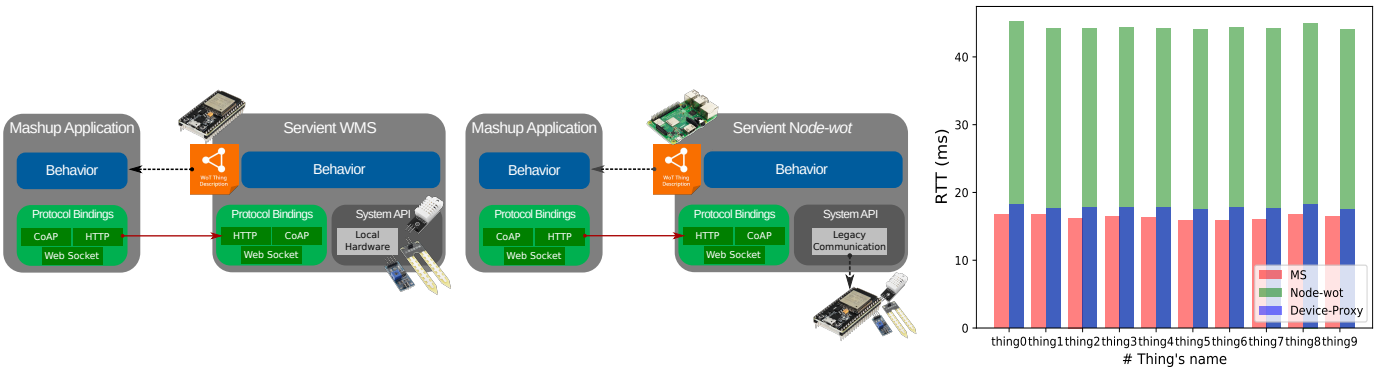
Finally, the binding templates libraries used have been created from scratch in order to handle the most popular IoT protocols. We defined a generic interface for handling the sending/receiving operations required by the Thing. This means that new protocol bindings can be easily added by simply writing the code of each method. Furthermore, in this way, the final sketch code of the Thing application can transparently call different protocol bindings without requiring any change to the application logic. In particular, we designed three different libraries that take advantage of existing libraries for microcontrollers, one for each protocol handled by WMS, i.e.: (i) *embeddedWoT_HTTP_LongPoll* implements HTTP - with the auxiliary addition of longpoll as *subprotocol* for the Web Thing events - (ii) *embeddedWoT_WebSocket* for the web sockets and (iii) *embeddedWoT_CoAP* for the support towards the CoAP protocol.

[1]https://github.com/UniBO-PRISMLab/micro-wot-servient
[2]https://www.electronjs.org/
[3]https://github.com/json-editor/json-editor
[4]https://github.com/arduino/arduino-cli

V. Performance Evaluation

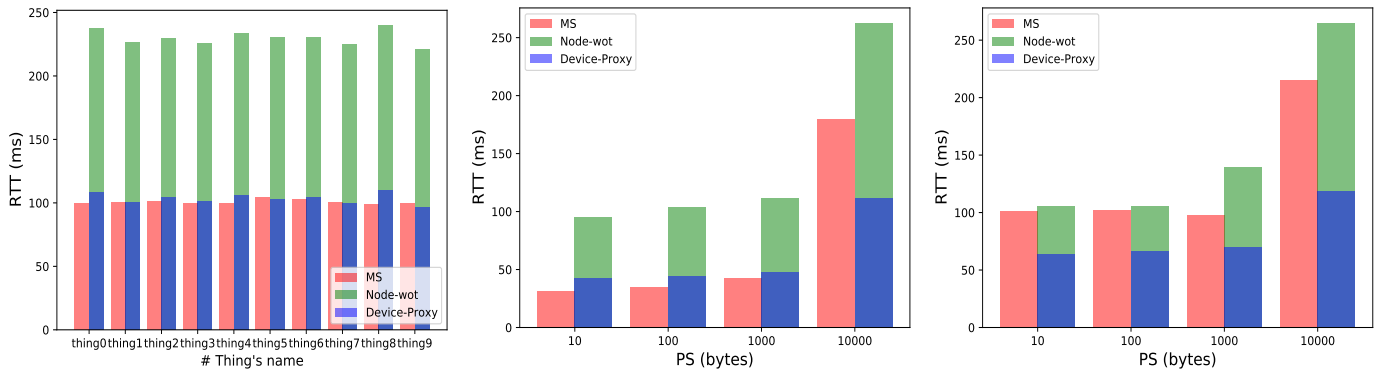
In this Section, we evaluate the performance of our solution against the usage of the regular *Node-wot Servient* in a Wireless Sensor Network (WSN) scenario. More in detail, we deployed a network of 10 IoT devices - respectively 5 NodeMCU and 5 ESP32 boards- able to collect environmental data (e.g. temperature/humidity) and to send it to a collector node. In the following, we denoted our solution simply as MS (Micro Servient) rather than WMS since we are not referring to the whole framework (e.g. including the GUI), but only on the code generated by WMS. In Figure 4(a) and Figure 4(b) we depict the two architectures of the evaluated scenarios. In the first scenario shown in Figure 4(a), each device is mapped to a Web Thing that is then exposed by the embedded MS; the collector node is a separate server, connected to the same WiFi network and hosting a simple *Mashup application* that queries each device in order to retrieve data and to compute the network statistics. All the Web Things expose a single Property that can be read, and whose response value depends on the *URI variable* passed to the query. The response is a buffer of bytes whose length is given by the aforementioned URI variable. Through this setting, the Mashup application can change on demand the payload size expected as response by each Thing. In the second scenario shown in Figure 4(b), the Web Things are hosted by a Raspberry Pi 3B+ node, that acts as a *proxy* component and that runs a *Node-wot Servient*; differently from the previous scenario, each device hosts a simple Web Server with a single endpoint. However, as in the previous case, the endpoint is used to retrieve a response value whose dimension depends on the *size* parameter in the *GET* request. Once the Mashup application, which runs in a separate server, issues a *Read Property* operation on a specific Thing, the proxy internally invokes a *GET* operation with the correct parameters on the right device. This implies that in the second scenario each data request involves two communication hops: from the Mashup application to the proxy with an HTTP client-server RESTful connection in order to be compliant with the W3C WoT standard, and from the proxy to the device using the same RESTful connection. This latter choice does not impact significantly the performance with respect to a simpler UDP direct connection as we demonstrated in [17]. We considered three different analysis:

1) in order to prove that the MS behaviour does not depend on the specific hardware in use, we evaluated the average Round Trip Time (RTT) experienced by each sensor. The Mashup application issues a total of 1000 requests for each device, while the payload size (PS) of the response message is set to 10B and 10KB, respectively.



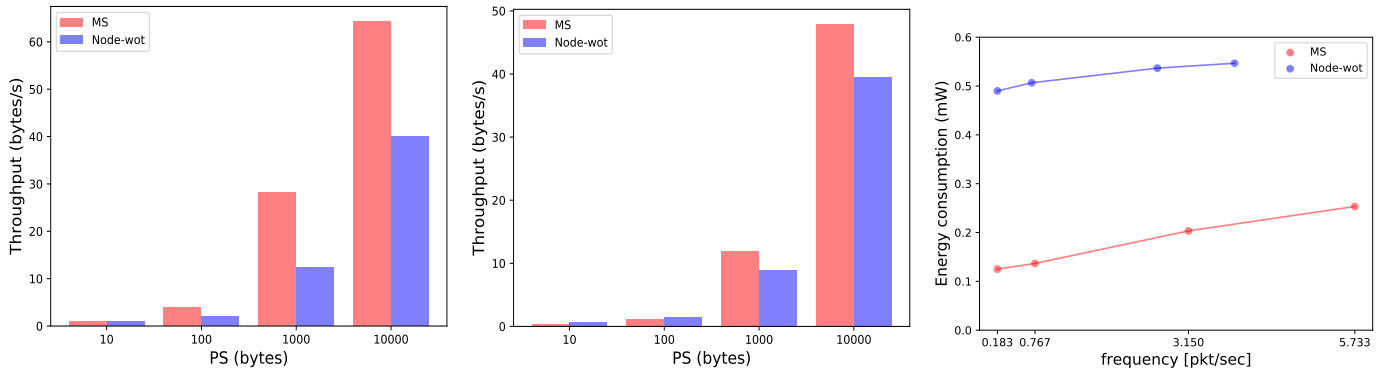
(a) (b) (c)

Fig. 4. The scenario of MS with Web Thing deployed directly on the microcontrollers is shown in Figure 4(a), while the scenario that uses the Raspberry Pi as *proxy* is shown in Figure 4(b). The average RTT for different micro-controllers is shown in Figure 4(c), with a fixed payload (PS) size of 10b.



(a) (b) (c)

Fig. 5. The average RTT for different micro-controllers is shown in Figure 5(a) with a fixed payload size (PS) of 10kb. The average RTT for varying PS, with delay equal to $d = 0$ and $d = 100$ms between two consecutive requests are shown in Figures 5(b) and 5(c), respectively



(a) (b) (c)

Fig. 6. The average throughput for varying PS, with delay equal to $d = 0$ and $d = 100$ms between two consecutive requests are shown in Figure 6(a) and Figure 6(b). The energy consumption of the two scenarios for varying transmission frequencies is shown in Figure 6(c).

2) In order to show the performance gain of MS in terms of delay and throughput under varying workloads, we compute the average RTT and throughput metrics as a function of the payload size (PS) and of the delay between consecutive requests (d).
3) In order to show the advantage in terms of energy saving of the entire system deployment, we compare the energy consumption of two scenarios when varying the frequency of requests generated by the Mashup application.

The results of the first analysis are shown in Figure 4(c) and Figure 5(a). For the *Node-wot* scenario, we plot separately the

RTT on each communication hop (denoted by different colors in each bar). From Figure 4(c) we can notice that the average RTT is approximately equal to $18ms$ for the MS, while it is two times larger for *Node-wot*. Same considerations apply also to Figure 5(a), where the average RTT is around $100ms$ when using MS, and more than $230ms$ when introducing the proxy. The delay reduction can be explained by the fact that in MS there is only one communication hop between data producers and data consumer. Also, in both Figures we depicted the RTT value for each IoT device; despite the hardware hetorogeneity, we can notice that the MS performance are not affected by the target device.

The results of the second analysis are depicted in Figure 5(b) and Figure 5(c); more specifically, we report the average RTT metric as a function of PS on the x-axis, for two different configurations of the delay parameter, i.e. $d = 0ms$ (sender is issuing a new request as soon as receiving the reply from the previous one) in Figure 5(b) and $d = 100ms$ in Figure 5(c), respectively. Clearly, the average RTT metric increases with the PS parameter, and - as expected - it is considerably lower in MS due to the shortest route between each IoT device and the connector. Same considerations can be derived from Figure 6(a) (d=0) and Figure 6(b) ($d = 100ms$) where we depict the network throughput over PS.

The previous analysis demonstrates that the increased complexity on edge devices does not impact the system performance. We further elaborate on this issue by analyzing the overall energy consumption of the two scenarios, again for varying workloads reflected by the frequency of requests issued by the Mashup application. The results are depicted in Figure 6(c): on the y-axis we report the total energy depletion of the MS, which is the summation of the energy depletion of the IoT devices, and the same metric for the *Node-wot* scenario, which includes also the energy overhead introduced by the proxy device. Although the energy consumption of the IoT device may be slightly higher in MS due to the increased software complexity on the edge node, the overall system consumption is considerably reduced in MS due to the removal of the proxy node. Due to the limited range communication between the IoT device and the proxy, only few devices may be connected to each proxy and hence we expect such performance gain to be even more effective on large-scale deployments.

VI. Use-case

In this Section we describe a possible application of the WMS tool for a smart-home use case, and specifically on the design and implementation of a smart pot. The application consists of an *ESP32* micro-controller equipped with a soil hygrometer sensor which senses the soil humidity and with a *DHT22* sensor which senses the air temperature and humidity. The micro-controller is connected to the sensors and hence in close proximity to the pot of the plant that we want to monitor. The sensor data is read by a mobile application, so that users can continuously monitor the environmental conditions of the plant, and act accordingly (e.g. supply water). The system has been deployed through the W3C WoT, specifically we created a Web Thing for the *SmartPot-WoT* Thing that models the three sensors, and a mobile application that consumes the previous Web Thing and is in charge of getting the data from the sensors and returning them to the end-user. Figure 7 depicts the use-case, with the mobile application on the left and the IoT sensors connected to the micro-controller on the right.

Interaction Affordance	Name	Type
Property	humidity	number
Property	hygrometer	number
Property	temperature	number
Event	cold-air	number

TABLE I
The Interaction Affordances of the SmartPot-WoT Web Thing

The SmartPot-WoT Web Thing is deployed directly on the micro-controller using the proposed WMS framework. As described in Section III, we first defined the Interaction Affordances for the SmartPot-WoT Thing. Specifically for the use-case, we defined three *properties*, corresponding to the three deployed sensors and one *event* that activates each time the air temperature is below a certain value (here, we set the threshold to 10 degree Celsius). Through the WMS GUI, we set the properties and the event, and we wrote the code for the Interaction Affordance handlers - that are in charge of dealing with the local hardware. To this purpose, we used the ESP32 libraries to communicate with the DHT22 and the hygrometer sensors. The values of these sensors are then bound to the three Thing properties that are listed in Table VI: *humidity*, *hygrometer*, and *temperature*. The event *cold-air* activates an on-board LED and triggers an alert message to the mobile application. The final TD for the SmartPot-WoT has been automatically generated by the WMS framework and loaded directly on the micro-controller device.

The mobile application has been implemented as an hybrid application, hence leveraging on Web technologies and in particular on the Ionic framework[5] and the *Node-wot* browser library. The goal of the mobile application is to directly communicate with the SmartPot-WoT Web Thing in order to monitor all its properties and to receive the events fired by the smart pot. The mobile application consumes the TD exposed by the SmartPot-WoT, then it periodically retrieves the properties' values via the HTTP protocol, that is chosen among the available Protocol Bindings supported by the WMS. Figure 7 shows a screenshot of the mobile application with the current sensor values.

VII. Conclusions

In this paper we described the design and the implementation of WoT MicroServient (WMS), a framework for the implementation of W3C Web Things on micro-controllers and devices characterized by constrained computational and energy resources. More in detail, WMS allows users to easily define the behaviour of the Things and to generate the code that

[5]https://ionicframework.com/

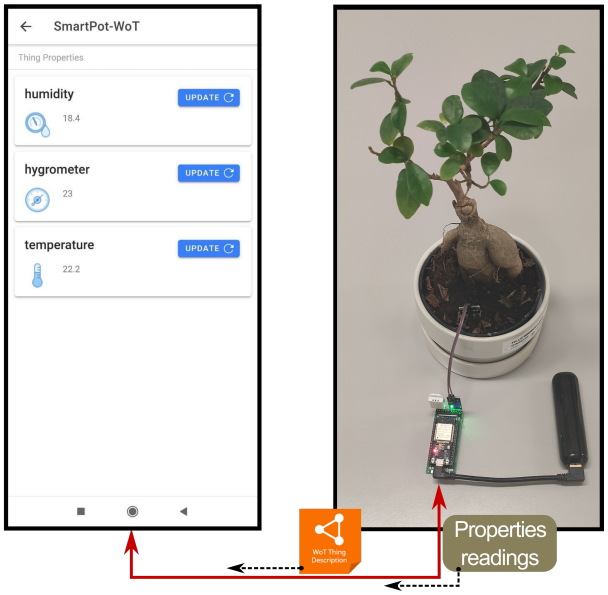


Fig. 7. The plant care scenario composed by (on the right side) the *ESP32* micro-controller that senses the environmental parameters and by (on the left side) the Android application that displays the sensors' value

can be then uploaded on the physical device; as a result, other WoT applications can natively consume the Thing and interact with the embedded device. WMS explicitly responds to the need of extending the list of devices that can natively support the emerging W3C WoT standards, hence without relying on additional hardware and software components that could degrade the system performance. To this purpose, we demonstrated through a small test-bed that the network performance (RTT, throughput) can significantly improve through the Micro Servient (MS) when compared to state-of-art solution based on the official *Node-wot* framework implementation which needs an extra hop of communication. Finally, we demonstrated the practical usefulness of the tool on a smart home use-case: specifically, we showed how to monitor vital information of a plant through the use of a Web Thing running on a micro-controller that is directly monitored through a Mobile APP available both for Android and iOS. Future works include the extension of the WMS tool in order to support additional micro-controller devices, the design and the implementation of the *consume* operation through which the micro-controller will be able to use data produced by other Web Things, further evaluation on real-world IoT scenarios characterized by the presence of heterogeneous sensor devices, such as Structural Health Monitoring applications and the optimization of the code automatically generated by the WMS.

Acknowledgements

The authors would like to thank Federico Rachelli for his valuable contributions on the software design and implementation. 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] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, "Internet of things: A survey on enabling technologies, protocols, and applications," *IEEE communications surveys & tutorials*, vol. 17, no. 4, pp. 2347–2376, 2015.

[2] L. Atzori, A. Iera, and G. Morabito, "The internet of things: A survey," *Computer networks*, vol. 54, no. 15, pp. 2787–2805, 2010.

[3] (2017) Interoperability and the Internet of Things. [Online]. Available: https://www.ndpanalytics.com/report-interoperability-and-iot

[4] "The internet of things: Mapping the value beyond the hype," *McKInsey Global Institute*, 2015.

[5] M. Noura, M. Atiquzzaman, and M. Gaedke, "Interoperability in internet of things: Taxonomies and open challenges," *Mobile Networks and Applications*, vol. 24, no. 3, pp. 796–809, 2019.

[6] Open Connectivity Foundation (OCF). (2015) IoTivity. [Online]. Available: https://iotivity.org/

[7] (2015) Arrowhead Framework. [Online]. Available: https://www.arrowhead.eu/

[8] A. Kamilaris and M. I. Ali, "Do "web of things platforms" truly follow the web of things?" in *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*. IEEE, 2016, pp. 496–501.

[9] M. Younan, S. Khattab, and R. Bahgat, "Wotsf: A framework for searching in the web of things," in *Proceedings of the 10th International Conference on Informatics and Systems*, 2016, pp. 278–285.

[10] M. Blackstock and R. Lea, "Toward interoperability in a web of things," in *Proceedings of the 2013 ACM conference on Pervasive and ubiquitous computing adjunct publication*, 2013, pp. 1565–1574.

[11] F. Paganelli, S. Turchi, and D. Giuli, "A web of things framework for restful applications and its experimentation in a smart city," *IEEE Systems Journal*, vol. 10, no. 4, pp. 1412–1423, 2014.

[12] L. Mainetti, V. Mighali, and L. Patrono, "A software architecture enabling the web of things," *IEEE Internet of Things Journal*, vol. 2, no. 6, pp. 445–454, 2015.

[13] W3C Working Group. (2021) WoT Reference Architecture (W3C Recommendation 9 April 2020). [Online]. Available: http://www.w3.org/TR/wot-architecture/

[14] ——. (2019) Eclipse Thingweb node-wot. Source repository. [Online]. Available: https://github.com/eclipse/thingweb.node-wot

[15] D. Ibaseta, A. García, M. Álvarez, B. Garzón, F. Díez, P. Coca, C. Del Pero, and J. Molleda, "Monitoring and control of energy consumption in buildings using wot: A novel approach for smart retrofit," *Sustainable Cities and Society*, vol. 65, p. 102637, 2021.

[16] B. Klotz, S. K. Datta, D. Wilms, R. Troncy, and C. Bonnet, "A car as a semantic web thing: motivation and demonstration," in *2018 Global Internet of Things Summit (GIoTS)*. IEEE, 2018, pp. 1–6.

[17] 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 *International Conference on Wired/Wireless Internet Communication*. Springer, 2019, pp. 3–14.

[18] S. Käbisch, T. Kamiya, M. McCool, V. Charpenay, and M. Kovatsch, "Web of things (wot) thing description," W3C Recommendation, Apr. 2020, https://www.w3.org/TR/wot-thing-description.

[19] Z. Kis, D. Peintner, C. Aguzzi, J. Hund, and K. Nimura, "Web of things (wot) scripting api," W3C Recommendation, Nov. 2020, https://www.w3.org/TR/wot-scripting-api/.

[20] M. Koster and E. Korkan, "Web of things (wot) binding templates," W3C Recommendation, Nov. 2020, https://www.w3.org/TR/wot-binding-templates/.

[21] A. G. Mangas and F. J. S. Alonso, "Wotpy: A framework for web of things applications," *Computer Communications*, vol. 147, pp. 235–251, 2019.

[22] "Smart Networks for Urban Participation (SANE)," 2021. [Online]. Available: https://sane.city/

[23] M. Iglesias-Urkia, A. Gómez, D. Casado-Mansilla, and A. Urbieta, "Automatic generation of web of things servients using thing descriptions," *Personal and Ubiquitous Computing*, pp. 1–17, 2020.

[24] M. Blank, H. Lahbaiel, S. Kaebisch, and H. Kosch, "Role models and lifecycles in iot and their impact on the w3c wot thing description," in *Proceedings of the 8th International Conference on the Internet of Things*, 2018, pp. 1–4.

[25] M. McCool and E. Reshetova, "Distributed security risks and opportunities in the w3c web of things," in *Workshop on Decentralized IoT Security and Standards (DISS)*, 2018.