Cloud-Based Network Virtualization: An IoT Use Case

Giovanni Merlino[1,2], Dario Bruneo[1], Francesco Longo[1], Salvatore Distefano[3,4], and Antonio Puliafito[1]

[1] Università di Messina, Dipartimento DICIEAMA, Contrada di Dio, 98166 Messina, Italy, {gmerlino,dbruneo,flongo,apuliafito}@unime.it

[2] Dipartimento DIEEI, Università di Catania, Viale Andrea Doria 6, 95125 Catania, Italy giovanni.merlino@dieei.unict.it

[3] Dipartimento DEIB, Politecnico di Milano, Piazza L. Da Vinci 32, 20133 Milano, Italy salvatore.distefano@polimi.it

[4] Kazan Federal University, Kazan, Russia s_distefano@it.kfu.ru

Abstract. In light of an overarching scheme about extending the capabilities of Internet of things (IoT) with Cloud-enabled mechanisms, network virtualization is a key enabler of infrastructure-oriented IoT solutions. In particular, without network virtualization infrastructure cannot really be considered flexible enough to meet emerging requirements, and even administrative duties, such as management, maintenance and large-scale automation, would turn out to be brittle and addressed by special casing, leading to loss of generality and a variety of corner cases. We propose a Cloud-based network virtualization approach for IoT, based on the OpenStack IaaS framework, where its networking subsystem, Neutron, gets extended to accomodate virtual networks and arbitrary topologies among virtual machines and globally dispersed smart objects, whichever the setup and constraints of the underlying physical networks. This work outlines a motivating use case for our approach, and the ensuing discussion is provided to frame the benefits of the underlying design.

Keywords: IoT, Cloud, OpenStack, network virtualization, WebSocket.

1 Introduction

In the domain of the Internet of Things (IoT) [1], existing solutions are mainly focused on a lower layer, mostly dealing with communication aspects to interconnect network-enabled devices and, generally, *things* to the Internet.

However, from a higher level perspective, specific facilities for management, organization, and coordination of devices, sensors, objects and things are also

N. Mitton et al. (Eds.): AdHocNets 2015, LNICST 155, pp. 199–210, 2015.
DOI: 10.1007/978-3-319-25067-0_16

required to build up a dynamic infrastructure. To this purpose, on the one hand the capabilities provided by existing solutions in the management of distributed systems, ensuring flexibility and dealing with the complexity of large scale systems, should be exploited to implement basic mechanisms and tools for the resource management, also taking into account IoT solutions. On the other hand, it is necessary to provide and implement advanced solutions and policies able to manage and control the IoT infrastructure, implementing strategies aiming at satisfying higher (applications and end users) requirements, on top of basic facilities provided at a lower level. This two-layer model recalls the *Software Defined Ecosystem* model, where the data plane provides basic, customizable functionalities and the control plane implements advanced mechanisms and policies to control the ecosystem by enforcing strategies on nodes and objects through the lower level basic mechanisms. Thus, the main idea proposed in this paper is to treat the IoT domain as a Software Defined Ecosystem, adopting a two-layer Software Defined model to manage the underlying infrastructure.

To implement such a concept, Cloud computing facilities, applying a service-oriented approach in the provisioning and management of resources, may be exploited. The Cloud-based approach could be a good solution to address IoT-related issues, fitting with the requirements of relevant service users and application providers: on-demand, elastic and QoS-guaranteed, to name a few, all needed properties for an IoT service platform, to be addressed mainly at the control plane.

The contribution of this paper can be summarized as: a requirement analysis for an enhanced IaaS framework able to include and provide facilities for reconfigurable and complex aggregations of IoT devices; an architecture of node-side modules and the corresponding mechanisms needed to empower ubiquitous virtualization of networking functions; a scenario coupled with a related use case, where the approach enables a seamless exploitation of field deployments for IoT devices.

The remainder of this paper is organized as follows: Section 2 describes the reference architecture of a framework implementing the network virtualization for IoT following a service-oriented Cloud model. Then, Section 3 discusses a use case, highlighting pros and cons of the approach. Some remarks and considerations in Section 4 close the paper.

2 Reference Architecture

2.1 Requirements for Cloud-Enabled IoT

The main actors in any IoT scenario are *contributors* and *end users*. Contributors provide sensing and actuation resources building up the “things” infrastructure pool. 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 raw resources and implementing applications and services on top of it. We assume that sensing and actuation resources are provided to the infrastructure pool via a number of hardware-constrained units, from now on referred to as *nodes*. Nodes host sensing and actuation resources and act as mediators in relation to the Cloud infrastructure.

In order to actually accomplish the prospect of a Cloud-based IoT system, a systematic requirement analysis is needed. A subset of requirements are the ones relative to the contributor:

– **Out-of-the-box experience** - letting nodes and the corresponding sensors and actuators be enrolled automatically in the Cloud at, e.g., 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).

and others coming from the end user such as:

– **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 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.
– **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., NAT) and configuration (e.g., firewall).
– **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.

A certain subset of end user requirements instead needs to be addressed by just providing the facilities for centralized orchestration of virtualized networking instances.

In relation to the latter, the list includes:

– **Service-oriented interfaces** - exposing primitives as asynchronous service endpoint, in order to ease development and third-party software integration.
– **Environment customization** - enabling runtime modifications to the software environment hosted by the node.
– **Topology rewiring** - providing mechanisms for the networking configuration underneath nodes to be modified at any time.

2.2 Sensing and Actuation as a Service for IoT

In the pursuit for integration of IoT infrastructure with paradigms and frameworks for heterogeneous resource management, we are trying to follow a bottom-up approach, consisting of a mixture of relevant, working frameworks and protocols, on the one hand, and interesting use cases to be explored according to such integration effort, on the other.

Indeed, beyond concerns about the scale of the effort, other requirements such as elasticity of the sensing-based services to be provided, as well as registration and provisioning mechanisms of the underlying heterogeneous sensor-hosting platforms deserve an Infrastructure Manager (IM) anyway. To this purpose, Cloud computing facilities, here also implementing a service-oriented [2] approach in the provisioning and management of sensing and actuation resources, are exploited to enable a *Sensing and Actuation as a Service* (SAaaS) paradigm for IoT. In fact, in the SAaaS perspective, sensing and actuation devices should be handled along the same lines as computing and storage abstractions in traditional Clouds, i.e., on the one hand virtualized and multiplexed over (scarce) hardware resources, and on the other grouped and orchestrated under control of an entity implementing high level policies. This way, 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.

A Cloud-oriented solution indeed may fit IoT scenarios, meeting most requirements by default to cater to the originally intended user base, while at the same time also addressing other more subtle functionalities, such as a tenant-based authorization framework, where several actors (owners, administrator, users) and their interactions with infrastructure may be fully decoupled from the workflows involved (e.g., transfer, rental, delegation). Bonus points include recycling existing (compute/storage-oriented) deployments, getting most visualization and monitoring technologies for free, as those are typically already available in such systems, possibly even enabling federation of different administrative Cloud-enabled domains.

In this sense, our choice leans towards OpenStack, as a centerpiece of infrastructure Cloud solutions for most commercial, in-house and hybrid deployments, as well as a fully OpenSource ecosystem of tools and frameworks upon which many EU projects, such as CloudWave (FP-7), are founding their Cloud strategies. Our prototype is thus based on OpenStack and named Stack4Things.

Indeed, choosing an industrial strength solution for infrastructure Clouds lets us eschew at the moment scalability and other generic performance issues, and focus most of the discussion on the challenges which are relevant to centralized management of IoT.

Putting aside the core IaaS framework, as anticipated some additional facilities are needed for our envisioned SAaaS paradigm and the specifics of the domain at hand (IoT), among which here we may describe two classes of mechanisms that are core to the overall approach: those needed to access locally and

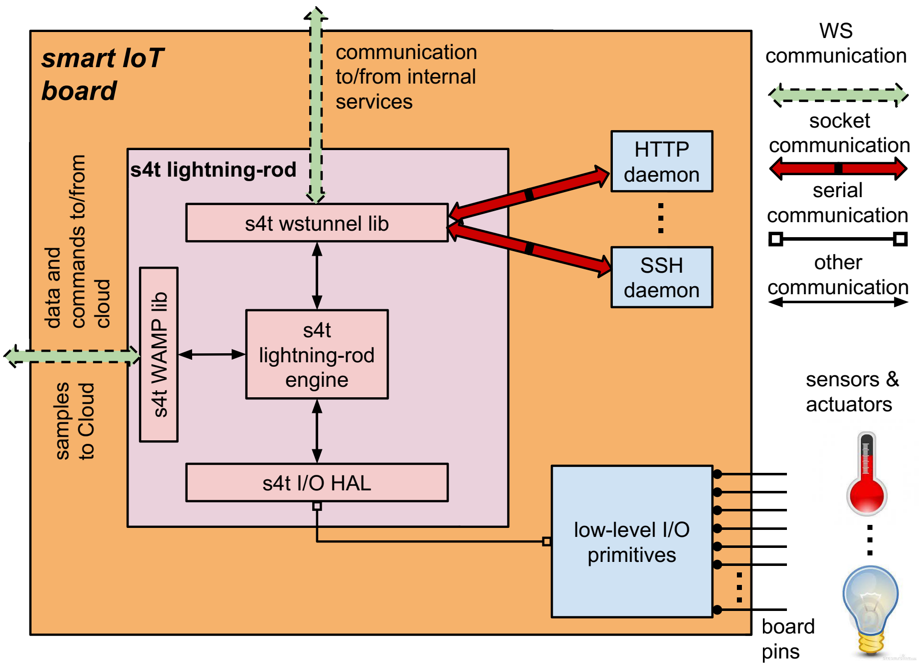

Fig. 1. Stack4Things node-side stack: logical architecture.

transparently remote (I/O) resources, and those to set up arbitrary topologies among nodes.

With regard to the former, in Figure 1 we find a logical architecture of the node-side stack needed for pub/sub or even RPC-style I/O primitives to be exposed to remote hosts through the Cloud. The *Stack4Things lightning-rod*, acting as SAaaS Client, runs on the IoT board 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. WebSocket is a standard HTTP-based protocol providing a full-duplex TCP communication channel over a single HTTP-based persistent connection. One of the main advantages of WebSocket is that it is network agnostic, by just piggybacking communication onto standard HTTP interactions. This is of benefit for those environments which block Web-unrelated traffic using firewalls. *Web Application Messaging Protocol (WAMP)* [3] is a sub-protocol of WebSocket, specifying a communication semantic for messages sent over WebSocket, providing both publish/subscribe and routed remote procedure call (RPC) mechanisms.

The I/O HAL (*hardware abstraction layer*) is equipped with a set of extensions exposing the board digital/analog I/O pins to the hosted environment. In particular, functionalities provided by the HAL include enumeration of the pins and exporting corresponding handlers for I/O in the form of i-nodes of a virtual filesystem.

The *Stack4Things lightning-rod engine* represents at the core of the board-side software architecture. The engine interacts with the Cloud by connecting to a WAMP router through a WebSocket-based 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, among other things, to the communication with the board digital/analog I/O pins and thus with the connected sensing and actuation resources. The communication with the Cloud is ensured 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., SSH, 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 to interact with the board service. New REST resources are automatically created exposing the user-defined commands on the Cloud side. As soon as such resources are invoked the corresponding code is executed on top of the smart board.

Cloud-Based Virtualized Networking for IoT. Figure 2 shows a conceptual depiction of the tunnel-based layering model employed for Cloud-enabled set up of virtualized bridged networks among nodes across the Internet.

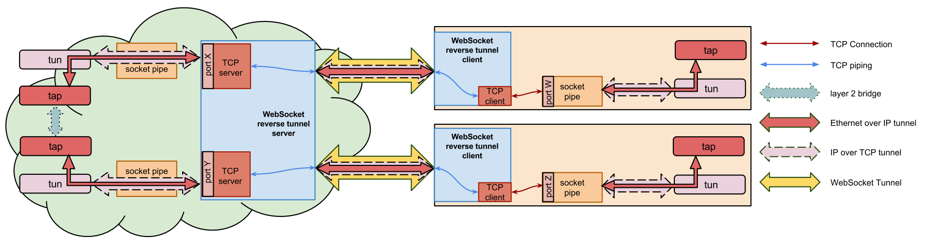

Fig. 2. Stack4Things tunnel-based layering: model.

It is important to remark that the kind of tunneling here mentioned is essential to obtain remote access to IoT resources whichever the constraints of the network nodes reside in, a prerequisite to expose node-hosted resources according to the aforementioned access patterns.

The basic remoting mechanisms are based on the creation of generic TCP tunnels over WebSocket (WS), a way to get client-initiated connectivity to any server-side local (or remote) service. In this sense, we devised the design and implementation of an incremental enhancement to standard WS-based facilities, i.e., a *reverse* tunneling technique, as a way to provide server-initiated, e.g., Cloud-triggered, connectivity to any board-hosted service.

Beyond mere remoting, level-agnostic network virtualization needs mechanisms to overlay network- and datalink-level addressing and traffic forwarding on top of such a facility. Here the novelty of setting up VPNs on top of WebSocket lies in the decoupled control machinery, and the inherent flexibility of an on-demand mechanism. The former indeed is enabled through a preliminarily activated and always-on WebSocket-based *control* reverse tunnel (rtunnel), acting as an out-of-band channel for command streams.

There are already certain solutions [4] for setting up VPNs on top of WS, but without decoupled control machinery nor the inherent flexibility of an on-demand mechanism.

Focusing the analysis on the instantiation of, e.g., a virtual bridge between two boards, over *data* (in-band) rtunnels, a first step lies in setting up a TCP connection based on a WS-based rtunnel, which consists in exposing, on the server side, a listening socket on a local port, as soon as the rtunnel server accepts a request for a new rtunnel. The TCP connection just established gets piped to the rtunnel that encapsulates TCP segments in a WS-based stream. On the WS rtunnel client side, as soon as the rtunnel is established, a new TCP client is brought up connecting to a local listening port, and such TCP connection gets piped to the rtunnel. A level-3 tunnel is then to be established over this TCP-based tunnel by launching an application that starts up in listening mode on both sides of the socket pipe and, on connection, starts exposing a virtual (TUN) device on either side, both set up with IP addresses of choice, as long as those belong to the same subnet. The workflow could end here if the request was for a layer-3 VPN.

In order to set up instead a level-2 encapsulation over the aforementioned IP-based communication, the system has to bring up a GRE tunnel, where the endpoints are the previously configured TUN IPs and the type of tunnel-hosting virtual device is set to TAP, thus exposing an Ethernet-compatible interface. Adding such interface to a dedicated virtual bridge on the server ends the workflow, in this case exposing a layer-2 VPN. For IP-based tunneling, we resorted to *Generic Routing Encapsulation (GRE)* [5], an IETF standard for a no-frills IP-in-IP tunneling protocol. GRE support is not limited to level-3 encapsulation, but also available for tunneling of level-2 (Ethernet) frames over to the corresponding virtual (TAP) device.

3 Use Case

Once the Cloud-based IoT scenario has been laid out, it is easier to frame the discussion in terms of a focused scenario, such as management of large-scale emergency situations.

A peculiar feature of such scenario lies in the lack of predefined boundaries in terms of the sensing infrastructure, which may span multiple geographical areas and administrative domains. Whichever the footprint of alerting and support activities for civilians, the foremost quality here is the dynamic involvement of infrastructure.

3.1 Opportunistic Exploitation and Transparent Field Upgrade of IoT-Based Facilities

In such a scenario a use case may be identified in the on-demand setup of facilities that are ready to react to certain events which could anticipate an impending emergency, and may avoid or at least contain damages and/or casualties. For instance, a bridge may be considered at risk and put under control by placing the required sensing infrastructure to monitor critical parameters, such as oscillations, load, and torque or compressive stress of certain sections and elements. In terms of actuators, the most fitting example may be gates at either side of the bridge, only involving entry lanes in order not to impact vehicular outflow, to be closed at the occurrence of such kind of event, as a precautionary step to be taken before deeper investigations.

Such potential infrastructure thus needs reactive mechanisms in place, possibly encoded as statements for a Complex Event Processing engine.

The aforementioned use case may indeed be implemented by deploying at least two transducers, a sensor and an actuator respectively, where a board driving an actuator hosts an application that operates it when triggered upon detection of an event of interest. The latter gets generated by a CEP engine every time predefined patterns (e.g., steady-state and/or structural anomalies) get recognized out of measurements by one or more sensors sampling the corresponding phenomena on (possibly other) boards.

Interesting patterns are set by loading rules written in an engine-specific language.

The interactions are here described, when requesting for a number of boards currently enrolled to the Cloud to be booked, mapped to an enumerable set of resources, ultimately exposed for seamless interaction to a CEP engine, and the corresponding rules, deployed in Cloud-hosted VM.

According to the description of our core mechanisms, built on top of the IaaS framework, the first request is a routine one for the framework once extended to include enrollment of IoT nodes, as well as the second and the fourth one even when IoT extensions are not considered. The third request instead requires the framework to deploy (IaaS-context) data into a VM, but then the enumeration may take place only if the WAMP subsystem is available. Exposing remote resources as local I/O needs a wrapper around the same subsystem too.

An opportunistic exploitation of resources yet gets feasible only when expecting to be able to avoid an operator to set up the whole (distributed sensing and actuation) system beforehand in the finest detail, including runtime adaptations such as, e.g., swapping part of the logic and replacing nodes to be involved, when needed.

In particular a useful approach may lie in the field deployment of an array of devices with sensing functions the (aggregate) coverage of which is not necessarily known in advance or perfectly partitioned somehow. As long as this set of resources may be set up as an inter-node addressable ensemble and has a running mechanism in place for the election and maintenance of a master node, event detection may be delegated to the latter. In turn the detection routines would leverage as much information as possible by aggregating data originating from whichever resource is part of the aforementioned ensemble. For such an autonomous system to work, sensors advertisement and discovery services are of course a prerequisite.

The master node may as well be leveraged to invoke one or more actuators (e.g., close the gates) should a predefined emergency event be detected, by also discovering relevant actuating resources through the same mechanisms. Unplanned field deployment coupled with this approach may thus lead to a seamless exploitation of resources, where even replacement for upgrade, or loss of a subset of nodes is not disruptive to the working status of the system.

A side effect of the choice to lift some decisions and duties off the operator may also lie in the ability to make the system fault tolerant by design, especially when employing an approach of redundant deployment on the actuating side, as resilience benefits in this case from the transparent addition or replacement of nodes.

An operator thus only needs to reserve a set of nodes once, roughly by function category or even better by geographical area, and just resort to the SAaaS framework for the corresponding setup (and runtime adjustments) of an inter-node configuration based on ***Virtualized bridging for IP-based transport of discovery services***.

As said previously, we are able to get remote access to the boards, for instance for deploying an application, from anywhere a client may connect to any Cloud-enrolled board, whichever its connectivity (e.g., node-side NAT or firewall notwithstanding). We may submit a request for certain nodes on demand as resources, and another to arrange a certain topology among boards by network virtualization, in order to accomodate the requirements of the application itself, by leveraging the (wide-area) *control plane*.

In particular, an interesting case is that of the AllJoyn [6] framework for IoT, a family of standards and reference implementations which comprises at its core a DBus-derived application protocol useful for messaging, advertisement and discovery of services, working via selected mechanisms on available transports. As long as the application is based on AllJoyn, services may be discovered automatically, and thus leveraged according to the logic of the application. The distributed system works by letting these boards interact through AllJoyn over an IP-based network and the corresponding transport implementation, where mDNS and a combination of multicast and broadcast UDP packets are used. A limitation indeed is that the protocol is currently designed to work only as long as the communicating boards are on the same broadcast domain. Therefore, such a case may be covered by being able to leverage the Cloud to instantiate a (virtualized, wide-area) bridged network among the nodes, coupled with the availability of remote access for deployment and execution of the required binaries.

Under the assumption that nodes are not globally addressable or otherwise reachable on whichever port, i.e., behind a firewall/NAT system due to an ubiquitous IPv4 setup, a complex interaction flow is required to provide this kind of abstractions and the underlying connectivity.

Supposing thus the boards to be bridged are already registered to the Cloud, a high-level description of the workflow, from the point of view of the user, comprises the following steps:

1. Book two (or more) managed boards.
2. Request for a bridge among the reserved boards.
3. Request for exposing SSH service on every reserved board.
4. Connect via SSH service to every reserved board for deploying and launching the AllJoyn application.

Focusing on the unique steps of the one under consideration, the first request gets serviced by leveraging the virtualized networking facilities and the second one by tunneled remoting.

The following list of sequences is then expected to take place, with (low-level) operations as depicted and numbered in Fig. 3.

1) The user requests the setup of a bridge between two specific boards, either through the s4t dashboard or, in alternative, through the s4t command line client.
2) The s4t dashboard performs one of the available s4t IoTronic APIs calls via REST, which pushes a new message into a specific AMQP IoTronic queue.
3) The s4t IoTronic conductor pulls the message from the AMQP IoTronic queue and correspondingly performs a query on the s4t IoTronic database. In particular, it checks if the board is already registered to the Cloud and looks up the s4t IoTronic WAMP agent to which the board is registered. At last, it decides the s4t IoTronic WS tunnel agent to which the user can be redirected and randomly generates a free TCP port.
4) The s4t IoTronic conductor pushes a new message into a specific AMQP IoTronic queue.
5) 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.
6) Through the s4t WAMP lib the s4t lightning-rod engine receives the message by the WAMP router.
7) The s4t lightning-rod engine sets up a rtunnel with the s4t IoTronic WS tunnel agent specified by the s4t IoTronic conductor, also providing the TCP port through the s4t wstunnel lib. It also brings up a number of sockets to be piped and overlaid over the rtunnel, plus the corresponding virtual interfaces, as described in Sec. 2.2.

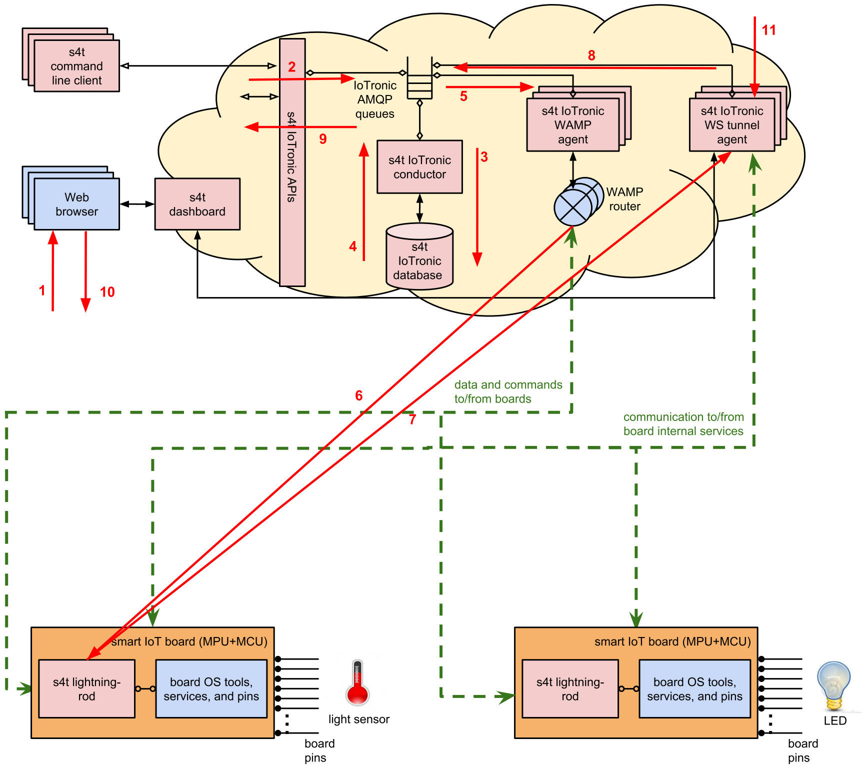

Fig. 3. Workflow and interactions between Cloud and board for the use case.

8) The s4t IoTronic WS tunnel agent follows up with its own set of server-side network virtualization duties, still according to Sec. 2.2. Then, it publishes a new message into a specific AMQP IoTronic queue confirming that the operation has been correctly executed.
9) The s4t IoTronic APIs call pulls the message from the AMQP IoTronic queue and replies to the s4t dashboard.
10) The user gets notified of the success of the operation.

This first sequence has to be replicated for both nodes, as well as the following two. In order not to stretch the description, here only phases which are different from the previous one are outlined. In particular, the second sequence (remote access) steps 2-6,9 remain unchanged, step 1,7-8,10 are changed as follows:

1) The user asks for a connection to the SSH service local to a specific board, either through the s4t dashboard or, in alternative, through the s4t command line client.
2) The s4t lightning-rod engine sets up a rtunnel with 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.

3) The s4t IoTronic WS tunnel agent brings up a TCP server on the specified port, and then publishes a new message into a specific AMQP IoTronic queue confirming that the operation has been correctly executed.
4) The s4t dashboard provides the user with the IP address and TCP port that she can use to connect to the SSH daemon running on the board.

And an additional step is present:

5) As the user employs an SSH client to connect to the specified IP address and TCP port, the session is tunneled right to the board.

4 Conclusions

In this paper, we presented a new paradigm that can be considered as an approach to provide a simplified and programmable exploitation of the underlying ecosystem of devices so that innovative and powerful services can be realized. Starting from the well known concept of Software Defined paradigms (e.g., separating control and data planes) a Cloud-based framework is proposed, taking advantage of off-the-shelf technologies (e.g., OpenStack) and extending the computing and storage virtualization concepts also to the sensing and actuating facilities. Architectural aspects have been discussed as well as implementation choices. Future work will include the validation of the whole architecture in a real-world scenario involving hundreds of devices, under the #SmartME project.

References

1. Gubbi, J., Buyya, R., Marusic, S., Palaniswami, M.: Internet of things (iot): A vision, architectural elements, and future directions. Future Generation Computer Systems 29(7), 1645–1660 (2013)
2. Distefano, S., Merlino, G., Puliafito, A.: Sensing and actuation as a service: A new development for clouds. In: 2012 11th IEEE International Symposium on Network Computing and Applications (NCA), pp. 272–275, August 2012
3. Fette, I., Melnikov, A.: The WebSocket Protocol. RFC 6455, RFC Editor, December 2011
4. VPN-WS. `https://github.com/unbit/vpn-ws`
5. Hanks, S., Li, T., Farinacci, D., Traina, P.: Generic Routing Encapsulation (GRE). RFC 1701, RFC Editor, October 1994
6. AllJoyn. `http://allseenalliance.org`