A Cloud-based and Dynamic DNS approach to enable the Web of Things

Zakaria Benomar*, Francesco Longo*†, Giovanni Merlino*†, Antonio Puliafito*†
*Department of Engineering, University of Messina, Italy
Email: {zbenomar,flongo,gmerlino,apuliafito}@unime.it
† CINI: National Interuniversity Consortium for Informatics, Rome, Italy

Abstract—Thanks to the evolution in the hardware and software fields, the Internet usage scope is continuously covering smaller and resource-constrained devices. Such devices, commonly called Internet of Things (IoT) devices, with sensing/actuation capabilities, are becoming capable of managing the complexity of communications over the Internet. Nevertheless, the IoT ecosystem is still fragmented due to the different used vertical solutions. This heterogeneity makes IoT devices/systems unable to communicate seamlessly, leading to limited cooperation and tightly coupled deployments. To deal with the interoperability issues in IoT, we propose, in this paper, a Cloud-based approach with a Dynamic Domain Name System (DDNS) mechanism enabling the IoT devices to communicate using the Representational state transfer (REST) model: an approach that follows the Web of things (WoT) paradigm. In particular, the system makes the IoT devices' hosted resources (e.g., sensors and actuators) able to be steered using globally resolvable (over the Internet) Uniform Resource Locators (URLs) even when deployed behind Network Address Translators (NATs) and firewalls. The system we conceived requires only one public registered domain name to associate, for all the distributed IoT devices, sub-domains of the public one while using a clever routing mechanism. An online implementation of the testbed is provided to show the feasibility of the approach. Further, a performance evaluation of the system is reported to assess the resource usage of the solution.

Index Terms—Internet of Things, Cloud computing, Web of Things, REST, Cyber Physical Systems, OpenStack, Web services.

1 Introduction

The significant progress in the field of embedded systems' has lead to a considerable decrease in smart devices' prices, hence their adoption in different fields (e.g., industry, healthcare, environmental monitoring). In this context, a trend aiming at embedding computational capabilities within standard objects/assets (e.g., industrial appliances, machines, etc.) has arisen under the aegis of pervasive/ubiquitous computing [1]. By enhancing the processing capabilities of standard objects/assets, they become smart enough to achieve practical tasks and provide added value services. As one of the widely known concepts of pervasive computing, the Internet of Things (IoT) [2] is a major paradigm that does not only aim at enhancing the computational capabilities of objects, but it goes beyond this view to connect objects to the Internet whislt making them able to communicate, sense, and interact with their local resources and/or surrounding environments. This IoT view of bridging objects to the Internet and associating digital artifacts of the physical world will undoubtedly mark a revolution in how people interact and use everyday objects. Systems nowadays are fundamentally grounded in the behavior of people/citizens who may generate data to conceive new services or make decisions based on the data collected [3]. In this regard, awareness is growing in relation to the social dimension of the role of such Cyber-Physical Systems (CPSs) [4]. Indeed, scholars are increasingly addressing these systems as Cyber-Physical Social Systems (CPSS) [5]. The massive growth in terms of the number and type of devices that CPSS deployments may push for would bring exciting opportunities, yet with several management and deployment challenges to tackle. Accordingly, we need, on the one hand, efficient approaches to deal with IoT infrastructure management and, on the other hand, suitable methods to integrate the devices within the Internet wisdom and bridge their compatibility gap [6]. In fact, the interoperability between devices from different manufacturers remains a major burden for IoT applications' developers owing to the different used vertical silos, communication protocols, and data formats [7].

The Web is considered today as the communication linchpin of the Internet. Web protocols provide suitable mechanisms connecting machines (e.g., computers, servers) from different manufacturers among each other. Systems' interoperability, in this context, is a key enabler to foster new services in distributed environments [8]. Considering the ever more lightweight footprint and resource requirements of (tiny) Web servers developed in recent years, IoT nodes are becoming increasingly suitable to host such services (i.e., Web servers). By extension, their resources can be abstracted and exposed through the same hypermedia they serve, a paradigm that is known as Web of Things (WoT) [9]. The IoT nodes then enhance their capability of being IP-enabled devices connected to the Internet to become able to communicate with other devices or Web systems using the same language. In such a homogeneous environment, the smart objects can provide their functionalities (e.g., sensed data) via Representational state transfer (REST) interactions; thus, new IoT-based services/applications become easier to conceive.

This paper is an extension of our previous work [10] that accommodates relatively powerful IP-enabled devices (e.g., single-board computers) to be part of the Web and expose their hosted resources (e.g., sensors and actuators) using RESTful APIs (i.e., based on HTTP). In this paper, we enhance our system by implementing a mechanism capable of enabling server-initiated (i.e., Cloud-triggered) connectivity to any device-hosted resource over User Datagram Protocol (UDP) transport (before, we used only TCP). That said, in a WoT context, the system can support plain (TCP-based) HTTP as well as other, newer, HTTP versions, such as HTTP over Quick UDP Internet Connections (QUIC). We also strengthen the approach by leveraging HTTPS instead of HTTP, using tooling for automatic issuance and validation of X.509 certificates in the IoT landscape.

The rest of the paper is organized as follows. Section II surveys the relevant literature. Section III, introduces the enabling technologies/approaches related to the design of our system. Section IV describes the tunneling approach we used to expose services (e.g., Web servers) hosted on IoT nodes deployed at the network edge to any audience over the Internet. Section V introduces our WoT-oriented Dynamic Domain Name System (DDNS) as well as a detailed functional workflow. Section VI describes an online accessible testbed that uses our WoT system. We also report the performance results of the system. Section VII closes the paper and hints at promising future work.

2 Related Works

2.1 RESTful Web services and IoT

The Web services concept plays a relevant role in enabling interoperable machine-to-machine communications over the Internet. On this basis, two main categories of Web services are used: the REST-compliant Web services and the arbitrary Web Services (WS-*). The difference between the two models resides in their way of managing communications. While the first approach uses systematic and well-defined operations (e.g., GET, POST, DELETE), the second approach, instead, makes use of arbitrary operations (e.g., using Simple Object Access Protocol (SOAP)). In this context, the choice of the suitable model to implement depends strictly on the use case. In [11], the authors recommend opting for the RESTful oriented Web services when dealing with ad-hoc services over the Web (so-called mashups). Likewise, in [12], authors outline the relevance of using the REST model in an IoT context to enable the WoT communication model and the design of flow-based applications (using tools such as Node-RED).

From the performance point of view, the RESTful compliant Web services are a convenient choice compared to WS* when it comes to IoT. In fact, the RESTful Web services are characterized by less overhead and parsing complexity while providing stateless interactions [13]. Besides, the fact that WS* supports only the Extensible Markup Language (XML) as an encoding pattern makes it unsuitable to be adopted in specific IoT scenarios such as low power and low data rate sensor networks. On the other hand, REST affords more choices (e.g., plain text and JavaScript Object Notation (JSON)) that make it a flexible model to be adapted depending on the use case. In particular, in IoT scenarios, the use of JSON ensure higher performance implementations than XML [14]. In [15], Yazar et al. made a performance comparison of different parameters (e.g., memory footprint, power consumption) between the two Web services approaches when deployed on constrained IoT nodes. The authors conclude that the REST model outperforms WS*. Besides, authors in [16] outline that the REST architectural design is more fit to be used with constrained IoT networks (e.g., Wireless Sensors Networks (WSNs)) as it can seamlessly be mapped to other protocols (e.g., Constrained Application Protocol (CoAP)).

Another critical parameter to consider when choosing the suitable Web services paradigm in IoT is the software development aspect. In order to promote external developers' communities in conceiving new IoT-based services/applications, providing a consistent software architectural style with accurate APIs is critical in adopting IoT services on a larger scale. In [17], the authors investigated applications developers' preferences. The results show that developers prefer the REST architectural style as it is less complex to implement and use.

In the literature, a set of IoT-based architectures and testbeds opt for the RESTful model. In [18], a system for smart cities energy management is described. In [19] another IoT REST-based system for warehousing is introduced. A system designed for domotic applications is presented in [20] with simulated performance results. Other solutions such as [21] and [22] promote the WoT architectural design. Although these aforementioned architectures/testbeds are relevant and promising, they did not tackle the important aspect of having IoT nodes deployed within restricted and masqueraded networks. Indeed, having IoT nodes or gateways reachable over the Internet is not always ensured or at least requires some specific networking configurations (e.g., port forwarding at the router's level). In most IoT deployments, nodes are often deployed behind NAT/firewalls. Thus, their reachability should not be taken for granted as it is a critical parameter in adopting the WoT model.

2.2 Secure Web services in IoT

In IoT deployments, particularly the WoT architectural design, we expect to have services (e.g., Web servers) and, therefore, data exposed publicly over the Web. Considering the distributed nature of IoT deployments, ensuring secure data transmission is challenging. In such an environment where the infrastructure (i.e., IoT nodes) is geographically distributed with networks' designers having limited control over the infrastructure (e.g., when it is contributed by volunteers), chances of malicious users to falsify the data being transmitted or even spoof IoT nodes is considerably high [23]. As the WoT paradigm aims to merge IoT with the Web, enabling HTTPS-based communications is a convenient choice. In fact, HTTPS is actually the de-facto protocol used in the Web that ensures peers identification and prevents against communications sniffing and data manipulation [24].

To ensure secure communications between peers, Certification Authorities (CAs) tend to make servers' administrators follow a set of manual tasks while configuring their domains/servers. However, in the kind of IoT

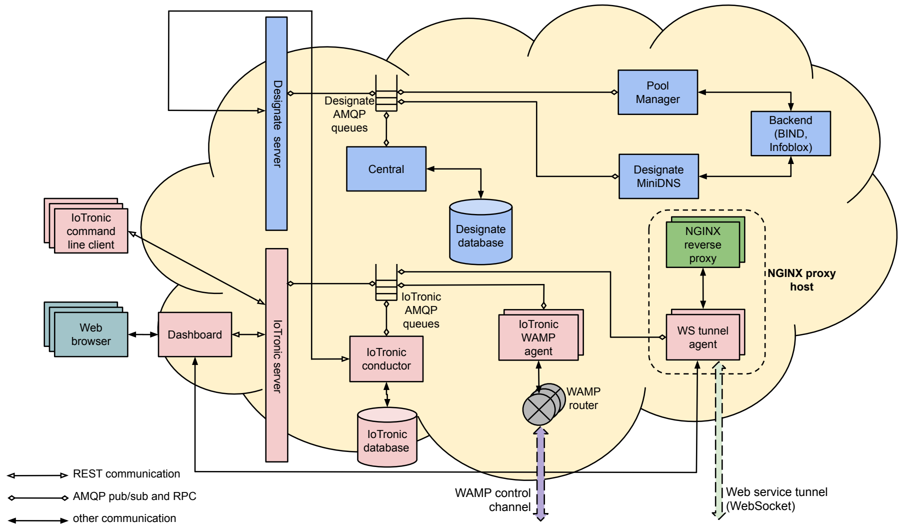


Fig. 1: IoTronic and Designate integration design.

deployment we are targeting, the manual configuration of servers' certificates may become an error-prone task besides being time-consuming [24] due to the high-density of IoT networks. In this context, the use of an efficient mechanism such as the Automated Certificate Management Environment (ACME) [25] protocol is essential in enabling secure data exchange in IoT and avoid any manual configuration. Even more so, efforts are actually on the way to enhance the ACME Protocol by using a distributed trust mechanism based on Bockchain [26].

3 Background on S4T, DNSaaS and WoT approaches

In this section, we give details about the different concepts we made use of to come up with our system. We introduce our Stack4Things middleware, then we describe briefly the architecture of the OpenStack DNS-as-a-Service (DNSaaS) subsystem (i.e., Designate), and finally, we discuss the WoT integration patterns.

3.1 Stack4Things

Stack4Things (S4T) [27] is an OpenStack-based platform tailored for IoT infrastructure management. Based on a set of refined and new mechanisms, S4T deals with the typical IoT deployments' constraints (e.g., NATs/firewalls traversal, connectivity disruption). The S4T design is composed of two parts: a Cloud-based deployment hosting the subsystem called IoTronic (see red subsystem in Fig. 1) and a number of (distributed) IoT nodes deployed at the network edge while hosting the device-side agent named Lightning-Rod (LR) (See Fig. 2). IoTronic is a subsystem designed with respect to the standard architecture of OpenStack services therefore, its integration with other subsystems (e.g., Keystone, Neutron) can seamlessly be realized to provide advanced user-facing features [28] [29]. To overwhelm networking issues such as NATs/firewalls traversals, communications between IoTronic and LR are based on a permanent full-duplex communication channel based on WebSocket with a reverse tunneling mechanism. As hardware setup for the IoT nodes we are targetting, we made a choice, on purpose, to use relatively smart devices (e.g., single-board computers) that are microprocessor (MPU)-powered. This specific hardware setup enables the nodes to host a (minimal) Linux distribution (e.g., OpenWRT) together with a set of runtime environments (e.g., C, Node.js, and Python).

3.2 The OpenStack DNSaaS system: Designate

In OpenStack deployments, the DNSaaS system, Designate, manages DNS records created by the Cloud users/tenants. The robustness of the subsystem relies on enabling automated updates to DNS records based on events triggered by other OpenStack services. As for all OpenStack subsystems, the architectural design of Designate is modular and, thus, composed of several components that communicate via Advanced Message Queuing Protocol (AMQP) (see blue subsystem in Fig. 1). Specifically, the system is made of an API server (designate-API) that exposes the systems' functionalities, a central controller (designate-central) managing the access to the database, and finally, the Designate-pool-manager and Designate-mdns that handle the communication with the backend DNS servers (e.g., BIND and PowerDNS).

3.3 Web of Things integration patterns

In this subsection, we discuss the WoT architectures as well as their environments characteristics and constraints.

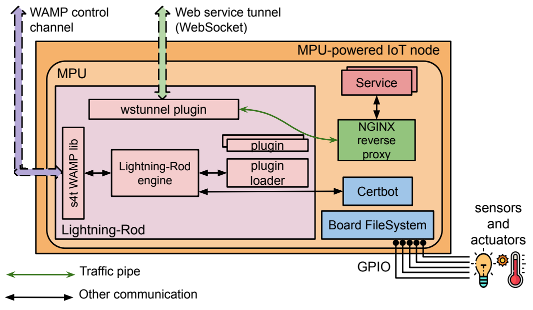


Fig. 2: S4T Lightning-Rod architecture.

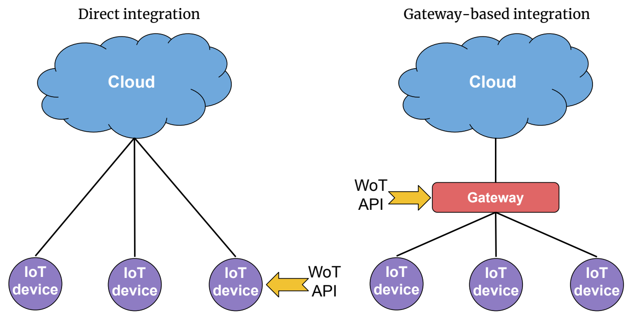


Fig. 3: WoT integration patterns.

3.3.1 Direct integration

In the direct integration pattern, the IoT nodes are powerful enough to be directly bridged to the Web wisdom (see the left integration pattern in Fig. 3). In particular, IoT nodes, in this case, are capable of managing TCP/IP and HTTP-based communications. Albeit appearing a simple architecture to deploy, it is not that straightforward. Indeed, in most of the IoT deployments, the IoT nodes do not have public IP addresses (i.e., they are deployed within masqueraded IPv4 networks) and, consequently, no publicly resolvable domain names. Therefore, exposing the IoT nodes' hosted resources (e.g., sensors and actuators) publicly and making them reachable through globally resolvable URLs is challenging.

3.3.2 Gateway-based integration

In specific IoT deployments, mainly when many IoT nodes are needed, opting for relatively powerful IP-enabled devices is not financially viable. In such environments, the devices used do not have enough resources to manage the complexity of Web-based communications (also because of energy constraints). Networks' designers then opt for intermediate well-powered smart IP-enabled gateways in order to bridge the constrained devices to the Web (see the right integration pattern in Fig. 3). In this architecture, the gateway is responsible of forwarding/routing requests to the different constrained devices.

4 Exposing Cloud-enabled IoT-hosted services

This section describes the tunneling approach we conceived to expose, publicly, via the Cloud, TCP or UDP services hosted on IoT nodes deployed at the network edge. This feature is then leveraged as an infrastructure-level enabling mechanism to expose the IoT nodes' hosted resources (i.e., sensors and actuators) by assigning them publicly resolvable domain names (a detailed description of this system is reported in the next section).

As remote infrastructure, the deployed IoT nodes will be reachable over restrictive and even masqueraded IPv4 networks. In this case, the unique assumption that can (always) be considered valid is outgoing Web traffic being authorized; that is, only device-initiated (TCP) communications over standard HTTP/HTTPS ports are permitted. To cope with the constraint mentioned above, we opted for a standard TCP-based HTTP-borne full-duplex communication, namely WebSocket (WS) coupled with a reverse (tunneling) mechanism, i.e., IoT nodes will initiate the process of setting up the tunnel to the Cloud. Besides providing bidirectional flows between two ends, WS is a network-agnostic protocol by turning communications into standard HTTP interactions. Any mechanism then that exploits WS can overwhelm the issues of reaching environments that block Web-unrelated traffic. An interesting feature that can be enabled using WS is establishing TCP tunnels over WS, a way to get client-initiated connectivity to any server/device-side service. For our system, we designed and implemented a suitable reverse tunneling over WS[1] solution as an approach to provide connectivity to any IoT node-hosted service. Even though we opt for the WS protocol that adds additional overhead to the packets and requires a handshake to establish the client-server connection as a transport medium for our tunneling system, those parameters do not significantly affect the system performance. Indeed, authors in [30] outline that during long WS sessions, the impact of those parameters becomes insignificant just after few messages' exchanges (see also subsection 6.3).

We depict in Fig. 4 the design of the system as well as the process of a WS reverse tunnel (rtunnel) creation. To expose a TCP service hosted on a remote IoT node, the rtunnel client (which is pre-configured with the IP address of the rtunnel server) sends a WS connection request to the rtunnel server (i.e., the Cloud). In particular, the request specifies a TCP port to be used on the server-side. Once the rtunnel server receives the request, it brings up a TCP server listening on the port indicated, and a WS connection used as a control channel is established (green arrow in Fig. 4). When an external TCP client connects to the TCP server on the rtunnel server (i.e., Cloud), the rtunnel client and server manage this event by instantiating a new WS connection (yellow arrows in Fig. 4), and the TCP session gets piped to it. On the rtunnel client-side, a similar mechanism is used by bringing up a TCP client connecting to the local (or remote) service involved and pipe afterward the traffic to the tunnel.

We depict in Fig. 4 all the three scenarios the system may enable. A first scenario, as we mentioned before, is node-provided access to a service running on the node itself (`service 1` in Fig.4). To illustrate a use case, we can consider a Web server exposing the IoT node's hosted resources (e.g., sensors and actuators). The second use case, still the service runs on the IoT node itself, but it can forward/map requests to other constrained nodes behind it (`service 2` in Fig.4). A relevant use case we can mention is a proxy relaying HTTP requests to a constrained network (e.g., 6LoWPAN-based) using an HTTP-COAP proxy. Last and not least, the system can also provide access to services running on other devices deployed on the same LAN as the gateway (`service 3` in Fig.4). In this case, the gateway, a relatively powerful node deals with the complexity of setting up the WS tunnel; therefore, only applications' flows are forwarded to the constrained devices behind it.

To expose UDP-based services hosted on remote IoT nodes, we accommodate UDP flows on both sides (i.e., rtunnel client and server) to fit the WS tunnels TCP server/-client. Specifically, we used an executable, namely `Socat`, that establish bidirectional byte streams between two extremities (i.e., ports) whatever the transport protocol used

1. https://github.com/MDSLab/wstun

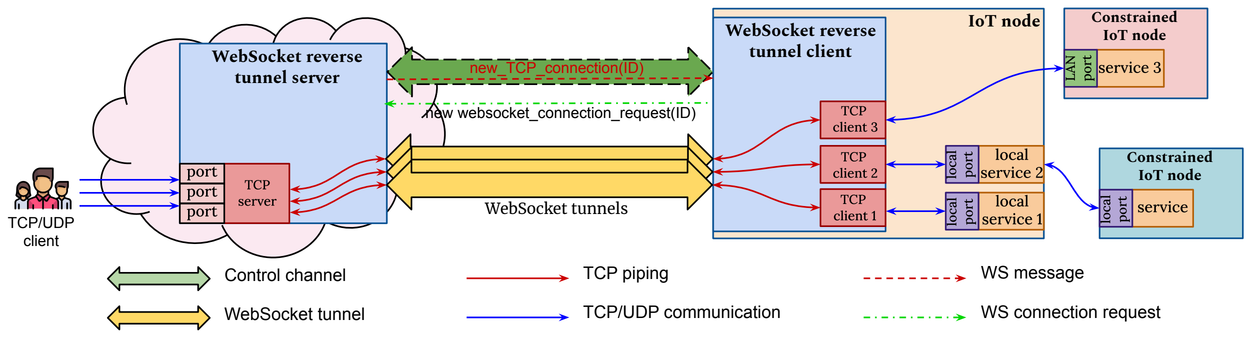


Fig. 4: The WebSocket tunneling system.

(UDP or TCP) and transfers data between them using TCP. Consequently, UDP packets on both sides (i.e., client and server) get adapted to fit the (TCP-based) WS tunnel. This way, a UDP flow coming from an external UDP client towards the Cloud gets tunneled and reaches services deployed at the network edge.

5 S4T Dynamic DNS system

In this work, we aim at introducing a novel approach to overcome interoperability issues in IoT by enabling RESTful interactions. By assigning (publicly) resolvable domain names to the Web servers hosted on the distributed IoT nodes, we can expose their resources (i.e., sensors and actuators) using REST APIs. Thus, we are moving towards a more decentralized yet developer-side uniform IoT ecosystem. In the following, we describe our OpenStack-based WoT system that uses IoTronic capabilities together with the DNSaaS subsystem, Designate.

5.1 Overview of the S4T Web of Things system

We report in this subsection an overview of our tunneled reverse proxying approach capable of assigning globally resolvable domain names to Web servers deployed within IPv4 masqueraded networks. In particular, the approach uses only one publicly registered domain name to make the distributed Web servers appear to be hosted on sub-domains of the public one. That says, no public IP or domain name on the local IoT node is needed.

To conceive our system, we are considering, as mentioned before, that the IoT nodes are typically deployed behind NATs and firewalls; therefore, they do not have routable public IP addresses. To expose the IoT nodes' hosted Web servers, and by transitivity sensors/actuators, our approach uses Designate to deal with the management of DNS records (which are sub-domains of the public one) associated with the edge-based Web servers. To route requests based on the URLs indicated, IoTronic and LR, together with two NGINX reverse proxies[2] deal with requests' forwarding (see Figs.1 and 2). In particular, for each sub-domain created and assigned to a Web server, IoTronic and LR manage the instantiation of a (reverse) WS tunnel between the Cloud and the IoT node (see Section 4). Afterward, clients' requests in the destination of the Web server are forwarded (through the WS tunnel) using the NGINX reverse proxies' rules managed, automatically, by IoTronic and LR.

2. https://www.nginx.com

Regarding the DV certificate issuance and validation, once the reverse WS tunnel gets created and the two NGINX proxies configured, LR manages the X.509 certificate issuance and validation using the ACME-based Let's Encrypt CA client, namely Certbot. Clients then, such as Web browsers and mobile applications, can communicate using HTTPS with the edge-based Web server running on the IoT node. Specifically, a client request is sent to the Cloud NGINX reverse proxy that manages, based on the URL indicated in the request (specifically, the sub-domain part), the forwarding/routing of the request through the suitable WS tunnel to reach the IoT node/service concerned.

5.2 Workflow of exposing a service

In the following, we report a detailed description of the workflow when a user wants to expose, publicly, a Web server running on an IoT node. The full domain name we are considering to be assigned to the Web server is *web-server.node-A.example.com*. Consequently, the registered public domain used is *example.com* whereas the rest of the domain name, i.e., *node-A* and *web-server* are managed by our DNS system to identify the IoT node and the service concerned, respectively (an IoT node can host multiple services). Regarding the Cloud NGINX reverse proxy, we consider that the host where it is running has as an IP address 1.1.1.1 while the Web server hosted on the IoT node runs on port 9000. We assume that the IoT node has already gone through a set of verification processes (e.g., authentication) required by S4T; thus, it is registered and connected to the Cloud. The following list of sequences takes place when exposing the Web server, with (low-level) operations as depicted and numbered in Fig. 5):

1) The user send a request to expose a service (e.g., Web server) running on a specific IoT node using the OpenStack dashboard or the Command-Line Interface (CLI). In particular, the user chooses the service name (in this case, named *web-server*), DNS zone that indicate the IoT node where the server is running (i.e., *node-A*), and the port on which the Web server is listening (i.e., 9000).
2) The dashboard/CLI sends a REST request to the IoTronic API server that pushes a new message into AMQP queue.

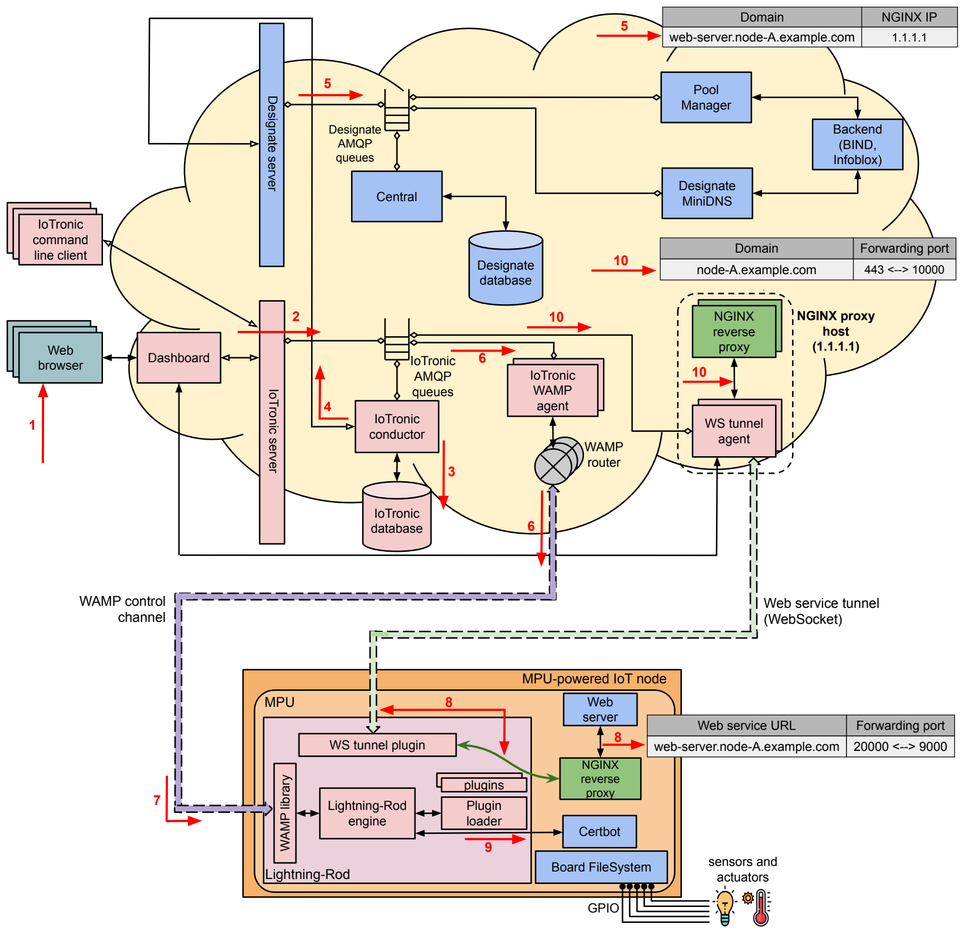


Fig. 5: A detailed workflow description of exposing a Web server hosted on an edge IoT node.

3) The IoTronic conductor gets the message from the queue and does a check using its database. Specifically, IoTronic verifies if the IoT node is registered and online. It also looks up the Web Application Messaging Protocol (WAMP) agent and the WS tunnel agent managing the IoT node concerned (the WS tunnel agent is co-hosted on the same machine where the Cloud NGINX proxy is deployed. it has as IP 1.1.1.1).
4) IoTronic interacts with the Designate API server to request the creation of a type A DNS record using the IP of the NGINX proxy from step 3 (i.e., 1.1.1.1) and the information provided by the user in step 1 (i.e., the sub-domain names: *web-server* and *node-A*).
5) Designate creates the DNS record within its backend environment.
6) The IoTronic conductor sends a Remote Procedure Call (RPC) to the WAMP agent managing the IoT node to start the configuration process. Subsequently, the IoTronic WAMP agent sends a WAMP-based RPC to the NGINX proxy running on the IoT node to instantiate a WS tunnel to the Cloud. A random port number is generated and sent as an argument of the RPC to create the tunnel. In our scenario, we consider as random port number 10000.
7) The IoT node receives the RPC through the WAMP library.
8) The LR agent manages the setup of the (reverse) WS tunnel by generating a random port number (20000 in our scenario) and taking into consideration the port number received as an argument of the RPC (i.e., 10000). Next, LR configures the NGINX reverse proxy to forward requests in destination of the URL *web-service.node-A.example.com* to the port on which the Web server is listening (specified by the user in step 1, 9000 in this case)
9) LR manages, using the local Certbot daemon, the issuance of the DV certificate from the Let's Encrypt CA.
10) The IoTronic conductor interacts with the WS tunnel agent managing the IoT node (using an RPC) to configure the Cloud-based NGINX reverse proxy. Specifically, the proxy is used to forward requests in the destination of *node-A.example.com* to the WS tunnel instantiated in step 6. That said, requests reaching the Cloud NGINX reverse proxy on port 443 and having as destination *node-A.example.com* are forwarded to port 10000 (i.e., the port on which the WS tunnel is running).

To illustrate a functional workflow that uses the mechanisms described before, we report in subsection 6.2 a detailed workflow of a request being routed using the tunneling system.

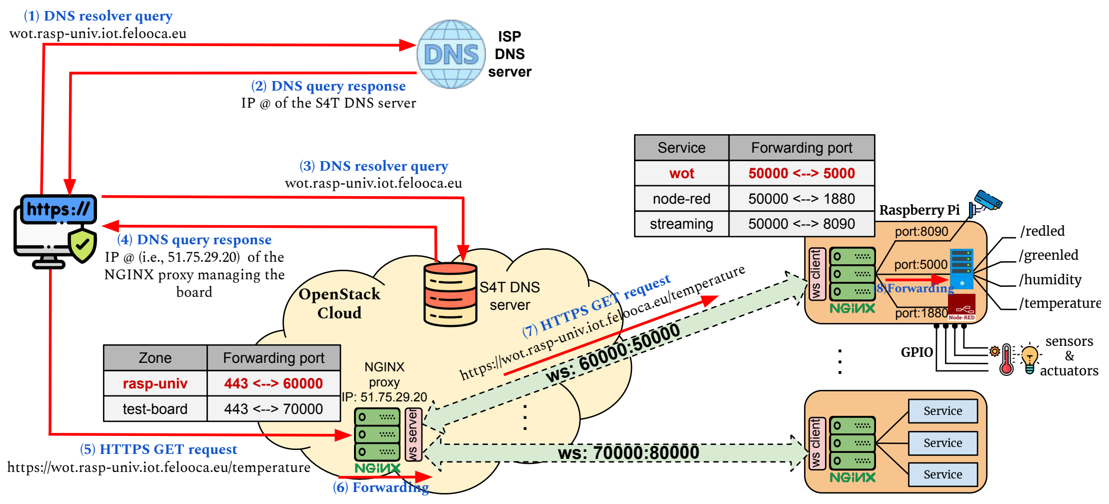


Fig. 6: The Stack4Things-based WoT routing mechanism.

6 Implementation and Experimental Results

In this section, we provide an online accessible testbed powered by the S4T DDNS system. We also report a set of experiments results to assess the performance of the approach.

6.1 Testbed description

To prove the feasibility of our approach, we used a Raspberry Pi single-board computer to host a Web server and therefore expose publicly a set of hosted sensors/actuators as depicted in Fig. 6. The OpenStack environment, including our IoTronic system, is hosted at the Department of Engineering, University of Messina, Italy. The Raspberry Pi runs the device-side agent (*LR*) agent and hosts a Web application that uses the built-in Flask Web server running on port 5000. We used the approach presented in Section 5 to expose the Web server. As URL, we choose *https://wot.rasp-univ.iot.felooca.eu* under which */temperature*, */humidity*, */redled* and */greenled* are made available as resources. We remind that the choice of the URL is up to the user except for its public part (in this case, *felooca.eu*). Other services hosted on the same board are exposed as well, such as a video streaming from a Web camera using *streaming.rasp-univ.iot.felooca.eu* as URL and a Node-RED instance[3]. We mention here that the registered (and globally resolvable) domain name the approach uses is *felooca.eu*. The *iot* sub-domain was created for management purposes as the domain (i.e., *felooca.eu*) is used for production by the smartme.IO[4] spin-off company. The *iot* sub-domain does not affect the workflows described before: it is transparent with regard to the approach here presented. To be aligned with the descriptions presented in section 5, we can consider that our public domain is *iot.felooca.eu*.

By enabling the use of globally resolvable URLs associated with geographically distributed services deployed at the network edge, we are able to create mashups based on distributed Web services. For instance, we used the video streaming URL to incorporate the streaming on the Web page. The mechanisms provided by the system make user agents, for example, web browsers, able to interact with the board-hosted resources using HTTPS. The reader can access the following URL: *https://wot.rasp-univ.iot.felooca.eu* that point out the Flask Web server hosted on the Raspberry Pi. In this deployment, all the requests made to get sensors values (i.e., temperature and humidity), as well as the LEDs status, are HTTPS-based GET requests. The Linux command-line tool, `curl`, can also be used to retrieve the value of a metric from a resource, for example, temperature: *curl -X GET -H "Accept: application" https://wot.rasp-univ.iot.felooca.eu/temperature*. Even more so, the reader can interact, in real-time, with the two LEDs using the corresponding Web page widgets by sending HTTPS POST requests under the hood.

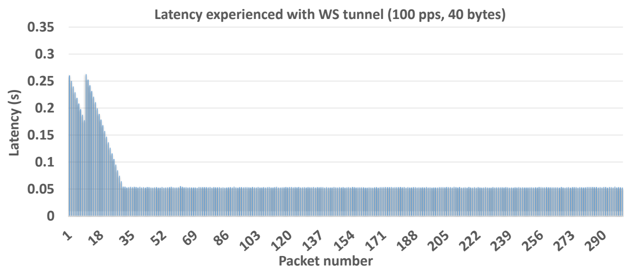
6.2 Functional workflow

We report in the following the functional workflow when a Web client requests the value of the temperature sensor. The URL we consider is the one mentioned before and made available over the Web: *https://wot.rasp-univ.iot.felooca.eu* and the resource involved is */temperature*. The complete workflow is reported in Fig. 6 (we skip the different TCP/TLS handshakes for a matter of simplicity and to make the workflow easier to grasp):

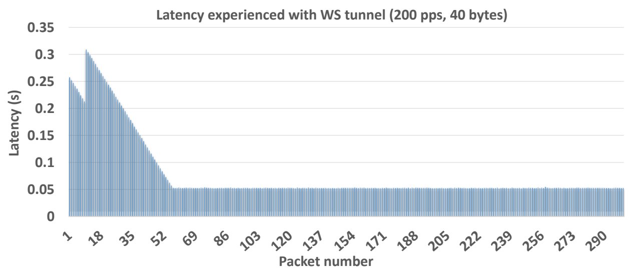
1) The client (a Web browser in this case) sends a DNS resolver query about the URL (i.e., wot.rasp-univ.iot.felooca.eu) to the Internet Service Provider (ISP) public DNS server.
2) The public DNS server sends a response back to the client about the *felooca* domain name. The response contains the public IP address of the S4T Cloud DNS server.
3) The client sends a new DNS request to the S4T DNS server.

3. We exposed three services that the readers can access:
- **The web page**: https://wot.rasp-univ.iot.felooca.eu
- **The video streaming**: https://streaming.rasp-univ.iot.felooca.eu/?action=stream/
- **A Node-RED instance**: https://node-red.rasp-univ.iot.felooca.eu

4. https://smartme.io/

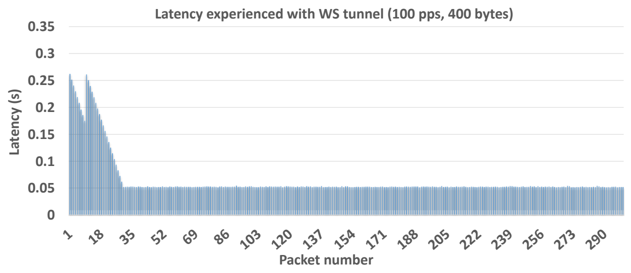


(a) Latency experienced with the WS tunnel using a packet rate of 100 pps and 40 bytes as payload length.

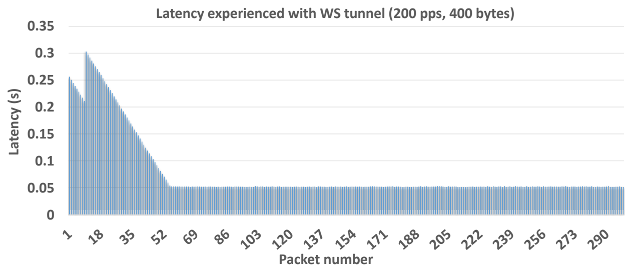


(b) Latency experienced with the WS tunnel using a packet rate of 200 pps and 40 bytes as payload length.

Fig. 7: Latency measured with the WS tunnel when using packets with 40 bytes of payload.



(a) Latency experienced with the WS tunnel using a packet rate of 100 pps and 400 bytes as payload length.



(b) Latency experienced with the WS tunnel using a packet rate of 200 pps and 400 bytes as payload length.

Fig. 8: Latency measured with the WS tunnel when using packets with 400 bytes of payload.

4) The DNS server sends a response back to the client with the IP address of the NGINX reverse proxy managing the board concerned (in the testbed we are making accessible online, the proxy IP address is 51.75.29.206).
5) The client sends an HTTP GET request to the IP address specified by the DNS server from the previous step (i.e., 51.75.29.206).
6) Once the NGINX reverse proxy receives the HTTP request, it checks the URL mentioned. Based on the sub-domain specified (i.e., *rasp-univ*) that indicates the board, the NGINX reverse proxy forwards the request through the appropriate WS tunnel connecting the Cloud to that board. Specifically, in our online testbed, requests received on port 443 and having as URL *https://wot.rasp-univ.iot.felooca.eu* are forwarded via the WS tunnel running on port 60000.
7) The request reaches the board NGINX reverse proxy through the WS tunnel.
8) The NGINX reverse proxy checks the URL of the request specifically, the service name (i.e., *wot*). Based on its forwarding rules (created when exposing the Web server, see Section 5.2), the proxy forwards the request to the port on which the Web server is listening. In our scenario, the service is named *wot* and runs on port 5000. As a result, the request reaches the Web server, and the response (value of the */temperature* resource) travels back the same way.

6.3 Performance evaluation

In this subsection, we evaluate firstly the performance of the tunneling approach we implemented. Then, we present the results of deploying the whole solution (i.e., WS tunnel client and NGINX) on an IoT node.

To assess the performance of the tunneling mechanism, we used two Virtual Machines (VMs) as UDP client and server communicating through a WS tunnel. Each of the VMs has two vCPUs and 2 GB of RAM. The VMs are hosted on a 2020 Intel i5 MacBook Pro while being timely synchronized using Network Time Protocol (NTP).

To evaluate the impact of the WS tunnel on the latency it may introduce, we fixed the latency between the 2 VMs (using the virtual bridge interface) at 50 ms. We measured then the delay between the timestamp when a packet with 40 bytes of data is sent by the client and the timestamp when the same packet reaches the server. For the sake of clarity, we state here that the latency we intend to evaluate is a one-way measure. Specifically, we used UDP as it brings more flexibility and control over the testbed since the packet sending rate can be controlled with higher granularity.

Figs. 7a and 7b depict the results of our experiments at 100 and 200 packets per second (pps), respectively. The packet number (x-axis) represents the sequential number of the packets received by the UDP server at a given packets sending rate. We mention here that to accommodate the UDP traffic sent/received by the client and make it tunneled through the (TCP) WS tunnel, we used the `Socat` tool as discussed in Section 4. As reported in Fig. 7a, the latency stabilizes at around 50 ms (i.e., the latency between the two VMs) after some packet transmissions. This result shows that the WS tunnel's impact on latency is negligible during long sessions. At the beginning of the communication, the higher latency value is due to the three-way TCP handshake used to set up the WS tunnel (the second peak is attributed

	Packets with 40 bytes of payload			*Packets with 400 bytes of payload*		
Packets generation rate (pps)	*WS tunnel client CPU usage (%)*	*Socat CPU usage (%)*	*Total CPU usage (%)*	*WS tunnel client CPU usage (%)*	*Socat CPU usage (%)*	*Total CPU usage (%)*
100	2.48	0.38	2.86	2.6	0.4	3
200	4.37	0.76	5.13	4.31	0.71	5.02
300	5.2	1.12	6.32	5.23	1.09	6.32
400	5.78	1.4	7.18	5.62	1.42	7.04
500	6.18	1.66	7.84	6.11	1.58	7.69

TABLE 1: WS tunnel client and `Socat` CPU usage with different packets' payload lengths.

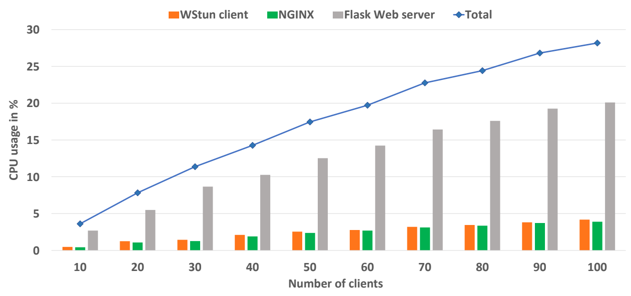


Fig. 9: CPU usage on the Raspberry Pi.

to the TCP windowing). Indeed, while the WS tunnel is being created, the UDP client keeps sending the packets at a fixed generation rate (i.e., 100 pps); those packets got queued and, therefore, delayed. For the packets generated at 200 pps (Fig. 7b), similarly, the first train of packets was affected by a higher latency, whereas a latency of approximately 50 ms identifies the steady-state response. In this case, we notice that a higher number of packets was affected by the WS tunnel setup (i.e., TCP handshake), as the packets' generation rate is higher than in the first case; thus, more packets were delayed in the queue. Albeit higher rate, the latency always stabilizes around the same value of 50 ms (i.e., the latency we fixed between the two VMs). It is worth mentioning that even though we depict only the first 300 packets in the graphs, we run each experiment for 10 minutes. For all cases, the latency remained constant and close to the values being shown. Regarding the performance of the tunnel vis-à-vis packets' sizes, we used larger packets' payloads. Specifically, as reported in Figs. 8a and 8b, we used packets with a payload 10 times larger than the first case (i.e, 400 bytes long) and we got the same results.

To overwhelm the tunnel's instantiation queuing issue at the beginning of the communications, a solution that would be conceivable when exposing services is to instantiate the WS tunnels more proactively (i.e., once the user expose the service and before receiving any request), anticipating the reception of the messages and keeping TCP sessions alive to avoid the three-way handshakes.

Another aspect we evaluated is the CPU usage since the WS tunneling mechanism we are introducing is meant to be implemented on IoT nodes. We report in Table 1 the CPU usage of the WS tunnel client. As we can notice, the packet size does not impact the CPU resource usage as the results when using packets with 40 and 400 bytes payload lengths are aligned. We mention that the CPU usage of `Socat` is reported as well since we are tunneling, in this case, a UDP traffic. The case when using TCP-based flows `Socat` is not required (see Section 4).

We conducted a set of other experiments to evaluate the performance of the WS tunnel and NGINX reverse proxy hosted on the IoT node. In particular, we measured the NGINX reverse proxy and the WS tunnel client CPU and RAM usage using a Raspberry Pi 3 Model B+ (Quad-Core 1.2GHz Broadcom BCM2837 64bit CPU with 1 GB of RAM). To generate (GET) HTTP requests, we simulated users by means of the open-source load testing tool, Locust[5]. We configured each (simulated) user to send one GET request per second (using the *constant_pacing* function). We report in Fig. 9 the CPU usage of the NGINX reverse proxy and the WS tunnel client. The CPU usage of the built-in Flask Web server used is reported as well (we opted for the Flask built-in Web server for a matter of implementation simplicity). Of course, other Web servers (e.g., Apache) can be used to improve the server performance. As shown in the figure, when generating ten requests per second by ten users, the WS tunnel client and NGINX uses 0.91/%. This value keeps increasing quite linearly to reach 8.07% when reaching 100 users. Regarding the RAM usage, the value remained constant all along with the experiments while varying the number of users. Practically, the NGINX reverse proxy and WS tunnel client combined use exactly 6.6% of the total amount of 2 GB of RAM available.

7 Conclusions and Future Work

In this paper, we introduced our tunneling Cloud based-approach with a dynamic DNS mechanism for exposing services hosted on IoT nodes deployed at the network edge. In particular, the solution deals with the typical constraints of IoT environments. We also showed the feasibility of the approach by providing an online accessible testbed. As further extensions of this work, we would like to implement a UDP-based tunneling system instead of the TCP-based one. Furthermore, in the actual solution, it is mandatory to go through the Cloud to route clients' requests. In this context, as future work, we would like to implement a Fog-based system with a distributed DNS mechanism to route requests based on users' localization.

References

[1] M. Satyanarayanan, "Pervasive computing: vision and challenges," *IEEE Personal Communications*, vol. 8, no. 4, pp. 10–17, 2001.
[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] L. T. Yang, B. Di Martino, and Q. Zhang, "Internet of everything," *Mobile Information Systems*, vol. 2017, 2017, cited By :21. [Online]. Available: www.scopus.com

5. https://locust.io

[4] R. Baheti and H. Gill, "Cyber-physical systems," *The impact of control technology*, vol. 12, no. 1, pp. 161–166, 2011.
[5] F.-Y. Wang, "The emergence of intelligent enterprises: From cps to cpss," *IEEE Intelligent Systems*, vol. 25, no. 4, pp. 85–88, 2010.
[6] A. Bröring, S. Schmid, C. Schindhelm, A. Khelil, S. Käbisch, D. Kramer, D. Le Phuoc, J. Mitic, D. Anicic, and E. Teniente, "Enabling iot ecosystems through platform interoperability," *IEEE Software*, vol. 34, no. 1, pp. 54–61, 2017.
[7] B. Di Martino, M. Rak, M. Ficco, A. Esposito, S. A. Maisto, and S. Nacchia, "Internet of things reference architectures, security and interoperability: A survey," *Internet of Things*, vol. 1, pp. 99–112, 2018.
[8] B. Di Martino, G. Cretella, and A. Esposito, "Cloud services composition through cloud patterns: a semantic-based approach," *Soft Computing*, vol. 21, no. 16, pp. 4557–4570, 2017.
[9] D. Guinard, V. Trifa, F. Mattern, and E. Wilde, "From the internet of things to the web of things: Resource-oriented architecture and best practices," in *Architecting the Internet of things*. Springer, 2011, pp. 97–129.
[10] Z. Benomar, F. Longo, G. Merlino, and A. Puliafito, "A stack4things-based web of things architecture," in *2020 International Conferences on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData) and IEEE Congress on Cybermatics (Cybermatics)*. IEEE, 2020, pp. 113–120.
[11] C. Pautasso and E. Wilde, "Why is the web loosely coupled? a multi-faceted metric for service design," in *Proceedings of the 18th international conference on World wide web*, 2009, pp. 911–920.
[12] C. Prehofer and I. Gerostathopoulos, "Modeling restful web of things services: Concepts and tools," in *Managing the Web of Things*. Elsevier, 2017, pp. 73–104.
[13] Z. Shelby, "Embedded web services," *IEEE Wireless Communications*, vol. 17, no. 6, pp. 52–57, 2010.
[14] A.-R. Breje, R. Győrödi, C. Győrödi, D. Zmaranda, and G. Pecherle, "Comparative study of data sending methods for xml and json models," *INTERNATIONAL JOURNAL OF ADVANCED COMPUTER SCIENCE AND APPLICATIONS*, vol. 9, no. 12, pp. 198–204, 2018.
[15] D. Yazar and A. Dunkels, "Efficient application integration in ip-based sensor networks," in *Proceedings of the First ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*, 2009, pp. 43–48.
[16] N. Naik, "Choice of effective messaging protocols for iot systems: Mqtt, coap, amqp and http," in *2017 IEEE international systems engineering symposium (ISSE)*. IEEE, 2017, pp. 1–7.
[17] D. Guinard, I. Ion, and S. Mayer, "In search of an internet of things service architecture: Rest or ws-*? a developers' perspective," in *International Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services*. Springer, 2011, pp. 326–337.
[18] B. N. Silva, M. Khan, and K. Han, "Integration of big data analytics embedded smart city architecture with restful web of things for efficient service provision and energy management," *Future generation computer systems*, vol. 107, pp. 975–987, 2020.
[19] S. Jabbar, M. Khan, B. N. Silva, and K. Han, "A rest-based industrial web of things' framework for smart warehousing," *The Journal of Supercomputing*, vol. 74, no. 9, pp. 4419–4433, 2018.
[20] B. N. Silva, M. Khan, K. Lee, Y. Yoon, D. Muhammad, J. Han, and K. Han, "Restful web of things for ubiquitous smart home energy management," in *2020 International Conference on Computing, Networking and Communications (ICNC)*. IEEE, 2020, pp. 176–180.
[21] A. Tiberkak, A. Hentout, and A. Belkhir, "Lightweight remote control of distributed web-of-things platforms: First prototype," in *2020 IEEE International Conference on Internet of Things and Intelligence System (IoTaIS)*. IEEE, 2021, pp. 103–108.
[22] 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.
[23] R. Yugha and S. Chithra, "A survey on technologies and security protocols: Reference for future generation iot," *Journal of Network and Computer Applications*, p. 102763, 2020.
[24] A. P. Felt, R. Barnes, A. King, C. Palmer, C. Bentzel, and P. Tabriz, "Measuring {HTTPS} adoption on the web," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017, pp. 1323–1338.
[25] R. Barnes, J. Hoffman-Andrews, and J. Kasten, "Automatic certificate management environment (acme)," *Internet-Draft draft-ietf-acme-acme-09, IETF Secretariat*, 2017.
[26] E. F. Kfoury, D. Khoury, A. AlSabeh, J. Gomez, J. Crichigno, and E. Bou-Harb, "A blockchain-based method for decentralizing the acme protocol to enhance trust in pki," in *2020 43rd International Conference on Telecommunications and Signal Processing (TSP)*. IEEE, 2020, pp. 461–465.
[27] F. Longo, D. Bruneo, S. Distefano, G. Merlino, and A. Puliafito, "Stack4things: An openstack-based framework for iot," in *2015 3rd International Conference on Future Internet of Things and Cloud*, 2015, pp. 204–211.
[28] Z. Benomar, F. Longo, G. Merlino, and A. Puliafito, "Cloud-based enabling mechanisms for container deployment and migration at the network edge," *ACM Trans. Internet Technol.*, vol. 20, no. 3, Jun. 2020. [Online]. Available: https://doi.org/10.1145/3380955
[29] Z. Benomar, D. Bruneo, S. Distefano, K. Elbaamrani, N. Idboufker, F. Longo, G. Merlino, and A. Puliafito, "Extending openstack for cloud-based networking at the edge," in *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, 2018, pp. 162–169.
[30] D. Skvorc, M. Horvat, and S. Srbljic, "Performance evaluation of websocket protocol for implementation of full-duplex web streams," in *2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE, 2014, pp. 1003–1008.

Zakaria Benomar is currently a final-year Ph.D. student at the Department of Engineering, University of Messina, Italy. His main research interests include Cloud computing, Internet of Things, Network virtualization and Edge/Fog computing. He is the recipient of the Outstanding Paper Award during the IEEE international conference on Internet of Things (iThings-2020).

Francesco Longo is a tenure-track Assistant Professor of computer engineering at the University of Messina, Italy. His main research interests include performance and dependability evaluation of distributed systems, IoT, Cloud, and Fog/Edge computing with applications in Smart Cities and Industry 4.0, Blockchain and DLT technologies and their use in building trustless systems.

Giovanni Merlino is tenure-track Assistant Professor of computer engineering at University of Messina, Italy. His main research interests revolve around mobile and distributed systems with particular emphasis on the Internet of Things for the Industry 4.0, Fog Computing and the Web of Smart Things, in support of Deep Learning-based biomedical applications, cyber-physical systems as software-defined Smart City infrastructure, and mobile crowdsensing

Antonio Puliafito is Professor of computer engineering at the University of Messina, Italy. His interests include distributed systems, wireless and Cloud computing. He is acting as an expert in ICT for the European Commission since 1998. He is currently the Director of the national CINI Lab on Smart cities and Communities. He participated in several European projects such as Reservoir, Vision,CloudWave and Beacon. He is the scientific director of the SmartMe.io spinoff company.