Stack4Things: an OpenStack-based framework for IoT

Francesco Longo*, Dario Bruneo*, Salvatore Distefano†§ Giovanni Merlino*‡, Antonio Puliafito*

* Università di Messina, Italy
Email: {flongo,dbruneo,gmerlino,apuliafito}@unime.it

† Politecnico di Milano, Italy
Email: salvatore.distefano@polimi.it

§ Kazan Federal University, Russia
Email: s_distefano@it.kfu.ru

‡ Università di Catania, Italy
Email: giovanni.merlino@dieei.unict.it

Abstract

In the wake of the massive adoption of embedded systems, mobiles, and other smart devices, as the scope of their involvement keeps broadening, complexity may quickly become overwhelming and vertical ad-hoc solutions will not cut it anymore. We propose to reuse as much tooling as possible, taking into account suitable options with regard to infrastructure management, then piggybacking as much advanced functionalities as possible in such kind of environment. In this sense, a widely used and competitive framework for Infrastructure-as-a-Service such as OpenStack, with its breadth in terms of feature coverage and expanded scope, looks like fitting the bill. This work therefore describes the approach and the solutions so far preliminary implemented for enabling Cloud-mediated interactions with droves of sensor- and actuator-hosting nodes by proposing Stack4Things, a framework for Sensing-and-Actuation-as-a-Service. In particular, we focused on describing the subsystem of Stack4Things devoted to resource control and management, highlighting relevant requirements and justifying how our proposed framework addresses them, while also opening up possibilities for a range of future extensions towards complete fulfillment of the Sensing-and-Actuation-as-a-Service vision.

Keywords

IoT, Cloud, OpenStack, WebSocket, WAMP, SAaaS

1. Introduction

Internet of Things (IoT) is the one of the hottest trend in Information and Communication Technology (ICT), being characterized by the presence of field-deployed, dispersed, sensor- and actuator-hosting platforms, usually available in several orders of magnitude bigger quantities, possibly heterogeneous along several axes (e.g., instruction set architecture, operating system, userspace, runtime). Nowadays, embedded systems are getting ever more powerful and flexible in terms of reprogrammable behavior and ease of use, gaining a "smart" labeling to indicate this evolution. This all-encompassing and much ambitious scenario calls for adequate technologies.

A number of solutions in the literature, mainly act at a lower (communication) layer, aiming at interconnecting network-enabled devices and, generally, any *thing* featuring a network interface, to the Internet and, thus, among themselves [1]. However, according to the vision of the Sensing-and-Actuation-as-a-Service paradigm [2], solutions for creating and managing a dynamic infrastructure of sensing and actuation resources are also required. In fact, in order to effectively control devices, sensors, and things, management, organization, and coordination mechanisms are strongly needed: a middleware devoted to management of both sensor- and actuator-hosting resources may help in the establishment of higher-level services. In this direction, most efforts revolve around managing heterogeneous devices by resorting to legacy protocols and vertical solutions out of necessity, and integrating the whole ecosystem by means of ad-hoc approaches [3].

In our vision, the Cloud may play a role both as a paradigm, and as one or more ready-made solutions for a (virtual) infrastructure manager (VIM), to be extended to IoT infrastructure. In particular, we propose to extend a well known framework for the management of Cloud computing resources, OpenStack [4], to manage sensing and actuation ones, implementing in our Stack4Things[1] solution an infrastructure-oriented approach, while coping with communication requirements and scalability concerns by leveraging Cloud-focused design choices and architectural patterns.

Stack4Things [5] is an OpenStack-based framework implementing the Sensing-and-Actuation-as-a-Service paradigm [2]. In [5], we described the architecture of the framework by focusing on data management and visualization aspects, also presenting some implementation details. In this paper, we mainly deal with the remote Cloud-based control and management of sensing and actuation resources, showing the portion of our reference architecture related to such functionalities and providing details about the exploited technologies. In designing Stack4Things, we are trying to follow a bottom-up approach, consisting of a mixture of relevant technologies, frameworks, and protocols. In addition to the already cited

1. From now on, Stack4Things is sometimes abbreviated as s4t.

978-1-4673-8103-1/15 $31.00 © 2015 IEEE
DOI 10.1109/FiCloud.2015.97

OpenStack, we take advantage of the WebSocket technology [6] and we base our communication framework on the Web Application Messaging Protocol (WAMP) [7]. Moreover, we are absolutely keen on openness, and thus we are developing all our software as Open Source, making it freely available through the Web [8].

As already mentioned above, there is much low hanging fruit to be reaped by following such an infrastructure-oriented approach, but in particular this paper focuses on the unique possibilities unlocked by leveraging the Cloud in order to get instant access to ubiquitous embedded devices regardless the connectivity options, as we'll see more in detail later.

The convergence of IoT and Cloud approaches and technologies is attracting growing interests in the research community. As an example, in [9] an architecture for the Web of Things, where the Cloud plays the role of a backbone for composition of services, leveraging *Node.js* for business code to be run on the Cloud have been investigated. A Machine-to-Machine network powering a Cloud-enhanced testbed, where OpenStack is exploited to host an IoT "portal" is also proposed in [10]. An interesting project about Cloud and IoT, ClouT [11], proposes the usage of an OpenSource framework for IaaS, in this case oVirt, in place of OpenStack, in the context of IoT-based scenarios, but only in terms of storage backend for data collection duties. While in [12] a framework for context-aware mobile applications leveraging IoT technologies, OpenStack and CloudFoundry, is implemented as an application-level development, where the focus lies in the composition of services.

To the best of our knowledge there are no effective research or implementation efforts towards the SAaaS vision. In particular we focus on the high-level management of the IoT infrastructure adopting a Cloud-/service-oriented approach, with specific regard to the IoT resource management and the interactions among such resources or nodes and the infrastructure manager. To address the communication issues and centralized management requirements we propose to adopt a WAMP-based protocol (or even WebSocket-based for that matter) on top of an (OpenStack-powered) IaaS subsystem for a variety of remote management interactions with field-deployed boards.

The rest of the paper is organized as follows. Section 2 describes the SAaaS scenario under consideration. Then, in Section 3, a reference architecture addressing the main requirements in this scenario is identified. Its specialization into the (Arduino) board domain through the Stack4Things framework is described in Section 4 also providing some implementation details. A simple use case of the framework thus implemented is reported in Section 5, while Section 6 discusses on how the framework can be extended to overcome the limitations and include additional features and functionalities. At last, Section 7 ends the paper with some closing considerations.

2. Sensing-and-Actuation-as-a-Service

Figure 1 represents the scenario that we take into consideration. In this scenario, Cloud computing facilities, implementing a service-oriented approach in the provisioning and management of sensing and actuation resources, are exploited to create a SAaaS Cloud. In fact, in the SAaaS vision, sensing and actuation devices should be handled along the same lines as computing and storage abstractions in traditional Clouds, i.e., virtualized and multiplexed over (possibly scarce) hardware resources. Thus, sensing and actuation devices have to be part of the Cloud infrastructure and have to be managed by following the consolidated Cloud approach, i.e., through a set of APIs ensuring remote control of software and hardware resources despite their geographical position. In other words, the idea is quite appealing and challenging since in such scenario an user could ask for handles on (physical world) items to be manipulated through the user interfaces of the Cloud framework. Services related to the (sensing and actuation) infrastructure provisioning should be provided on-demand in an elastic and QoS-guaranteed way. In this way, on top of such services, other application level services can be easily implemented and provided to final users.

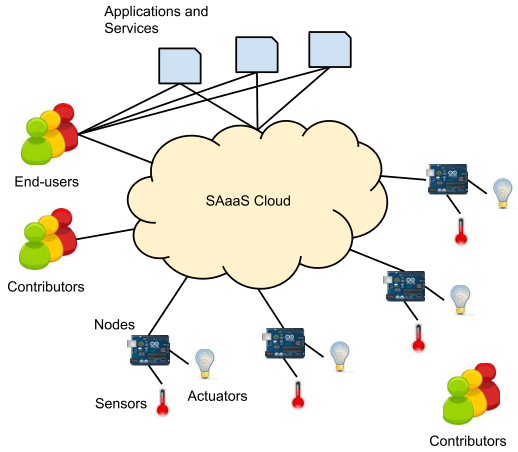

Figure 1. The reference scenario.

The main actors in the scenario are *contributors* and *end users*. Contributors provide sensing and actuation resources building up the SAaaS infrastructure. Examples of contributors are sensor networks owners, device owners, and people offering their PDAs as a source of data in a crowdsourcing model [13]. End users control and manage the resources provided by contributors. In particular, end-users may behave as infrastructure administrators and/or service providers, managing the SAaaS infrastructure and implementing applications and services on top of it. We assume that the sensing and actuation resources are provided to the infrastructure via a number of hardware-constrained units, which we refer to as *nodes*. Nodes host sensing and actuation resources and act as mediators in relation to the Cloud infrastructure. They need to

have connectivity to the Internet in order for our approach to be applicable.

3. The proposed solution

In order to implement the above scenario, a specific solution for addressing the SAaaS issues and challenges has to be provided. Once identified the specific requirements, in the following we propose an high-level reference architecture highlighting the main modules and issues to be addressed.

3.1. Requirements

The scenario so far outlined presents several critical aspects that have to be suitably taken into account through an accurate requirements analysis. Requirements may be distinguished into functional and non-functional requirements. Functional requirements can in turn be defined with respect to the contributor and end-user points of view.

3.1.1. Functional requirements. Main requirements from the **contributor** point of view are:

- **plug-and-play experience:** letting nodes and the corresponding sensors and actuators be enrolled automatically in the Cloud, e.g., at unpacking time;
- **uniform interaction model:** resources should be hooked up (or unenrolled, when preferred) with the minimum amount of involvement for the contributor to feed the enrollment process with details about their hardware characteristics;
- **contribution profile:** each contributor should be able to specify her profile for contribution in terms of resource utilization (CPU utilization, memory or disk space) and contribution period (frame time when the contributor is available for contribution).

Main requirements from the **end-user** point of view are:

- **status tracking:** monitoring the status (presence, connectivity, usage, etc.) of nodes and corresponding resources, in order to, e.g., track significant outages or load profiles;
- **lifecycle management:** exposing a set of available management primitives for sensing and actuation resources, in order to, e.g., change sampling parameters when needed or, e.g., reap a pending actuation task to free the resource for another higher-priority duty;
- **service-oriented interfaces:** exposing primitives as asynchronous service endpoint to ease development and third-party software integration;
- **ubiquitous access:** enabled through instant-on bidirectional communication with resources as exposed from sensor-hosting nodes, whichever the constraints imposed by node-side network topology, e.g., Network Address Translation (NAT), and configuration (e.g., firewall);
- **delegation capabilities:** providing client-less (Cloud-enabled) interactions, by switching to alternative Cloud-hosted controlling surfaces (e.g., Web-based graphical or textual terminal) as needed, e.g., clients may need to disconnect at any time;
- **ensemble management:** letting nodes and the corresponding sensors and actuators be made available as pools of resources, e.g., to be partitioned in, and allocated as, groups according to requirements;
- **uniform information model:** resources (e.g., down to single I/O pins) should be indexed according to a suitable model and searchable through standardized query syntax and predefined rules;
- **instance provisioning:** resources should be made available (provisioned) subject to certain user-mandated constraints (geographical context, etc.), decoupling specific instances from the function they embody;
- **environment customization:** enabling runtime modifications to the node software environment, falling under the following categories
  - logic injection: deployment of any new (or modified) application-level object code;
  - OS-level reconfiguration: pushing any system-level userspace (or even kernelspace) binaries, possibly interacting with a package and/or a service supervisor (init);
- **topology rewiring:** providing mechanisms for the networking configuration underneath nodes to be modified at any time;
- **orchestration:** exposing interfaces for the orchestration (e.g., dependency-based startup, endpoint wiring, etc.) of ensembles of resources.

3.1.2. Non-Functional requirements. Main non-functional requirements in this scenario are:

- **scalability:** each contributor should be able to provide a great number of sensing and actuation resources and, similarly, end-users should be able to interact with them in big quantities;
- **QoS:** QoS level should be maintained as high as possible while interacting with nodes through the SAaaS Cloud, e.g., latency experienced during communication with nodes should be kept low;
- **fault tolerance and reliability:** the whole infrastructure should be resilient to faults on both the Cloud and the node sides;
- **security:** authentication, authorization, and security in both the control communication and data should be guaranteed.

3.2. Reference architecture

This way, an high level architecture addressing the above requirements is shown in Figure 2. It is a logical architecture mainly grouping basic functionalities and mechanisms to be provided by the SAaaS Cloud. At the bottom of the two-layer schema there is the node level, including both devices (mobiles, tablets, smart objects) and boards. On top of them

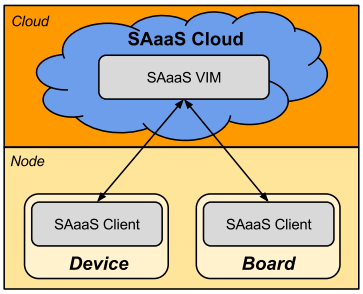

Figure 2. SAaaS reference architecture.

the Cloud level enrols the sensing resources into a sensing Cloud infrastructure to be provided as a service, elastically, on-demand, according to the above specified non-functional requirements, i.e. the SAaaS.

The reference architecture therefore implements a kind of client server model, where the SAaaS VIM manages the nodes by interacting with them through an SAaaS Client. This highlights that the two main functionalities to be provided by the SAaaS system are the management of sensing resources and the communication of a node SAaaS Client with the VIM. In the following we will provide details on how to address these specific issues on sensing boards.

4. Stack4Things

While designing our solution, we restricted the SAaaS scope to just IoT sensing board, mainly focusing on the high, Cloud provisioning model. As a reference with respect to the IoT nodes to be connected to the SAaaS infrastructure, we take into account the latest *Arduino YUN* [14]-like boards. Such kind of devices is usually equipped with (low power) micro-controller (MCU) and micro-processor (MPU) units. Typically, they can interact with the physical world by means of a set of sensing and actuation resources connected to their digital/analog I/O pins. Moreover, they can connect to the Internet thanks to Ethernet and WiFi network interfaces. Such kind of smart boards marry the ease of programming the MCU (typically with simple languages derived from C/C++) with the power of a Linux distribution running onboard on the MPU. This allows not only to sense the physical world and actuate on it, but also to partially elaborate information on-site taking decisions and resorting to external communication only if necessary.

Figure 3 shows the Stack4Things overall architecture, focusing on communication between end users and sensor- and actuator-hosting nodes. On the board side, the *Stack4Things lightning-rod*, acting as SAaaS Client, runs on the MPU and interacts with the OS tools and services of the board, and with sensing and actuation resources through I/O pins. It represents the point of contact with the Cloud infrastructure allowing the end users to manage the board resources even if they are behind a NAT or a strict firewall. This is ensured by a WAMP and WebSocket-based communication between the Stack4Things lightning-rod and its Cloud counterpart, namely the *Stack4Things IoTronic service*. The Stack4Things IoTronic service, acting as SAaaS VIM, is implemented as an OpenStack service providing end users with the possibility to manage one or more smart boards, remotely. This can happen both via a command-line based client, namely *Stack4Things command line client*, and a Web browser though a set of REST APIs provided by the Stack4Things IoTronic service.

4.1. Board-side

Several technologies exist allowing the MCU and the MPU to communicate between each other, usually resorting to serial communications. Following recent developments for YUN-like devices, *BaTHOS* [15] on the MCU side and a set of Linux kernel modules on the MPU side has been adopted, thus enabling the digital/analog I/O pins to be directly accessed from the Linux distribution.

With respect to network connectivity and presence/reachability, *WebSocket* [6] is the leading technology. WebSocket is a standard HTTP-based protocol providing a full-duplex TCP communication channel over a single HTTP-based persistent connection. The protocol is designed to be easily implemented in Web browsers and servers and it is compliant with HTTP because its handshake is interpreted by HTTP servers as an Upgrade request. The main reason leading to the design of WebSocket is the necessity to go beyond the long-polling and Asynchronous JavaScript and XML (AJAX) approaches. In fact, among use cases, WebSocket allows the server to send content to the browser without being solicited. Messages can thus be passed back and forth while keeping the connection open creating a two-way (bi-directional) ongoing conversation between a browser and the server.

However, one of the main advantages of WebSocket is that it is network agnostic. In fact, the communications are performed over TCP port number 80. This is of benefit for those environments which block non-Web Internet connections using a firewall. For this reason, WebSocket immediately started to be of interest also for communications that are not based on Web browsers and servers and several application-level protocols started to rely on this Web-based transport protocol for communication (see for example the use of eXtensible Messaging and Presence Protocol (XMPP) over WebSocket), also in the IoT field.

Web Application Messaging Protocol (WAMP) [7] is a sub-protocol of WebSocket, specifying a communication semantic for messages sent over WebSocket. Differently from other application-level messaging protocols, e.g., XMPP, Advanced Message Queuing Protocol (AMQP), ZeroMQ, WAMP is natively based on WebSocket (even if it also allows for different transport protocols) and provides both publish/subscribe (pub/sub) and remote procedure call (RPC) mechanisms. In particular, WAMP implements routed RPCs. Indeed, in WAMP RPC implementation, the *caller* and the *callee* are completely decoupled thanks to the presence of a *dealer*. The dealer

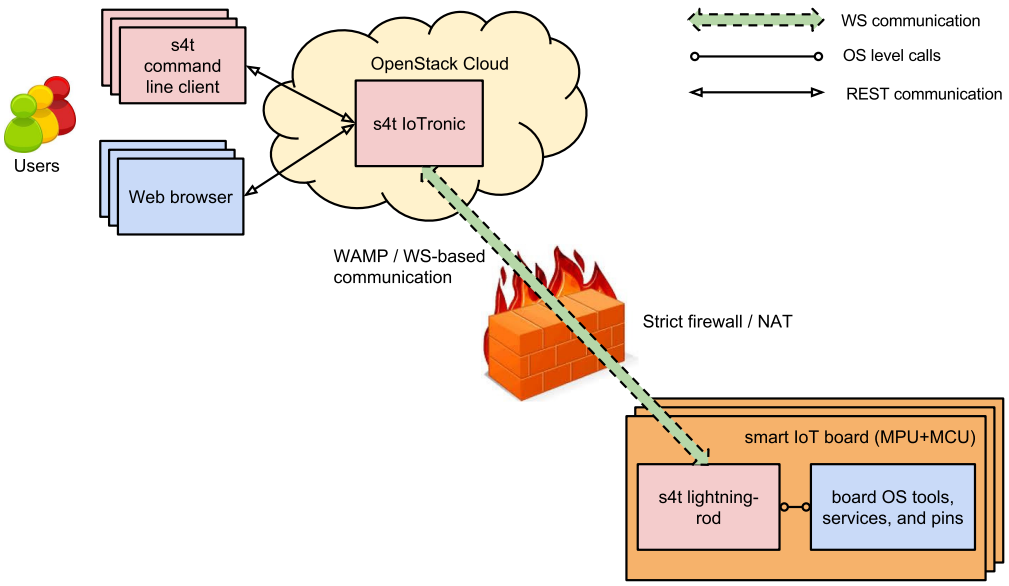

Figure 3. Stack4Things overall architecture for Arduino YUN-like boards.

is usually implemented together with a pub/sub *broker* in the WAMP *router*. The WAMP router is then responsible of brokering pub/sub messages and routing remote calls, together with results/errors.

Figure 4 shows the Stack4Things architecture with more focus on the board side. We assume that *BaTHOS* runs on the board MCU while a Linux OpenWRT-like distribution runs on the MPU. BaTHOS is equipped with a set of extensions (from now on indicated as MCUIO extensions) that expose the board digital/analog I/O pins to the Linux kernel. The communication is carried out over a serial bus. The Linux kernel running on the MPU is compiled with built-in host-side *MCUIO modules*. In particular, functionalities provided by the MCUIO kernel modules include enumeration of the pins and exporting corresponding handlers for I/O in the form of i-nodes of the Linux sysfs virtual filesystem. Upwards the sysfs abstraction, which is compliant with common assumptions on UNIX-like filesystems, there is the need to mediate access by means of a set of MCUIO-inspired libraries, namely *Stack4Things MCUIO sysfs libraries*. Such libraries represent the interface with respect to the MCUIO sysfs filesystem dealing with read and write requests in terms of concurrency. This is done at the right level of semantic abstraction, i.e., locking and releasing resources according to bookings and in a way that is dependent upon requirements deriving from the typical behavior of general purpose I/O pins and other requirements that are specific to the sensing and actuating resources.

The *Stack4Things lightning-rod engine* represents the core of the board-side software architecture. The engine interacts with the Cloud by connecting to a specific WAMP router (see also Figure 5) through a WebSocket full-duplex channel, sending and receiving data to/from the Cloud and executing commands provided by the users via the Cloud. Such commands can be related to the communication with the board digital/analog I/O pins and thus with the connected sensing and actuation resources (through the Stack4Things MCUIO sysfs library) and to the interactions with OS tools and/or resources (e.g., filesystem, services and daemons, package manager). The communication with the Cloud is assured by a set of libraries implementing the client-side functionalities of the WAMP protocol (*Stack4Things WAMP libraries*). Moreover, a set of WebSocket libraries (*Stack4Things wstunnel libraries*) allows the engine to act as a WebSocket reverse tunneling server, connecting to a specific WebSocket server running in the Cloud. This allows internal services to be directly accessed by external users through the WebSocket tunnel whose incoming traffic is automatically forwarded to the internal daemon, e.g., Secure Shell (SSH), Hypertext Transfer Protocol (HTTP), Telnet, under consideration. Outgoing traffic is redirected to the WebSocket tunnel and eventually reaches the end user that connects to the WebSocket server running in the Cloud in order to interact with the board service.

The Stack4Things lightning-rod engine also implements a plugin loader. Custom plugins can be injected from the Cloud and run on top of the plugin loader in order to implement specific user-defined commands, possibly including system-level interactions, such as, e.g., with a package manager and/or the init/runlevels subsystem.

In this sense the authors may resort, in future efforts, to previous work[16] of their own related to runtime customization for further enhancements to the architecture.

New REST resources are automatically created exposing the user-defined commands on the Cloud side. As soon as such

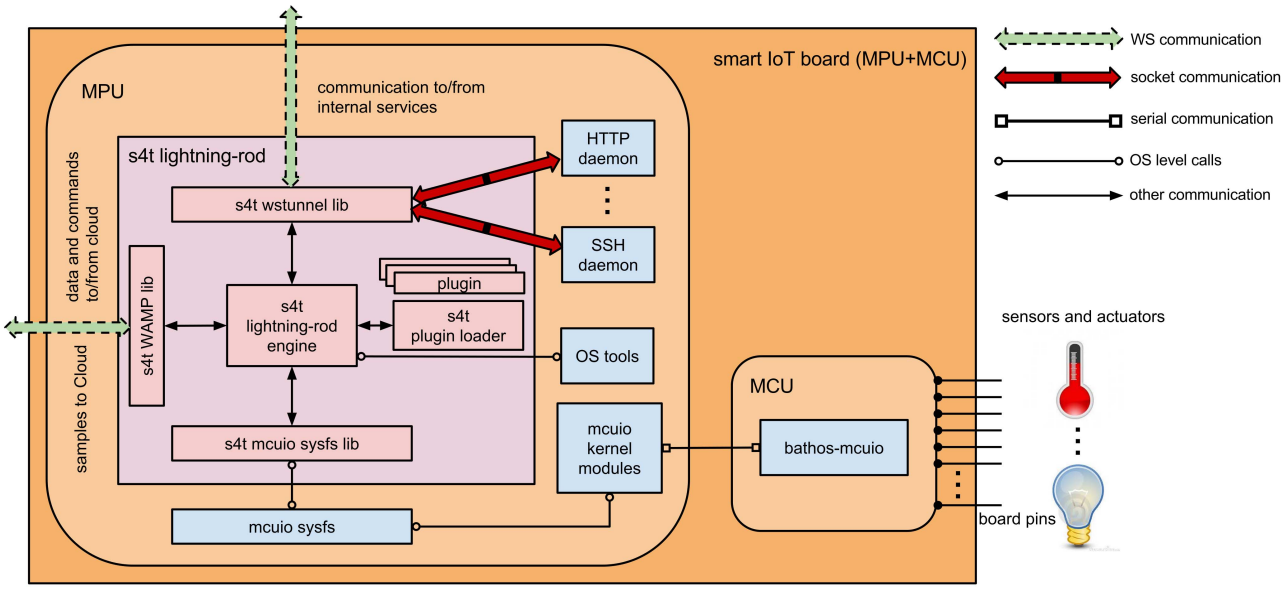

Figure 4. Stack4Things board-side architecture.

resources are invoked the corresponding code is executed on top of the smart board.

4.2. Cloud-side - control and actuation

As already mentioned, with respect to the virtual infrastructure manager, OpenStack [4] is the technology of reference. OpenStack is a centerpiece of infrastructure Cloud solutions for most commercial, in-house, and hybrid deployments, as well as a fully Open Source ecosystem of tools and frameworks upon which many EU projects, such as CloudWave (FP-7) [17], are founding their Cloud strategies. Moreover, with an independent consortium featuring a diverse partnership, including world-class industrial players and smaller enterprises, and a global community of thousands developers, OpenStack has the clout to withstand most shifts in paradigms and approaches, and holds the promise to keep its relevance in time within the increasingly crowded Infrastructure-as-a-Service space. Currently OpenStack allows to manage virtualized computing/storage resources, according to the infrastructure Cloud paradigm. Our main goal in this paper is to propose an extension of OpenStack for the management of sensing and actuation resources.

The Stack4Things Cloud-side architecture (see Figure 5) consists of an OpenStack service we called IoTronic. The main goals of IoTronic lie in extending the OpenStack architecture towards the management of sensing and actuation resources, i.e., to be an implementation of the SAaaS paradigm. IoTronic is characterized by the standard architecture of an OpenStack service. The *Stack4Things IoTronic conductor* represents the core of the service, managing the *Stack4Things IoTronic database* that stores all the necessary information, e.g., board-unique identifiers, association with users and tenants, board properties and hardware/software characteristics as well as dispatching remote procedure calls among other components. The *Stack4Things IoTronic APIs* exposes a REST interface for the end users that may interact with the service both via a custom client (*Stack4Things IoTronic command line client*) and via a Web browser. In fact, the OpenStack Horizon dashboard has been enhanced with a *Stack4Things dashboard* exposing all the functionalities provided by the Stack4Things IoTronic service and other software components. In particular, the dashboard also deals with the access to board-internal services, redirecting the user to the *Stack4Things IoTronic WS tunnel agent*. This piece of software is a wrapper and a controller for the WebSocket server to which the boards connect through the use of Stack4Things wstunnel libraries.

Similarly, the *Stack4Things IoTronic WAMP agent* controls the WAMP router and acts as a bridge between other components and the boards. It translates Advanced Message Queuing Protocol (AMQP) messages into WAMP messages and vice-versa. AMQP is an open standard application layer protocol for message-oriented middleware, a bus featuring message orientation, queueing, routing (including point-to-point and publish-subscribe), reliability and security. Following the standard OpenStack philosophy all the communication among the IoTronic components is performed over the network via an AMQP queue. This allows the whole architecture to be as scalable as possible given that all the components can be deployed on different machines without affecting the service functionalities, as well as the fact that more than one *Stack4Things IoTronic WS tunnel agent* and more than one *Stack4Things IoTronic WAMP agent* can be instantiated, each of them dealing with a sub-set of the IoT devices. In this way, redundancy and high availability are also guaranteed. As

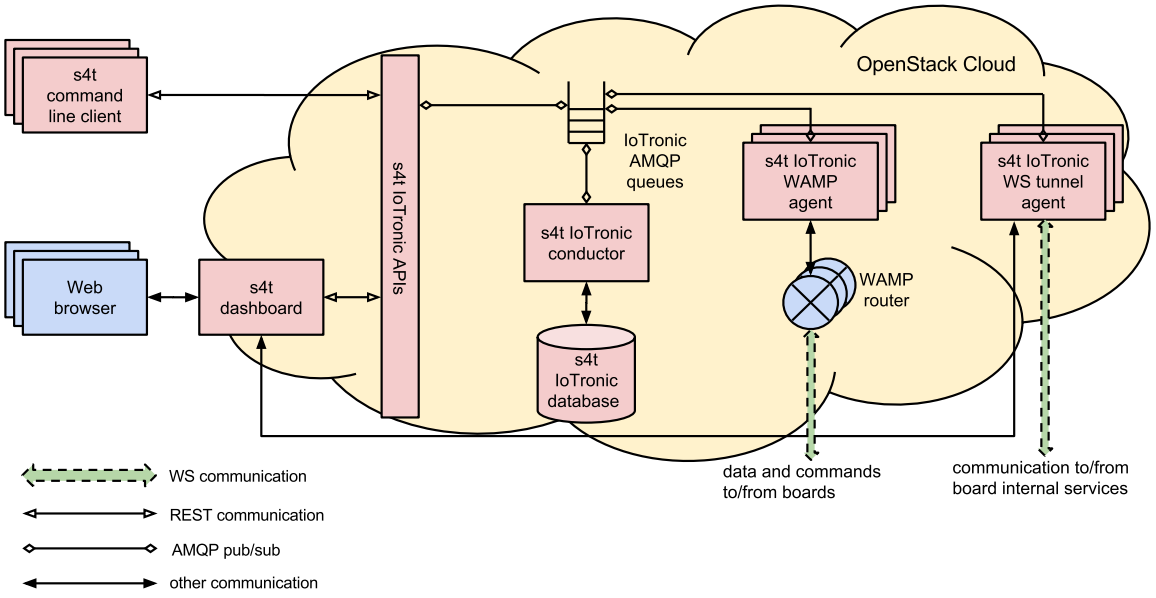

Figure 5. Stack4Things Cloud-side architecture.

mentioned above, a crucial reason for choosing WAMP as the protocol for node-related interactions, apart from possibly leaner implementations and smoother porting, lies in WAMP being a WebSocket subprotocol and supporting two application messaging patterns, Publish & Subscribe and Remote Procedure Calls, the latter being not available in AMQP. A prototype of the architecture so far described has been implemented and source code is freely available through the Web [8].

5. Use case: SSH connection toward a node

Figure 6 depicts an example use case, showing how the architecture components interact among themselves. The use case that we take into consideration is a common one in classical Cloud scenarios, i.e., SSH access into a virtualized resource. In our case, we consider the creation of an SSH connection toward a node through the help of the Cloud management system. Given the assumption that nodes are behind a firewall/NAT a complex interaction flow is needed in order to fulfill the goal.

As a use case prerequisite, we assume the node to be reached is already registered to the Cloud. The following operations are then performed: i) the user asks for a connection to the SSH daemon running on a specific board through the s4t dashboard (or alternatively through the s4t command line client); ii) the s4t dashboard performs one of the available s4t IoTronic APIs calls via REST. The call pushes a new message into a specific AMQP IoTronic queue; iii) the s4t IoTronic conductor pulls the message from the AMQP IoTronic queue and it performs a query to the s4t IoTronic database. In particular, it checks if the board is already registered to the Cloud and queries for the s4t IoTronic WAMP agent to which

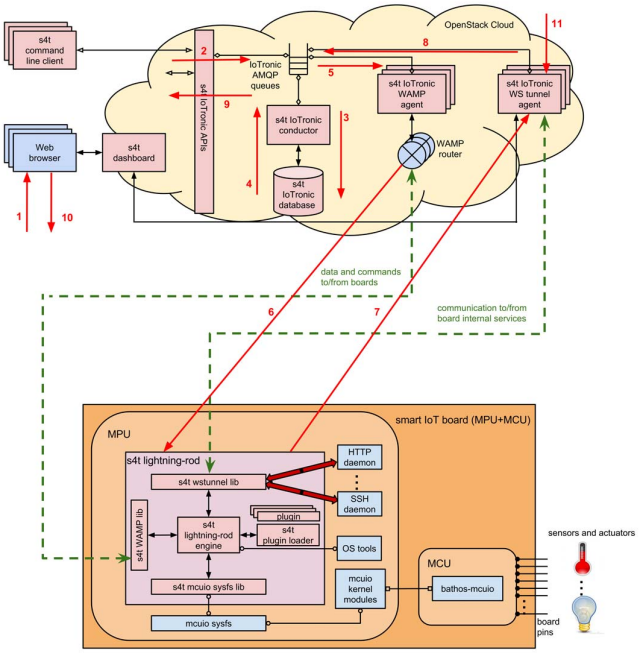

Figure 6. Creation of an SSH connection toward a node.

the board is registered. Finally, it decides the s4t IoTronic WS tunnel agent to which the user can be redirected and randomly generates a free TCP port; iv) the s4t IoTronic conductor pushes a new message into a specific AMQP IoTronic queue; v) the s4t IoTronic WAMP agent to which the board is registered pulls the message from the queue and publishes a new message into a specific topic on the corresponding WAMP

router; vi) through the s4t WAMP lib the s4t lightning-rod engine receives the message by the WAMP router; vii) the s4t lightning-rod engine opens a reverse WebSocket tunnel to the s4t IoTronic WS tunnel agent specified by the s4t IoTronic conductor also providing the TCP port through the s4t wstunnel lib. It also opens a TCP connection to the internal SSH daemon and pipes the socket to the tunnel; viii) the s4t IoTronic WS tunnel agent opens a TCP server on the specified port. Then, it publishes a new message into a specific AMQP IoTronic queue confirming that the operation has been correctly executed; ix) the s4t IoTronic APIs call pulls the message from the AMQP IoTronic queue and replies to the s4t dashboard; x) the s4t dashboard provides the user with the IP address and TCP port that he/she can use to connect to reach the SSH service on the board; xii) the user connects to the specified IP address and TCP port via an SSH client and the connection is tunneled to the board.

6. Future work

Future work will tackle secure communications by upgrading the WebSocket-based communications to Secure WebSockets (was). It is also of interest to extend and/integrate other OpenStack services with SAaaS functionalities. In particular, as shown in [5], the Ceilometer service, whose aim lies in providing a monitoring framework for the Cloud infrastructure, could be extended and integrated with the IoTronic service to collect metrics generated by sensing resources, also providing advanced functionalities such as complex event processing. The Neutron service may also be extended, in order to manage creation, configuration, and destruction of virtual networks among nodes at data-link or network layer, thus enabling more interesting use cases such as the porting of legacy IoT solutions that rely on a single broadcast domain such as AllJoyn [18].[2] The latter may be considered an enablement step for the extension of orchestration mechanisms (the domain of the Heat subsystem in OpenStack) to Cloud-controlled boards. Finally, authentication, authorization, and tenant management could be provided by integrating the IoTronic service with the OpenStack Keystone service enabling the implementation of ad-hoc security mechanisms for the control communication and data exchange among the Cloud infrastructure and the sensor- and actuator-hosting nodes.

7. Conclusions

The paper described the implementation of the SAaaS vision in the OpenStack framework by proposing Stack4Things. The considered scenario with actors and entities have been described and the main requirements have been enumerated and discussed with particular reference to the sensor- and actuator-hosting node control and management. The related Stack4Things architecture have been illustrated together with enabling technologies. Finally, a discussion about the architecture and about possible future research directions have been provided.

2. University of Messina has recently been admitted as a Sponsored Member of the AllSeen Alliance, https://allseenalliance.org/about/members, the foundation acting as steward of the AllJoyn family of standards.

Acknowledgment

This work is partially funded by the EU 7th Framework Programme under grant Agreement 610802 for the "CloudWave" project, by a PhD programme under grant PON R&C 2007/2013 "Smart Cities" and by the SIGMA Project - Italian National Operative Program (PON01_00683) 2007-2013.

References

[1] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of things (iot): A vision, architectural elements, and future directions," *Future Generation Computer Systems*, vol. 29, no. 7, pp. 1645–1660, 2013.
[2] S. Distefano, G. Merlino, and A. Puliafito, "Sensing and actuation as a service: A new development for clouds," in *Network Computing and Applications (NCA), 2012 11th IEEE International Symposium on*, Aug 2012, pp. 272–275.
[3] L. Sanchez, J. Galache, V. Gutierrez, J. Hernandez, J. Bernat, A. Gluhak, and T. Garcia, "Smartsantander: The meeting point between future internet research and experimentation and the smart cities," in *Future Network Mobile Summit (FutureNetw), 2011*, June 2011, pp. 1–8.
[4] "OpenStack documentation [URL]," http://docs.openstack.org.
[5] G. Merlino, D. Bruneo, S. Distefano, F. Longo, and A. Puliafito, "Stack4things: Integrating IoT with OpenStack in a Smart City context," in *Proceedings of the IEEE First International Workshop on Sensors and Smart Cities*, 2014.
[6] "WebSocket [URL]," https://tools.ietf.org/html/rfc6455.
[7] "WAMP [URL]," http://wamp.ws.
[8] "Stack4Things source code [URL]," https://github.com/MDSLab.
[9] T.-M. Grønli, G. Ghinea, and M. Younas, "A lightweight architecture for the web-of-things," in *Mobile Web Information Systems*, ser. Lecture Notes in Computer Science, F. Daniel, G. A. Papadopoulos, and P. Thiran, Eds. Springer Berlin Heidelberg, 2013, vol. 8093, pp. 248–259.
[10] H. Mahkonen, T. Rinta-aho, T. Kauppinen, M. Sethi, J. Kjällman, P. Salmela, and T. Jokikyyny, "Secure m2m cloud testbed," in *Proceedings of the 19th Annual International Conference on Mobile Computing & Networking*, ser. MobiCom '13. New York, NY, USA: ACM, 2013, pp. 135–138.
[11] P. Wright and A. Manieri, "Internet of things in the cloud - theory and practice," in *CLOSER 2014 - Proceedings of the 4th International Conference on Cloud Computing and Services Science, Barcelona, Spain, April 3-5, 2014.*, 2014, pp. 164–169.
[12] C. Doukas and F. Antonelli, "Compose: Building smart amp; context-aware mobile applications utilizing iot technologies," in *Global Information Infrastructure Symposium, 2013*, Oct 2013, pp. 1–6.
[13] S. Papavassiliou, C. Papagianni, S. Distefano, G. Merlino, and A. Puliafito, "M2m interactions paradigm via volunteer computing and mobile crowdsensing." in *Machine-To-Machine Communications - Architectures, Technology, Standards, and Applications*, J. M. Vojislav Misic, Ed. Oxford: Taylor & Francis, 2014.
[14] "Arduino [URL]," http://www.arduino.cc.
[15] "BaTHOS [URL]," https://github.com/ciminaghi/bathos-mcuio.
[16] M. Fazio, G. Merlino, D. Bruneo, and A. Puliafito, "An architecture for runtime customization of smart devices," in *Network Computing and Applications (NCA), 2013 12th IEEE International Symposium on*. IEEE, 2013, pp. 157–164.
[17] D. Bruneo, T. Fritz, S. Keidar-Barner, P. Leitner, F. Longo, C. Marquezan, A. Metzger, K. Pohl, A. Puliafito, d. raz, A. Roth, E. Salant, I. Segall, M. Villari, Y. Wolfsthal, and C. Woods, "Cloudwave: Where adaptive cloud management meets devops," in *Computers and Communication (ISCC), 2014 IEEE Symposium on*, vol. Workshops, June 2014, pp. 1–6.
[18] "AllJoyn [URL]," http://allseenalliance.org.