Ann. Telecommun. (2017) 72:53–70
DOI 10.1007/s12243-016-0528-5

Stack4Things: a sensing-and-actuation-as-a-service framework for IoT and cloud integration

Francesco Longo[1] · Dario Bruneo[1] · Salvatore Distefano[1,2] · Giovanni Merlino[1] · Antonio Puliafito[1]

Received: 30 September 2015 / Accepted: 30 May 2016 / Published online: 7 June 2016

Abstract With the increasing adoption of embedded smart devices and their involvement in different application fields, complexity may quickly grow, thus making vertical ad hoc solutions ineffective. Recently, the Internet of Things (IoT) and Cloud integration seems to be one of the winning solutions in order to opportunely manage the proliferation of both data and devices. In this paper, following the idea to reuse as much tooling as possible, we propose, with regards to infrastructure management, to adopt a widely used and competitive framework for Infrastructure-as-a-Service such as OpenStack. Therefore, we describe approaches and architectures so far preliminary implemented for enabling Cloud-mediated interactions with droves of sensor- and actuator-hosting nodes by presenting Stack4Things, a framework for Sensing-and-Actuation-as-a-Service (SAaaS). In particular, starting from a detailed requirement analysis, in this work, we focus on the subsystems of Stack4Things devoted to resource control and management as well as on those related to the management and collection of sensing data. Several use cases are presented justifying how our proposed framework can be viewed as a concrete step toward the complete fulfillment of the SAaaS vision.

Keywords IoT · Cloud · OpenStack · WebSocket · WAMP · SAaaS

✉ Giovanni Merlino
gmerlino@unime.it

[1] Università degli Studi di Messina, Messina, Italy

[2] Kazan Federal University, Kazan, Russia

1 Introduction

In the last years, the Internet of Things (IoT) has emerged as one of the hottest trend in ICT thanks to the proliferation of field-deployed, dispersed, and heterogeneous sensor- and actuator-hosting platforms. Recently, with the increasing development of powerful and flexible embedded systems characterized by reprogrammable behavior and ease of use, such *things* are gaining a "smart" labeling to indicate this evolution. This all-encompassing and much ambitious scenario calls for adequate technologies.

Several solutions are already present in the literature mainly focusing on lower (communication) layer and in particular on how to interconnect (among themselves and to the Internet) any network-enabled *thing* [1]. However, in order to realize the Sensing-and-Actuation-as-a-Service (SAaaS) vision [2], other aspects have to be also taken into account such as solutions for creating and managing a dynamic infrastructure of sensing and actuation resources. In fact, in order to effectively control devices, sensors, and things, several mechanisms are strongly needed, e.g., management, organization, and coordination. Then, a middleware devoted to management of both sensor- and actuator-hosting resources may help in the establishment of higher-level services.

In this direction, the integration between IoT and Cloud is one of the most effective solutions even if up to now 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, by presenting Stack4Things[1] an OpenStack-based framework implementing the SAaaS paradigm. Thanks to such a framework, it is possible to manage in an easily way fleets of sensor- and actuator-hosting boards regardless of their geographical position or their networking configuration.

Preliminary details of the Stack4Things[2] architecture have been presented in [6]. In this paper, starting from a detailed requirement analysis, we describe the whole Stack4Things architecture by focusing on both Cloud and board components. In designing Stack4Things, we followed a bottom-up approach, consisting of a mixture of relevant technologies, frameworks, and protocols. In addition to the already cited OpenStack, we take advantage of the WebSocket [7] technology and we base our communication framework on the web application messaging protocol (WAMP) [8].

The rest of the paper is organized as follows: Section 2 features a review of the literature; Section 3 introduces OpenStack and presents all the technologies exploited for the implementation of the Stack4Things framework; Section 4 describes the Stack4Things architecture both from the point of view of the IoT devices and of the Cloud infrastructure focusing on the control and management of sensing and actuation resources and on the collection of sensing data; Section 5 gives a detailed view of the defined REST API; Section 6 shows some specific use cases that have been implemented and tested in the Stack4Things middleware; and Section 7 closes the paper with some discussion about the business perspectives around the proposed paradigm, and how the framework can be extended to include additional features and functionalities.

2 Related work

The convergence of Cloud and IoT, especially in terms of the Cloud as a support for IoT-based applications, has been extensively investigated in recent years. For instance, some authors [9] have surveyed commercial and open source platforms for data gathering and processing platforms for smart devices, while also proposing an architecture for a platform of this kind, mostly focused on data interchange formats and communication layers. In [10], a Cloud platform for fully distributed interacting objects is presented, where most of the issues are tackled in terms of networking capabilities and dissemination of system-wide metadata by means of distributed hash tables. An overview of principles for how to engineer an IoT/Cloud integration is exposed in [11], hinting at so-called "software-defined machines" as collections of IoT resources and Cloud-provided ones, a way to spread execution of the (IoT) application stack across IoT devices and Clouds, albeit still looking at IoT applications as hierarchies of engines for analytics, at increasingly smaller scales as processing gets pushed to the edge. There is also a popular line of research in the community which looks at the integration of IoT and Cloud as a way to establish Cyber-Physical Cloud Computing Systems [12], thus exploring virtual sensing in order to compose and orchestrate sensor-based services, in the end devising a hierarchy of control. What characterises our approach in comparison with the aforementioned efforts is a lack of any predefined, possibly rigid, scheme with regards to how to mix and match resources to empower the IoT application, thus leading to higher degrees of freedom in how to engineer any vertical of interest. This advantage stems naturally from the Infrastructure-as-a-Service approach to IoT devices as sensor- and actuator-hosting nodes, while also including further stakeholders and actors with respect to plain IaaS, e.g., third-parties as contributors in addition to owners alone. In this sense, leveraging an industrial-strength open source IaaS platform brings benefits and functionalities of its own, e.g., a service-oriented role-based model for authentication and delegation, built-in to the preexisting codebase, compared to ad-hoc solutions, further stressing the horizontal nature of the paradigm and design. Moreover, integrating IoT management and resource provisioning in OpenStack has not been independently explored by the development community in terms of prototypes or even blueprints yet. Still clinging to the approach, there is also scope for unique design choices, such as Websocket-based communication and messaging which, to the best of our knowledge, has not been investigated nor applied in the research domain of IoT/Cloud integration.

3 Background

All these efforts are manly focused on a data-centric perspective, mainly aiming at managing (IoT sensed) data by the Cloud. In [2], a different approach is adopted, where the goal is to provide actual sensing and actuation resources that could be handled by their (Cloud) users, as computing and storage resources in IaaS or DaaS Clouds, i.e., virtualized and multiplexed over (scarce) hardware resources. In other words, the proposed approach aims at adopting the service-oriented/Cloud paradigm in the management

[1] We are developing all our software as Open Source, making it freely available through the Web [5].

[2] From now on, Stack4Things is sometimes abbreviated as s4t.

sensing resources and things instead of considering the Cloud as just a complementary technology, adopting a device-centric perspective [13] toward the SAaaS and the Things as a Service (TaaS) [14] paradigm.

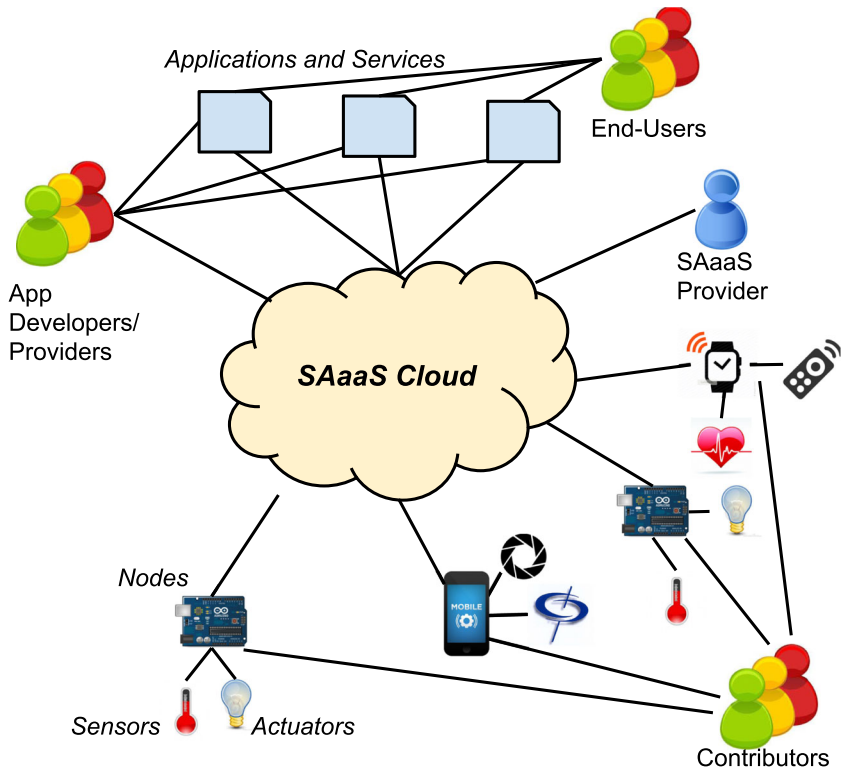
The scenario thus enabled is depicted, from a high level point of view, in Fig. 1. Four main actors are identified: (i) the *contributors*, voluntarily providing their sensing and actuation resources hosted in their nodes, i.e., boards or devices, to (ii) the *SAaaS provider*, building up and maintaining the infrastructure, the platform, and the services for gathering, configuring, and providing these resources as services to (iii) *app developers/providers*, deploying their applications on the sensing and actuation nodes, finally used by (iv) *end-users*.

In designing our solution for implementing this SAaaS vision, we based our efforts on Open Source technologies and standards. The latest *Arduino YUN* [15]-like boards represent our reference for the IoT nodes. Such a kind of devices is usually equipped with (low power) micro-controller (MCU) and micro-processor (MPU) units. They can interact with the physical world through a set of digital/analog I/O pins while connection to the Internet is assured by Ethernet and Wifi network interfaces. A Linux distribution (usually derived from the OpenWRT project) can run on the MPU. Recently, the use of *BaTHOS* [16] on the MCU side has spread, thus enabling the digital/analog I/O pins to be directly accessed from the MPU.

With respect to network connectivity, presence, and reachability, *WebSocket* [7] 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. WebSocket allows the HTTP server to send content to the browser without being solicited: messages can be passed back and forth while keeping the connection open creating a two-way (bi-directional) ongoing conversation between a browser and the server. One of the main advantages of WebSocket is that 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, 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) [8] 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 and provides both publish/subscribe (pub/sub) and (routed) remote procedure call (RPC) mechanisms. In WAMP, a router is responsible of brokering pub/sub messages and routing remote calls, together with results/errors.

Fig. 1 SAaaS scenario



3.1 OpenStack

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. Currently, OpenStack allows to manage virtualized computing/storage resources, according to the infrastructure Cloud paradigm. In Fig. 2, a conceptual architecture of OpenStack depicting components, as boxes, and the services they provide to other components, with arrows, are shown, respectively. Nova, the compute resource management subsystem, lies at the core of OpenStack and provisions VMs, with the help of a number of subsystems that provide core (e.g., networking in the case of Neutron) and optional services (e.g., block storage, in the case of Cinder) to the instances. Horizon is the dashboard and as such provides either a (web-based) UI or even a command-line interface to Cloud end users. Ceilometer, the metering and billing subsystem, like most other components of the middleware, cannot be fully analyzed on its own, as it needs to interface to, and support, Nova. In particular, while both Nova and any of the aforementioned

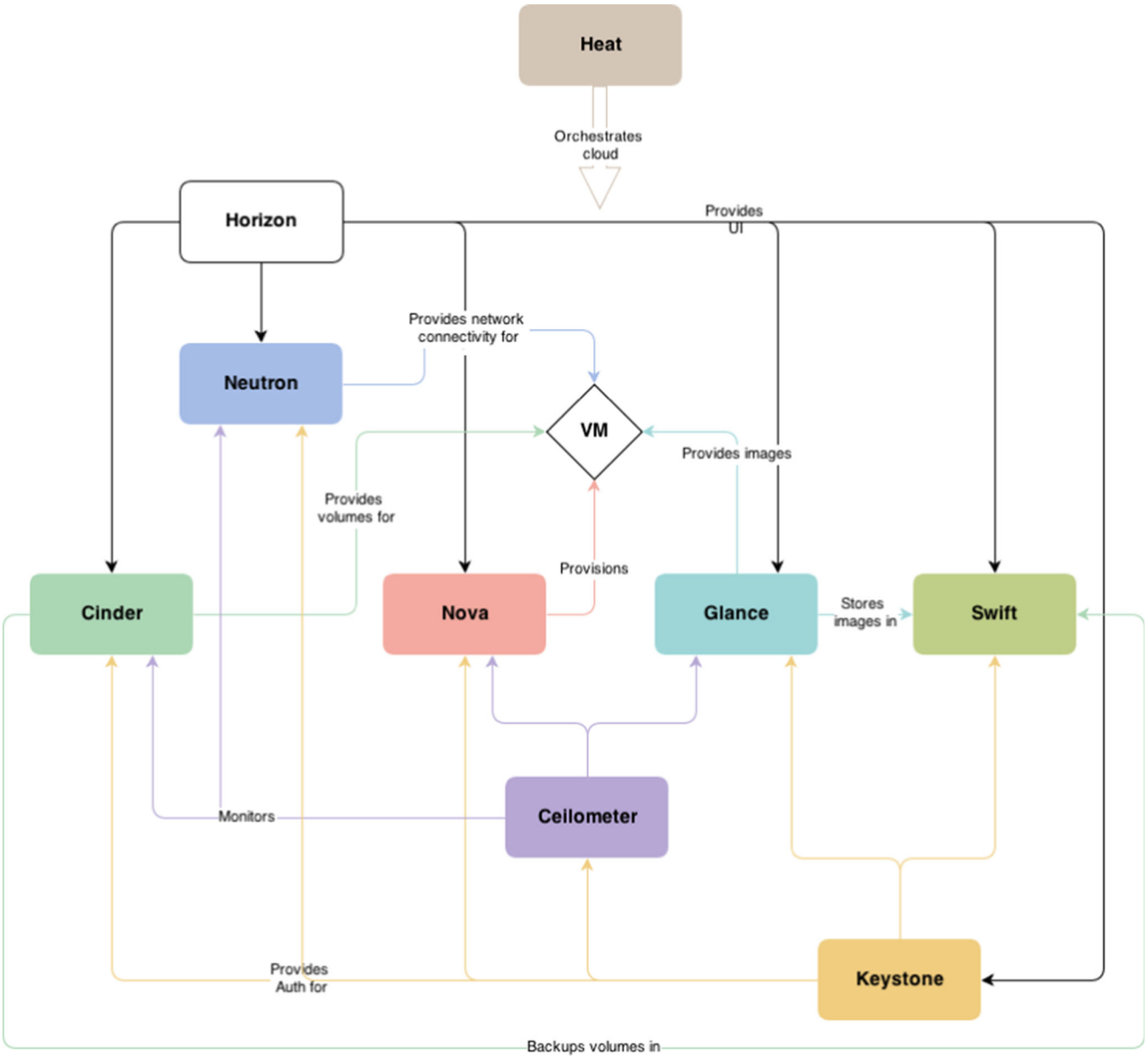


Fig. 2 OpenStack: conceptual architecture

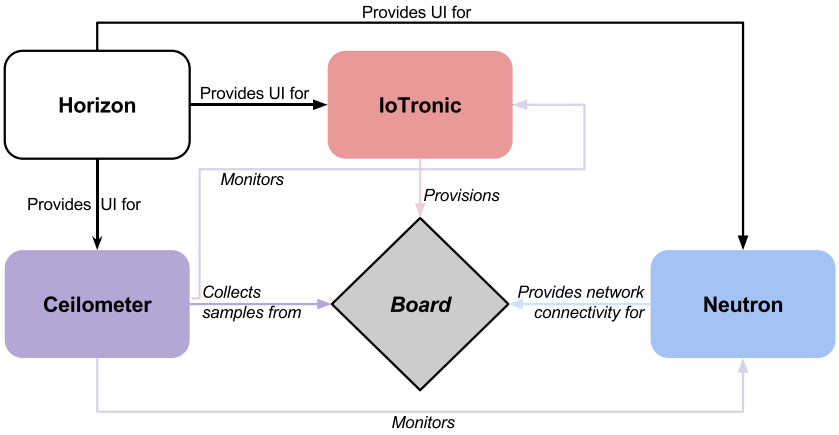


Fig. 3 Stack4Things core subsystems: conceptual architecture

subsystems exploit a common bus, the former alone dictates a hierarchy on participating devices, including their role and policies for interaction. Indeed, Nova requires a machine operating as Cloud *controller*, i.e., centrally managing one or more *Compute nodes*, which are typically expected to provide component-specific services (e.g., computing) by employing a resource-sharing/workload-multiplexing facility, such as an *hypervisor*. Our main goal in this paper is to propose an extension of OpenStack for the management and service-oriented exploitation of sensing and actuation resources.

Thus, shifting our analysis to the Cloud-side subsystem, the OpenStack component for metering (and billing) is Ceilometer, while Horizon is the dashboard, hence we turned first our attention on these frameworks in order to tackle data acquisition and visualization duties, for samples coming from the "things."

Highlighting this focus, as can be seen in Fig. 3, the same kind of conceptual architecture as in Fig. 2 gets simplified in order to take into account just core components for our approach, in particular introducing a novel subsystem, IoTronic, devoted to the provisioning of configuration and tasks for board-hosted sensing and actuation resources. Indeed, in place of a VM, in this case, we have a diamond-shaped box symbolizing a (transducer-hosting) board, and corresponding interactions are described as text in italic along the arrows.

With regard to Ceilometer, while metering is a feature needed for billing purposes, with monitoring as its enabling pattern, itself a core duty for any Cloud solution, in our case we partly reverse the perspective, as we are mostly interested in measurements for their own sake, collecting samples in order to monitor vital physical world parameters.

4 Stack4Things architecture

Figure 4 shows the Stack4Things overall architecture, focusing on communication between end users and sensor- and actuator-hosting nodes. We assume each of such nodes is an Arduino YUN-like smart board. On the board side, the *Stack4Things lightning-rod* 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

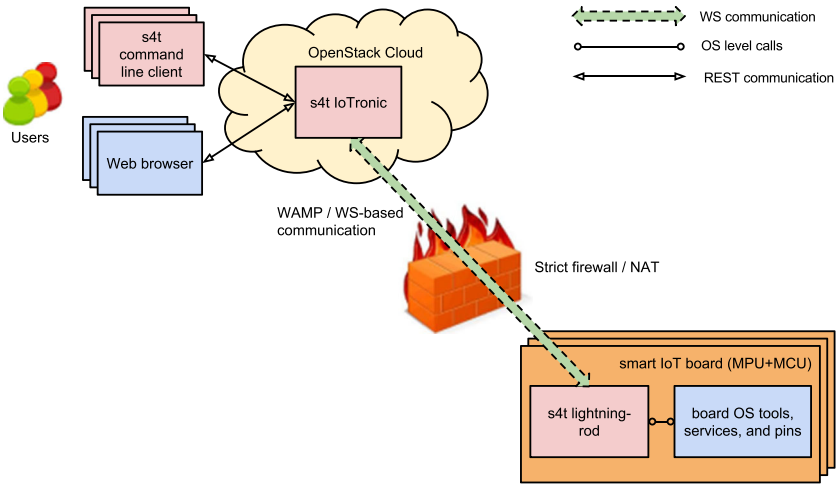


Fig. 4 Stack4Things overall architecture in the case of Arduino YUN-like boards

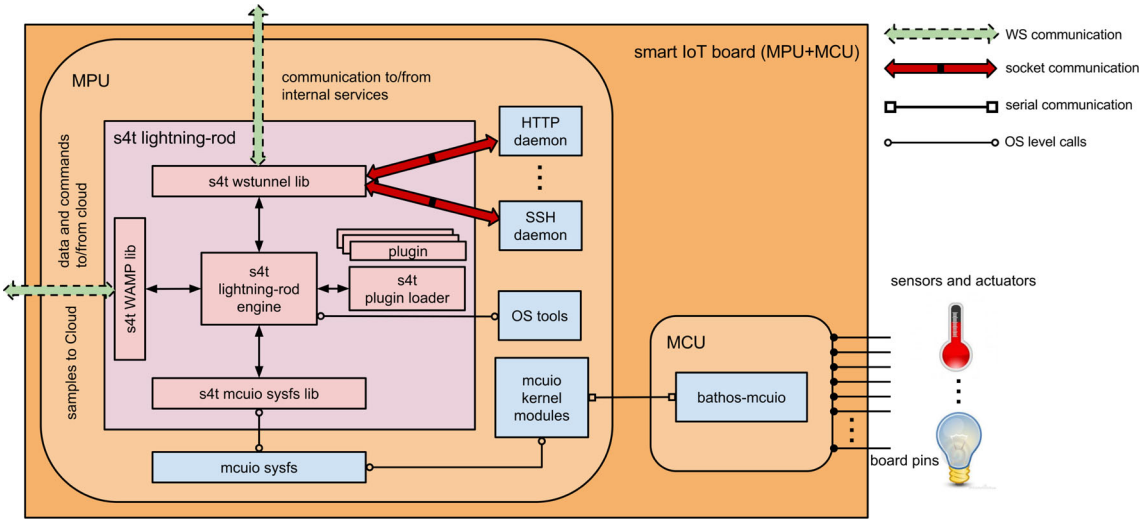


Fig. 5 Stack4Things board-side architecture

the point of contact with the Cloud infrastructure allowing end users to manage board resources even if they are behind a NAT or a strict firewall. This is ensured by a WAMP-based communication between the Stack4Things lightning-rod and its Cloud counterpart, namely the *Stack4Things IoTronic service*. The Stack4Things IoTronic service 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

Figure 5 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. Upward 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 Fig. 6) 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., 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 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

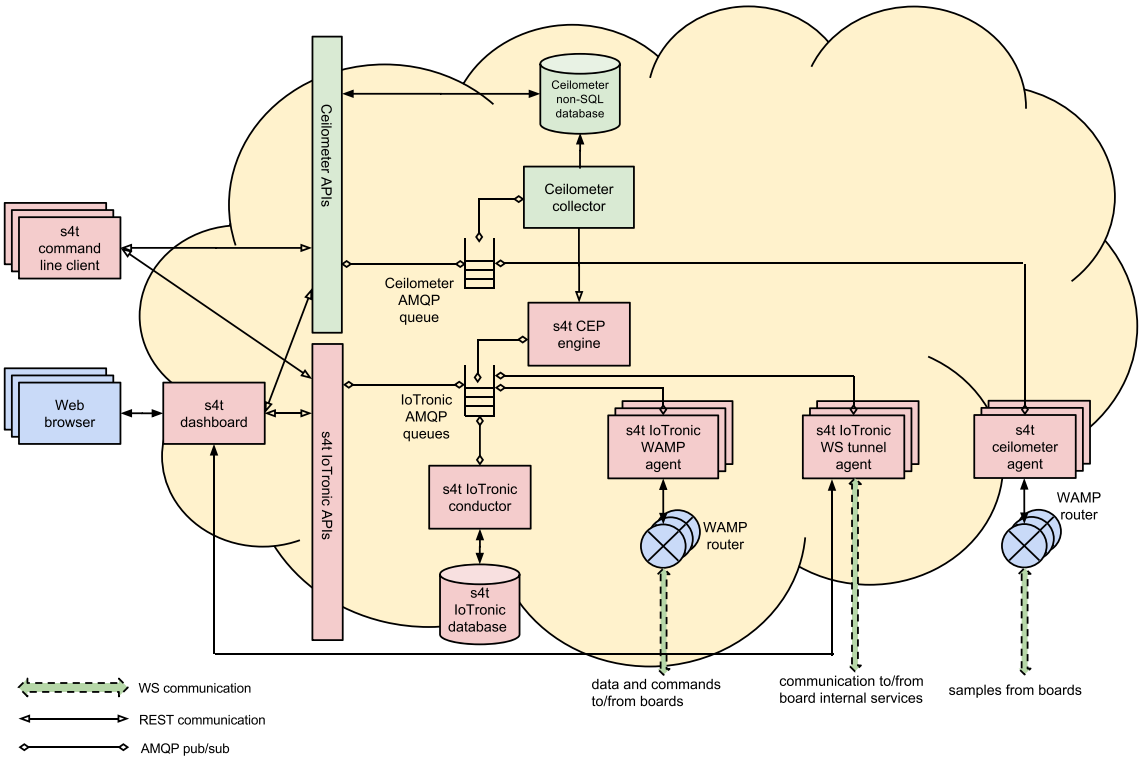


Fig. 6 Stack4Things Cloud-side architecture

Cloud and run on top of the plugin loader in order to implement specific user-defined commands, possibly including system-level interactions, e.g., with a package manager or the runlevels management subsystem.

In this sense, the authors may resort, in future efforts, to previous work [17] 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 resources are invoked, the corresponding code is executed on top of the smart board.

4.2 Cloud-side—control and actuation

The Stack4Things Cloud-side architecture (see Fig. 6) consists of an OpenStack service we called IoTronic. The main goals of IoTronic lie in extending the OpenStack architecture toward 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* expose 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 already mentioned in Section 3, a prominent 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.

4.3 Cloud-side—sensing data collection

The OpenStack service that collects monitoring data and events from the infrastructure (mainly for billing and elasticity purposes) is Ceilometer. We built on top of it in order to allow collection of metrics coming from the smart boards. In particular, we provide a *Stack4Things Ceilometer agent* to which smart boards that need to send metrics can connect. Such an agent translates the WAMP messages received by the boards to AMQP messages in the form of OpenStack notifications. Such notifications are then translated by the Ceilometer framework in samples that are collected by the Ceilometer collector and then stored in a non-SQL database (usually MongoDB). Metrics and events can be accessed through the Ceilometer APIs. The Stack4Things dashboard and the Stack4Things command line client are also able to interact with such APIs in order to obtain/visualize real-time and historical data. The Stack4Things framework also provides complex event processor (CEP) functionalities through the *Stack4Things CEP engine*. This engine can be programmed in order to detect specific situations of interest that can then be signaled to the Stack4Things IoTronic conductor which, in turn, can send commands to the smart boards in order to react to the situation by actuating actions or changing their behavior.

A prototype of the architecture so far described has been implemented and source code is freely available through the Web [5].

5 Stack4Things REST API

Table 1 reports an extract of the Stack4Things IoTronic RESTful API with exploited HTTP methods, URLs, semantics, input parameters, and return types. We focus on API methods that relate to nodes, corresponding pins, services that can be accessed on the nodes, jobs that can be scheduled to send sensor readings to the Cloud, and injection of CEP statements with specific reactions.

API calls listed under the Nodes section provide a list of the nodes currently registered to the Cloud (NodeCollection JSON data type provided in the body of the response) and, if necessary, detailed information about each node (Node JSON data type). An example of NodeCollection JSON response is the following:

```
{
 "nodes": [

   "description"
    "links": [
     {
      "href": "http://s4t.org/v0.1/nodes/eaaca217-
          e7d8-47b4-bb41-3f99f20eed89",
      "rel": "self"
     },
     {
      "href": "http://s4t.org/nodes/eaaca217-e7d8-47
          b4-bb41-3f99f20eed89",
      "rel": "bookmark"
     }
    ],
    "uuid": "eaaca217-e7d8-47b4-bb41-3f99f20eed89"
  }
 ]
}
```

while the following is an example of Node JSON response:

```
{
 "created_at": "2000-01-01T12:00:00",
 "description": "Sample node",
 "extra": {},
 "links": [
  {
   "href": "http://s4t.org/v0.1/nodes/eaaca217-e7d8
       -47b4-bb41-3f99f20eed89",
   "rel": "self"
  },
  {
   "href": "http://s4t.org/nodes/eaaca217-e7d8-47b4-
       bb41-3f99f20eed89",
   "rel": "bookmark"
  }
 ],
 "updated_at": "2000-01-01T12:00:00",
 "uuid": "eaaca217-e7d8-47b4-bb41-3f99f20eed89"
}
```

API calls listed under the Pins section provide the interface to access pins on nodes. In particular, it is possible to retrieve a node layout in terms of pins and their modes and it is possible to set/unset modes on a pin. Finally, it is possible to set/read a value from a pin. The RESTful interface hides the complexity.

6 Use cases

In this section, we propose some specific use cases that have been implemented and tested in the Stack4Things middleware, highlighting the interactions between the latter and application developers/providers as main consumers of the Stack4Things services. We therefore do not

Table 1 The s4t IoTronic RESTful API

#	Method	URL	Semantics	Parameters	Return type
	Nodes				
1	GET	/v0.1/nodes	Retrieve a list of nodes.	–	NodeCollection
2	GET	/v0.1/nodes/{node_uuid}	Retrieve information about the given node.	node_uuid (uuid)	Node
	Pins				
3	GET	/v0.1/nodes/{node_uuid}/pins	Retrieve a list of pins on a node.	node_uuid (uuid)	NodeLayout
4	GET	/v0.1/nodes/{node_uuid}/pins/{pin_name}/mode	Retrieve the mode of a pin on a node.	node_uuid (uuid) pin_name (unicode)	PinMode
5	POST	/v0.1/nodes/{node_uuid}/pins/{pin_name}/mode	Set the mode of a pin on a node.	node_uuid (uuid) pin_name (unicode) PinMode (json in body)	–
6	DELETE	/v0.1/nodes/{node_uuid}/pins/{pin_name}/mode	Clear mode setting for a pin on a node.	node_uuid (uuid) pin_name (unicode)	–
7	GET	/v0.1/nodes/{node_uuid}/pins/{pin_name}/value	Retrieve the value of a pin on a node.	node_uuid (uuid) pin_name (unicode)	PinValue
8	POST	/v0.1/nodes/{node_uuid}/pins/{pin_name}/value	Set the value of a pin on a node.	node_uuid (uuid) pin_name (unicode) PinValue (json in body)	–
Services					
9	GET	/v0.1/nodes/{node_uuid}/services	Retrieve a list of services on a node	node_uuid (uuid)	ServiceCollection
10	POST	/v0.1/nodes/{node_uuid}/services	Activate a new service on a node	node_uuid (uuid) Service (json in body)	–
11	DELETE	/v0.1/nodes/{node_uuid}/services	Delete a service on a node	node_uuid (uuid) Service (json in body)	–
Jobs					
13	GET	/v0.1/nodes/{node_uuid}/jobs	Retrieve a list of jobs on a node	node_uuid (uuid)	JobCollection
14	POST	/v0.1/nodes/{node_uuid}/jobs	Activate a new job on a node	node_uuid (uuid) Job (json in body)	–
15	DELETE	/v0.1/nodes/{node_uuid}/jobs	Delete a job on a node	node_uuid (uuid) Job (json in body)	–
CEP statements					
16	GET	/v0.1/cep	Retrieve a list of cep statements on the cloud	–	CepStatementWithReactionCollection
17	POST	/v0.1/cep	Activate a new cep statements on the cloud	CepStatementWithReaction	–
18	DELETE	/v0.1/cep	Delete a cep statements on the cloud	CepStatementWithReaction	–

investigate on application level use cases since the interaction of end users with such services strongly depends on the application logic, falling out of the scope of our work on the Stack4Things-app developers and providers interface. We therefore show how the architecture components interact among themselves to fulfill the specific goals. Each use case corresponds to a specific call in the Stack4Things REST APIs from potential app developers and providers.

6.1 Use case: provide the list of nodes registered to the cloud

This use case is the basic one, consisting in a listing of all the nodes currently registered to the Cloud. It corresponds to call #1 in Table 1. The listing is useful to retrieve the unique identifiers of the different nodes that can later be used to directly interact which each of them. As a use case

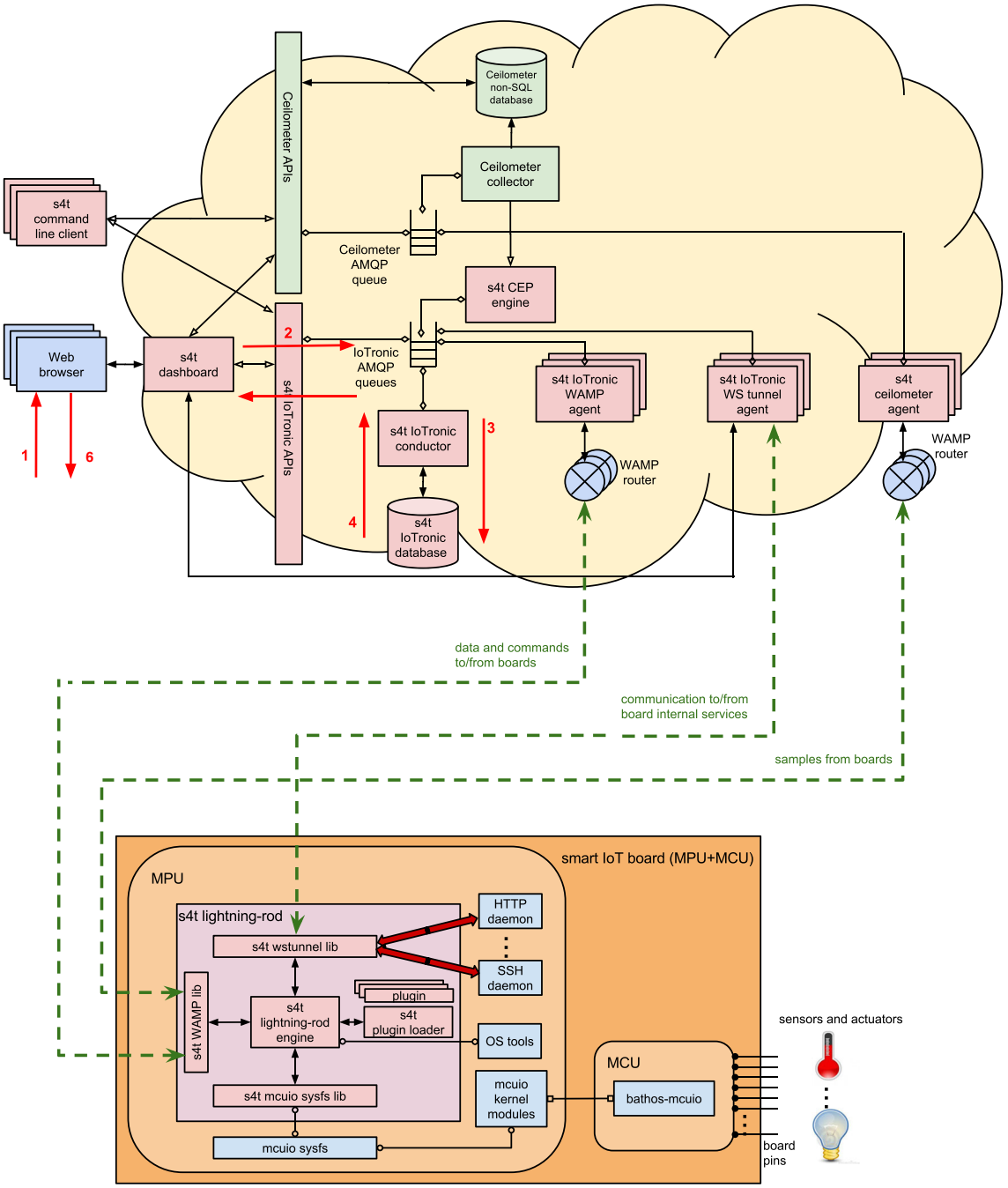


Fig. 7 Provisioning of the list of nodes registered to the cloud

prerequisite, we assume that one or more nodes are already registered to the Cloud. The following operations are then performed (see Fig. 7).

1. The user asks for the list of the nodes registered to the Cloud through the s4t dashboard (or alternatively through the s4t command line client).
2. The s4t dashboard performs one of the available s4t IoTronic APIs calls via REST (specifically, call #1). The call pushes a new message into a specific AMQP IoTronic queue.
3. The s4t IoTronic conductor pulls the message from the AMQP IoTronic queue and it performs a query to the s4t IoTronic database retrieving the list of nodes registered to the Cloud.
4. The s4t IoTronic conductor pushes a new message into a specific AMQP IoTronic queue.
5. The s4t IoTronic APIs call pulls the message from the AMQP IoTronic queue and replies to the s4t dashboard.
6. The s4t dashboard provides the user with the list of boards registered to the Cloud.

In this use case, no interaction with any board is necessary given that all the information is stored on the s4t IoTronic database. However, if desired, the connectivity status of the boards that are currently registered to the Cloud can be retrieved from the WAMP router using the presence mechanisms that it provides natively. In particular, such an information can be collected on demand or periodically. In the first case, when the call for retrieving the list of nodes is issued, after querying the s4t IoTronic database, the s4t IoTronic conductor can contact the s4t IoTronic WAMP agents to which the boards are registered obtaining news about their connectivity status. In the second case, the s4t IoTronic WAMP agents can periodically store such an information on the s4t IoTronic database in a proactive way so that the s4t IoTronic conductor is able to find it when necessary. Of course, a tread-off between freshness of the information (first case) and performance of the call (second case) arises.

6.2 Use case: retrieve the current value of a pin on a specific board

This use case is slightly more complex than the first one as it requires an interaction with a specific board. It corresponds to call #7 in Table 1. As a use case prerequisite, we assume that one or more nodes are already registered to the Cloud and that the user knows both the unique identifier of the desired board (maybe retrieved by issuing call #1 in Table 1) and the name of the pin from which he/she wants to read the current value (maybe retrieved by issuing call #3 in Table 1 to get the list of pins of a specific node). The following operations are then performed (see Fig. 8).

1. The user asks for the current value of a pin on a specific board through the s4t dashboard (or alternatively through the s4t command line client).
2. The s4t dashboard performs one of the available s4t IoTronic APIs calls via REST (specifically, call #7). The call pushes a new message into a specific AMQP IoTronic queue.
3. 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 if the required pin actually exists. Finally, it queries for the s4t IoTronic WAMP agent to which the board is registered.
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 uses the s4t mcuio sysfs lib to read the value of the specified pin and through the s4t WAMP lib replies to the s4t IoTronic WAMP agent by writing a message into a specific topic on the corresponding WAMP router.
8. The s4t IoTronic WAMP agent receives the message from the WAMP router and publishes a new message into a specific AMQP IoTronic queue with the value that has been read from the pin on the specified board.
9. The s4t IoTronic APIs call pulls the message from the AMQP IoTronic queue and replies to the s4t dashboard.
10. The s4t dashboard provides the user with the value that has been read from the pin on the specified board.

As already mentioned, scalability in this kind of use cases is assured by instantiating more than one s4t IoTronic WAMP agent so that each one of them can deal with a subset of the boards connected to the Cloud infrastructure.

6.3 Use case: create an SSH connection toward a node

This use case 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. The use case corresponds to call #10 in Table 1 that the user has to issue specifying the standard SSH port, i.e., port number 22. Given the assumption that nodes are behind a firewall/NAT,

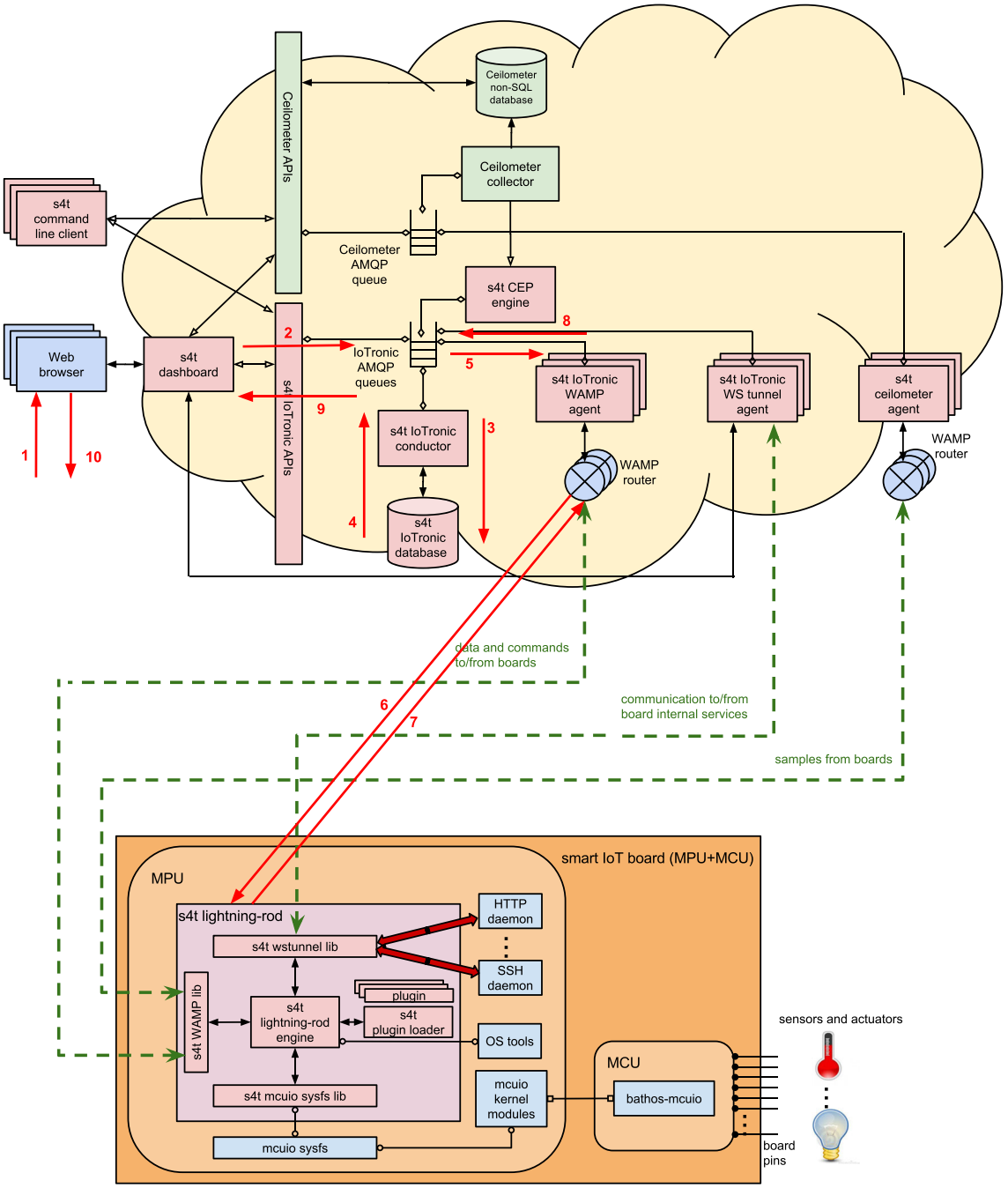


Fig. 8 Retrieving the current value of a pin on a specific board

a more complex interaction flow is needed in order to fulfill the goal with respect to a standard Cloud scenario in which compute nodes hosting virtual machines and frontend nodes through which the connection is guaranteed are usually within the same local area network. In fact, in order for a connection to be created with a service hosted on a board, a board-initiated tunnel needs to be created to the Cloud. As a use case prerequisite, we assume the node of interest is already registered to the Cloud and the SSH daemon is already listening on its standard port.[3] The following operations are then performed (see Fig. 9).

[3]Note that start/stop/restart commands for standard services running on the boards registered to the Cloud have been implemented through the RPC functionalities provided by the WAMP mechanisms but we do note report the corresponding execution flows for a matter of space.

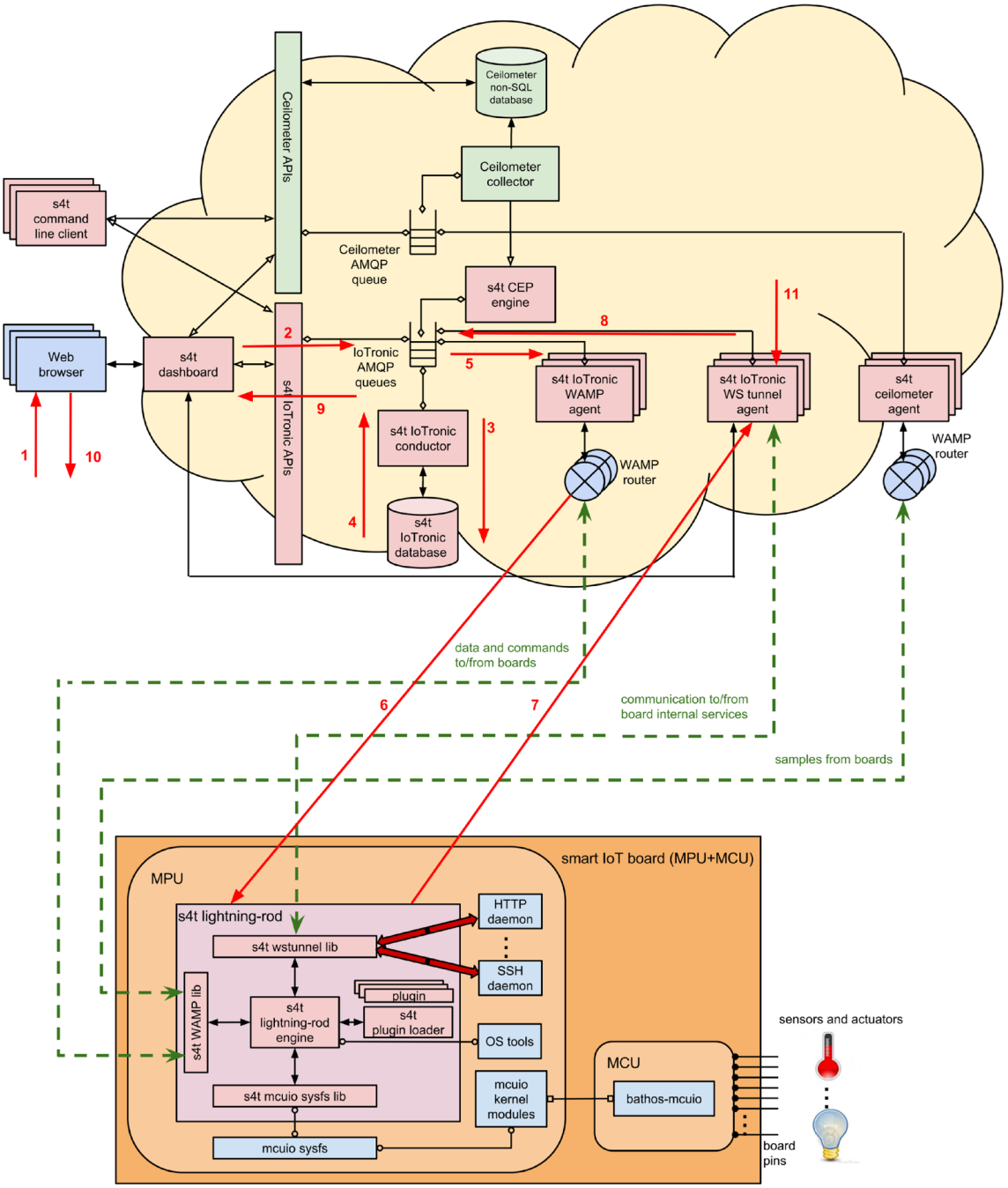


Fig. 9 Creation of an SSH connection toward a node

1. 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).
2. 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.
3. 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 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.
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 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.
8. 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.
9. The s4t IoTronic APIs call pulls the message from the AMQP IoTronic queue and replies to the s4t dashboard.
10. 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.
11. The user connects to the specified IP address and TCP port via an SSH client and the connection is tunneled to the board.

6.4 Use case: store readings from a sensor in the cloud

This use case envisions the use of the Cloud infrastructure not only for the management and control of the nodes but also as a storage platform for sensing data. In fact, the readings coming from a specific sensor attached to a specific board can be stored in the Ceilometer database and could be potentially used to take decisions and react to specific situations (see the next use case). The use case corresponds to call #14 in Table 1. Other storage systems could be used different from the Ceilometer one. For example, we also implemented mechanisms for the storing of sensing data in external MongoDB databases and or Open Data-compliant CKAN-enabled storages such as the one provided by the FIWARE infrastructure [19]. As a use case prerequisite, we suppose a sensor is attached to a specific pin of a specific board. We also suppose the board is already registered to the Cloud (Fig. 10).

1. The user asks for the readings from a sensor attached to a specific pin of a specific board to be stored in the Cloud with a specific sampling time[4] through the s4t dashboard (or alternatively through the s4t command line client).
2. 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.
3. 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 the board is registered. Finally, it decides the s4t ceilometer agent to which the readings from the sensor should be sent.
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 pushes 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 connects to the WAMP router of the specified s4t ceilometer agent through the s4t WAMP lib. Then, the s4t lightning-rod engine periodically reads from the specified pin through the s4t mcuio sysfs lib and pushes a message into a specific topic on the WAMP router.
8. The s4t ceilometer agent pulls each message from the WAMP router and pushes a corresponding message into the Ceilometer AMQP queue.
9. The Ceilometer collector collects the messages from the Ceilometer AMQP queue and stores the contained metrics into the Ceilometer non-SQL database. It also sends them to the s4t CEP engine via REST for further semi real-time analysis.

Besides periodic sampling of the readings of a sensor, the user is also allowed to program the system to send samples when specific situations are detected, e.g., a threshold is passed, a positive/negative change in the value occurs. Mixed data dispatching modes are also available, e.g., allowing users to program the system to send samples periodically and each time a threshold is passed.

6.5 Use case: inject a CEP rule and set a reaction

This use case shows how real-time analysis capabilities can be injected in the Cloud to react to specific situation of interest. In fact, the readings coming from all the sensors attached to the boards registered to the Cloud that have been configured to be stored in the Ceilometer database can be also redirected to the CEP engine that can be programmed to detect user-specified data patterns. The user can specify both the pattern of interest (in the form of a ESPER

[4]Other mechanisms have been implemented in which the sending of the sensor readings is triggered by a change or by the overpassing of a specified threshold but we do not report them here for a matter of space.

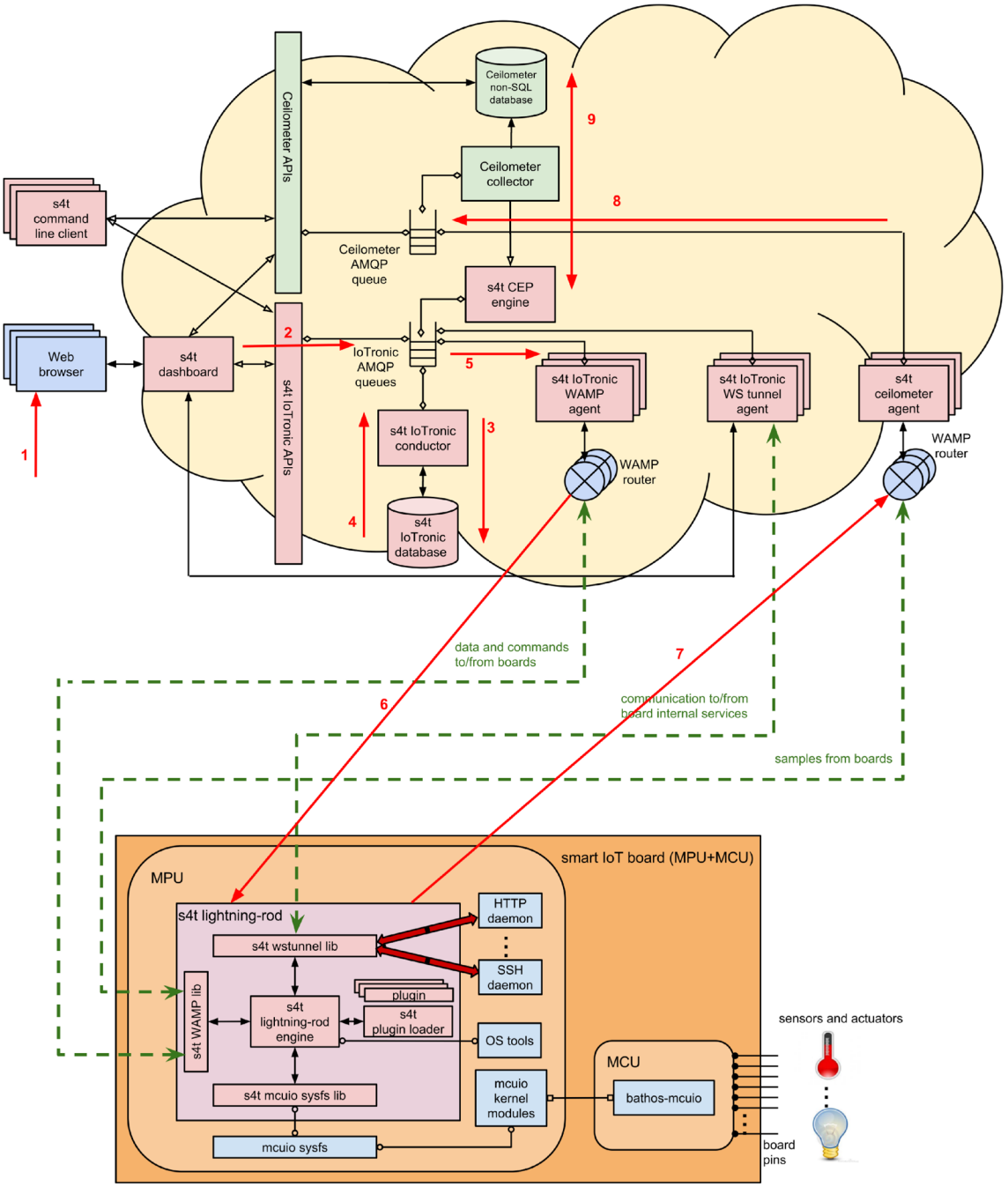


Fig. 10 Storing readings from a sensor in the cloud

statement) and the reaction that he/she desires the system to have as soon as a detection is performed. We designed the system so that reactions can be in the form of REST calls to the IoTronic interface so that, e.g., actuation commands can be send to the pins of specific boards, the configuration of the system can be changed injecting CEP rules, and so on. The use case corresponds to call #17 in Table 1. As a use case prerequisite, we suppose that a set of metrics have already be programmed so that they are sent to the Cloud (Fig. 11).

1. The user asks a CEP rule to be injected in the system together with a corresponding reaction through the s4t dashboard (or alternatively through the s4t command line client).
2. 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.
3. The s4t IoTronic conductor pulls the message from the AMQP IoTronic queue and it stores in the s4t IoTronic database the reaction that it founds in the

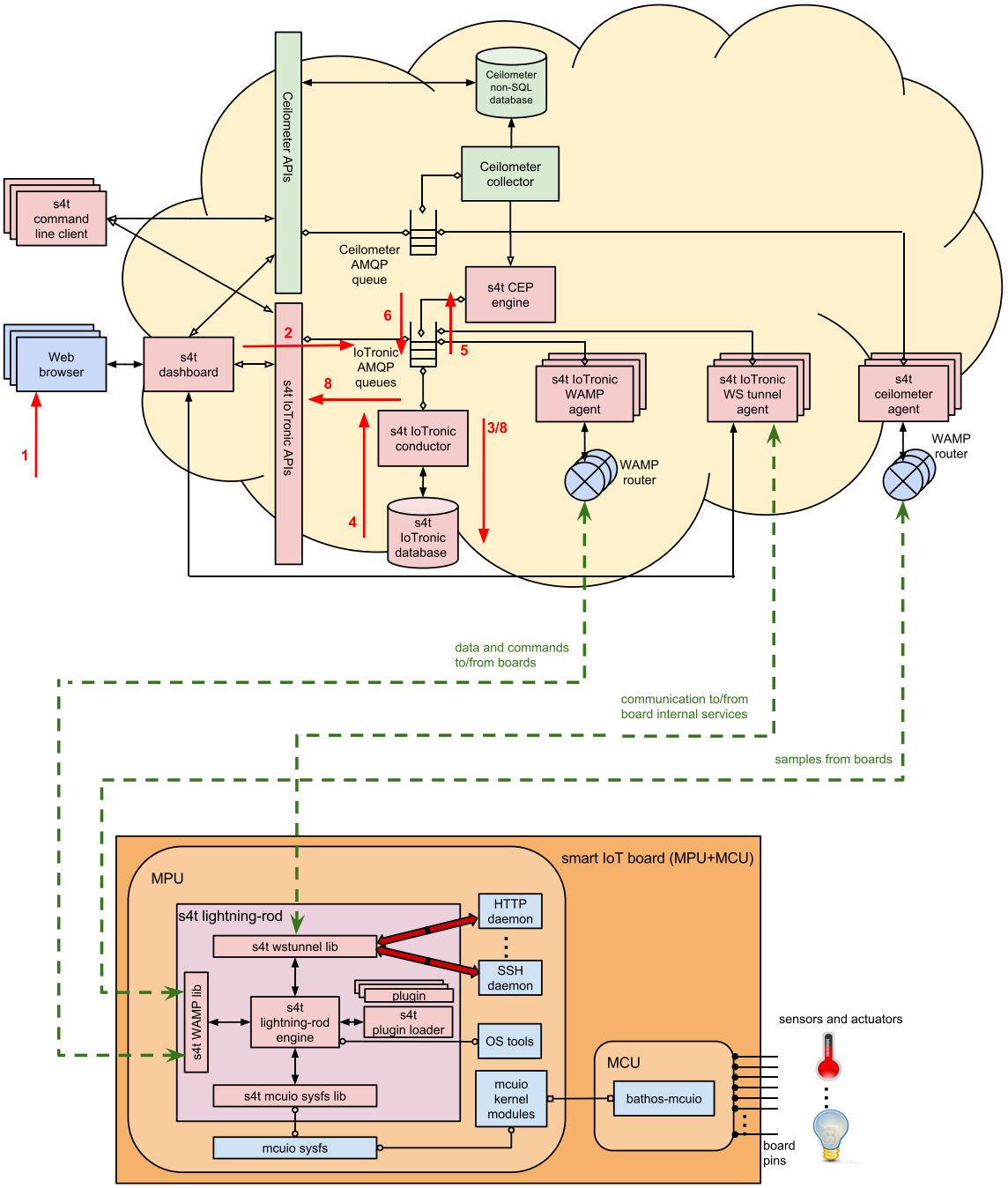


Fig. 11 Inject of a CEP rule and the corresponding reaction

message so that it will be able to retrieve it if the s4t CEP engine signals that the situation of interest occurs.

4. The s4t IoTronic conductor pushes a new message into a specific AMQP IoTronic queue.
5. The s4t CEP engine pulls the message from the queue and dynamically load the CEP rule so that it can be continuously checked.
6. If a CEP rule is triggered the s4t CEP engine pushes a new message into a specific AMQP IoTronic queue.
7. The s4t IoTronic conductor pulls the message from the queue and it queries the database for the reaction that has been associated to that rule.
8. The s4t IoTronic conductor issues the call specified in the reaction to the s4t IoTronic API actuating the reaction.

7 Discussion and conclusions

In this paper, we proposed Stack4Things: an implementation of the SAaaS vision through the OpenStack framework. We described the considered scenario highlighting actors and entities as well as identifying the main requirements. The Stack4Things architecture has been illustrated also discussing enabling technologies and potential use cases for application developers and providers.

This effort can be framed into the IoT scenario as a new way of exploiting this paradigm toward an utility approach. In the new global environment indeed, the ability to deliver sustainable high-value services is becoming ever more critical to increase competitiveness. This process is currently challenged by two significant trends. On the one hand, vendors have demonstrated strong competence in developing IoT and especially Cloud technologies and services (e.g., Google and Amazon), which are gaining an increasingly stronger acceptance in the global marketplace. On the other hand, the globalization of skills has enabled companies in emerging economies to develop advanced ICT capabilities as well, leveraging low-cost centers of software competence. Despite the work done so far, there is still a great need to focus on tackling the many challenges related to IoT, sensors, and objects involvement in the next generation software systems. However, IoT is actually about a new ecosystem that cuts across vertical areas. New business models will require the creation of an IoT ecosystem that benefits from open platforms such as Stack4Things and interoperability and the other advanced features they provide.

From a business perspective, several potential benefits from the Stack4Things adoption could be envisioned. Among them, one of the most interesting is to establish an open IoT/sensing resource market. Through Stack4Things, anyone can sell or buy sensing, actuation, and smart object resources, from the single resource owners and the sensor network owners up to and including IaaS providers. Environmental-related services may be easily developed and new generation of applications strongly interacting with the surrounding environment may spring up as a result. A service developer can look for either physical or virtual resources of third parties in case it is not able to satisfy the incoming requests; otherwise, it can sell its resources to a Stack4Things-powered provider, or even give them for free. This establishes a new idea of pervasive service provisioning as well as new business models with several cooperating actors (e.g., infrastructure providers, sensor network owners, volunteer end-users, resource brokers). Increasing the SME competitiveness. A large part of the improvement will originate from newly opened avenues for small and medium businesses to join the service economy. Previously, the setup and deployment of complex commercial services had been largely outside the reach of SMEs and individual entrepreneurs due to skill, cost, and complexity considerations. The Stack4Things framework fits for powering cash-strapped startups in launching (and iterating over) innovative web-oriented services, like oft-cited geolocation-related ones, in next to no time, reducing the entry barriers to the service economy and enabling SMEs to offer new competitive services to their customers. Moreover, advanced revenue models may be inspired by the volunteer-enabled SAaaS approach, leading toward a dynamic market for smart objects.

With the Stack4Things capabilities added to the base repertoire of Web services, it is easy to envision configurations where commercial enterprises plan IoT-based infrastructure sized for their average demands, and outsource excess capacity requirements to service providers, i.e., Cloudbursting, thus offering risk mitigation for unforeseen spikes in demand. The new capabilities of Stack4Things will pave the way for new business models for ICT service providers—who would support the rising wave of both SME service providers and enterprises requiring occasional support for their ICT needs.

Furthermore, the Stack4Things technology enables seamless interoperability among physical (hardware) resources, virtualized resources, and IoT items by means of customization, deployment, and, as a consequence, communication bridging facilities. We expect Stack4Things to greatly improve the ability of Cloud-based applications to take advantage of IoT-originated data, thus offering new business opportunities to IoT-powered service providers. Considering that the number of intelligent, communicating devices on the network will outnumber "traditional computing" devices by almost two to one by 2016, these new IoT-related economic opportunities will be potentially of very high impact.

Lastly, we believe that our work will contribute to the adoption and widespread diffusion of new software design methodologies, eased by tailored APIs and customization features, paving the way toward the proliferation of services as a fundamental workflow element of all business and government activities such as telecommunication systems, energy and other utility industries, healthcare, travel, entertainment, and more. As a growing number of functions will be delivered through services, the software industry will advance into an era of "Everything as a Service" approaches.

Future work on Stack4Things will be therefore devoted to improve the technologies to make real this outlook, by extending and integrating other OpenStack services (e.g., Neutron) with SAaaS functionalities 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].[5] 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.

Acknowledgment This research effort was supported by the EU 7th Framework Programme under Grant Agreement n. 610802 for the "CloudWave" project and by the EU Horizon 2020 Research and Innovation Program under the Grant Agreement n. 644048 for the "BEACON" project, and has been carried out in the framework of the CINI Smart Cities National Lab.

References

1. Gubbi J, Buyya R, Marusic S, Palaniswami M (2013) Internet of things (iot): a vision, architectural elements, and future directions. Futur Gener Comput Syst 29(7):1645–1660
2. Distefano S, Merlino G, Puliafito A (2012) Sensing and actuation as a service: a new development for clouds. In: 11th IEEE international symposium on network computing and applications (NCA), 2012, pp 272–275
3. Sanchez L, Galache J, Gutierrez V, Hernandez J, Bernat J, Gluhak A, Garcia T (2011) Smartsantander: the meeting point between future internet research and experimentation and the smart cities. In: Future network mobile summit (FutureNetw), vol 2011, pp 1–8
4. OpenStack documentation, http://docs.openstack.org
5. Stack4Things source code, https://github.com/MDSLab
6. Merlino G, Bruneo D, Distefano S, Longo F, Puliafito A (2014) Stack4things: integrating IoT with openstack in a smart city context. In: Proceedings of the IEEE 1st international workshop on sensors and smart cities
7. WebSocket, https://tools.ietf.org/html/rfc6455
8. WAMP, http://wamp.ws
9. Emeakaroha VC, Cafferkey N, Healy P, Morrison JP (2015) A cloud-based iot data gathering and processing platform. In: 3rd international conference on future internet of things and cloud (FiCloud), 2015, pp 50–57
10. Forsström S, Kardeby V, Österberg P, Jennehag U (2014) Challenges when realizing a fully distributed internet-of-things-how we created the sensiblethings platform. In: The 9th international conference on digital telecommunications ICDT, 2014, pp 13–18
11. Truong HL, Dustdar S (2015) Principles for engineering IoT cloud systems. IEEE Cloud Comput 2(2):68–76
12. Taherkordi A, Eliassen F (2014) Toward independent in-cloud evolution of cyber-physical systems. In: 2014 IEEE international conference on cyber-physical systems, networks, and applications (CPSNA), pp 19–24
13. Distefano S, Merlino G, Puliafito A (2015) Device-centric sensing: an alternative to data-centric approaches. IEEE Syst J PP(99):1–11
14. Distefano S, Merlino G, Puliafito A (2013) Toward the cloud of things sensing and actuation as a service, a key enabler for a new cloud paradigm. In: Xhafa F, Barolli L, Nace D, Venticinque S, Bui A (eds) 8th international conference on P2P, parallel, grid, cloud and internet computing, 3PGCIC 2013. IEEE, Compiegne, pp 60–67. [Online]. Available: doi:10.1109/3PGCIC.2013.16
15. Arduino, http://www.arduino.org
16. BaTHOS, https://github.com/ciminaghi/bathos-mcuio
17. Fazio M, Merlino G, Bruneo D, Puliafito A (2013) An architecture for runtime customization of smart devices. In: 12th IEEE international symposium on network computing and applications (NCA), 2013. IEEE, pp 157–164
18. AllJoyn, http://allseenalliance.org
19. FIWARE platform https://www.fiware

[5] 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.