Virtual IoT Systems: Boosting IoT Innovation by Decoupling Things Providers and Applications Developers

Andrea Detti*, Giuseppe Tropea*, Giulio Rossi*, Juan A. Martinez†, Antonio F. Skarmeta†, and Hidenori Nakazato‡
*CNIT: Electronic Engineering Dept., University of Rome "Tor Vergata", Italy
†Odin Solutions S.L, Alcantarilla, Spain
‡ Waseda Research Institute for Science and Engineering, Waseda University, Tokyo, Japan

***Abstract*—This paper proposes the VirIoT platform that enables virtualization of IoT systems, formed by virtual things and brokers. Our goal is to decouple developers of IoT applications from providers of things. VirIoT allows owners of IoT infrastructures to share them with many IoT application developers, which can simply rent the virtual things and the brokers their applications need. VirIoT can be useful for small stakeholders whose applications require large-scale IoT infrastructures, who are nevertheless unable to handle the infrastructure deployment. VirIoT can also be useful for owners of IoT infrastructures, in order to create isolated development environments where to run experimental services, before final deployment in the production system.**

***Keywords*-IoT; virtualization; cloud**

I. Introduction

Nowadays, most of the real-world IoT solutions operate within isolated silos containing both the infrastructure and the full-stack software. For small stakeholders, the infrastructure provisioning might be an insurmountable barrier that prevents their entering in the IoT arena, even though they might have innovative ideas. For instance, let us consider use cases for smart lighting or crime prevention systems in a big city. Tens of thousands of presence sensors, cameras and intelligent light bulbs are necessary, with a very high initial capital expenditure. Such high costs would be affordable to a small number of large corporations only, thus preventing fair competition and, even worse, slowing down the innovation pace, which instead is fast when thousands of small stakeholders take the field.

For almost all of today's applications running in production environments, Infrastructure-as-a-Service solutions have provided a convenient and widely adopted approach for renting the needed computing resources. Tenants can just focus on their applications, because the infrastructure, formed by computing, storage and network resources, is offered as a service by a cloud provider.

In this paper, we present an IoT virtualisation platform, named VirIoT , which re-uses cloud concepts but adapts them to the IoT world. VirIoT provides IoT developers with virtual IoT systems, named *Virtual Silos*, which are isolated environments formed by *Virtual Things* and *IoT Brokers*. Virtual things appear to a tenant as dedicated sensors or actuators that expose their data through a configurable broker technology (e.g. oneM2M [1], FIWARE [2], and the likes). Just like a cloud offers virtual servers with configurable virtual hardware and operating system (OS), VirIoT offers Virtual IoT Systems with a configurable set of Virtual Things (the hardware) and a Broker (the OS).

VirIoT decouples IoT infrastructure providers from application developers, thus making possible: for the providers, to better use their IoT devices by sharing their data with different tenants, and, for the tenants, to configure the IoT infrastructure they need, quickly. Provider and tenant may also coincide, exploiting VirIoT for running experimental services within the private infrastructure in use every day, raising higher the security bar by running applications and their things inside isolated environments.

II. Related Work

A. IoT Cloud Services

IoT applications require sophisticated coordination across connected objects, multiple clouds and networks, and the mobile front-ends. This is a complex endeavour, and developers do not want to do it from scratch. Hence cloud services for IoT are quickly emerging to facilitate IoT development, supported by providers that range from hardware vendors (Intel IoT platform, Bosch IoT Cloud) to system integrators (IBM Watson IoT) to the known ICT giants (Google Cloud IoT, AWS IoT, Microsoft Azure IoT).

All of the above IoT cloud services operate on similar architectures. The basic idea is to invite the user to bring her own set of sensors and actuators to their architecture, and they offer many functions on top, ranging from analytics to simplified device integration, from automated dashboards to improved security, from the scalability of billions of sensors/messages to flexible deployments. Accordingly, specific SDKs are provided to support application development.

For instance, connecting a RaspberryPi-based device to the AWS IoT cloud is a matter of generating a pair of security keys through the graphical console, then registering the device in the same console and deploying the C SDK on the Raspberry, which then securely connects via MQTT to the AWS cloud. A Thing Shadow (the cloud counterpart of

978-1-7281-2171-0/19/$31.00

the device) is then available for UPDATE, GET or DELETE methods, via both MQTT or RESTful APIs.

Though the above providers have similar levels of functionality and enterprise reliability, some peculiarities are worth noticing. For instance, AWS offers a customised version of FreeRTOS for incorporating low-power devices such as small microcontrollers within the AWS IoT ecosystem. Google IoT, on the other hand, has a major focus on machine learning and makes possible running TensorFlowLite over Linux and AndroidThinghs based edge devices/gateways.

While the platforms mentioned above mainly offer cloud services to IoT devices of customers, our VirIoT is instead focused on offering *things as-a-service*, by acquiring (control of) an ever-growing number of devices out there in the field, and by virtualising them to supply a scalable layer of horizontally share-able IoT resources to customers. Moreover, the virtual things rented by a customer can be, in turn, connected to upstream cloud service platforms as if they were real, un-shared, IoT devices. In this sense, the VirIoT services are complementary to most of the existing solutions and can interoperate with them in an extended IoT chain (see bottom-right of fig. 4).

B. IoT Brokers

IoT information is collected and distributed by a specific component usually named as *Broker*. The design and development of the broker component are focused on efficiently managing a plethora of IoT use-cases by employing request-response and publish/subscribe messaging patterns and exposing a public API based on open and standard protocols. An IoT broker stores information according to a specific data-model and exposes a secure API for publishing, fetching and discovering IoT data items, devices, and the likes. Additionally, by using a distributed approach, many brokers can be interconnected to scale out the system. E.g., an IoT platform can comprise set of "edge" brokers connected to a core broker.

Two different IoT platforms whose brokers that have gained much interest by both industry and academia are oneM2M [1], [3] and FIWARE [2].

1) oneM2M: The oneM2M platform [3] represents IoT resources in a hierarchy whose main entities are Application Entities (AEs), Containers and Content Instances (the actual data items). Every IoT device or IoT application is associated with an AE, which contains Containers that store Content Instances, i.e., the actual IoT data items. For instance, a sensor can be a source of content instances; an actuator can be a consumer of content instances which represent its status (e.g., on/off); an application logic can fetch Content Instances from different Containers, make some reasoning on top of them and publish a new state information in a Container where the actuator is registered to. An AE can also include semantic information about the contained data.

Applications interact with the platform through a single or a hierarchy of Common Service Entity (CSE) whose API supports data publishing, authentication, information discovering and subscriptions to name a few. HTTP and MQTT are used as transport protocols. Currently, many CSE implementations exist including Mobius [4], OpenMTC, Eclipse OM2M, etc. Actually, the CSE server can be considered as the oneM2M broker.

2) FIWARE: FIWARE [5] is an initiative funded by the European Commission aiming at making easier the development of smart applications by promoting the use of a global catalogue for sharing ready-made platform components, called Generic Enablers (GEs).

One of the most significant platform components is the Context Broker, the entity responsible for the distribution of the information. So far, there are two implementations: Orion Context Broker and Aeron IoT Broker which provide a publish/subscribe messaging pattern, as well, as a method to query the stored context information. They adopted Next Generation Service Interfaces (NGSI) REST API, a technology standardised at Open Mobile Alliance (OMA) [6], [7]. Additionally, thanks to the work of ETSI ISG CIM workgroup [8], NGSI has evolved into NGSI-LD (based on JSON-LD) allowing for a richer representation of information. FiWARE uses a component called IDAS, which is a Backend for Device Management. This component makes uses of IoT Agents for translating the information coming from IoT lightweight protocols such as MQTT or CoAP, among others to the NGSI representation.

III. Concepts and Use Cases

Fig. 1 shows the main concepts behind the VirIoT platform. On the left we have many *IoT Systems*, where an IoT System is made of a network of real things (sensors, actuators, etc.) exposing information through an IoT platform, such as FIWARE Orion or Mobius [4] oneM2M. Hence, an IoT System is formed by a collection of real things and by the platform that manages them.

Information coming from different IoT Systems, and possibly from other data sources (e.g. open data), forms a *Root Data Domain*, from which VirIoT gathers information. Specifically, a group of VirIoT components named *ThingVisors* fetch the information and generate data items associated with *Virtual Things*. Consequently, a Virtual Thing is an emulation of a real thing that produces data obtained by processing/controlling data coming from the root data domain.

Fig. 2 presents a diagram where we have emulated four virtual things (right side of the figure) from three real things (left side of the figure). The three real things are a stationary camera, a camera-equipped drone and a thermometer, whereas there four virtual things are a face detector, a person counter, a moving camera and a thermometer.

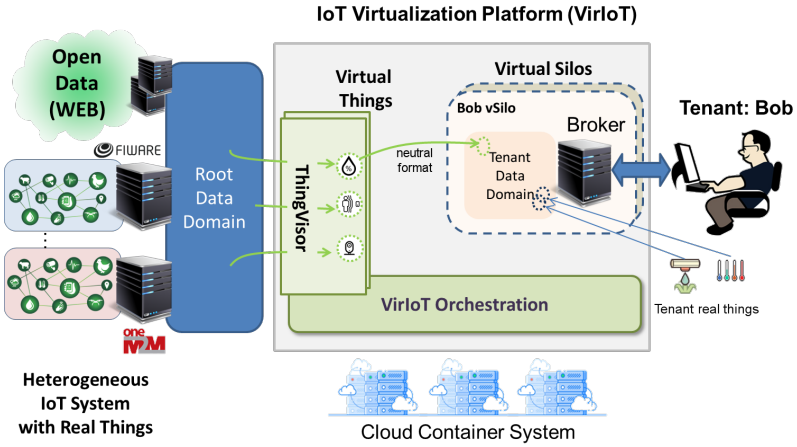

Figure 1: VirIoT Platform

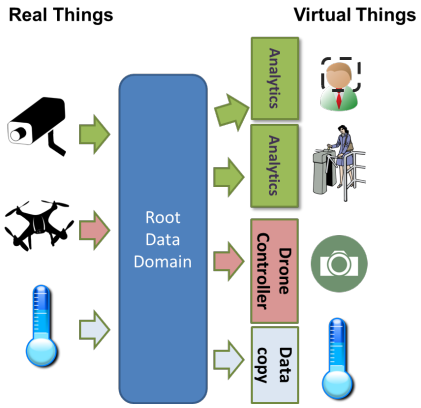

Figure 2: Virtual Things (vThings)

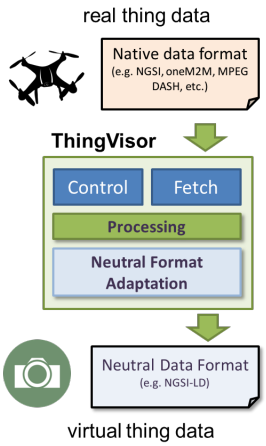

Figure 3: ThingVisor

The things' virtualisation concept that we are considering in this paper may go beyond traditional data processing since it can also involve "control" of the real things. Let us explain this concept by detailing the virtualisation process made to obtain the virtual things presented on the right side of the figure. The virtual face detector and virtual person counter obtain their data by performing analytics on the video stream coming from the real camera. The virtual thermometer obtains its data by merely copying data coming from the real thermometer. Finally, the virtual moving camera is a camera that takes pictures at a very slow rate (e.g. one frame per hour) which a user/tenant can relocate to one or more given positions, such as interesting hot spots of a harbour in need of statistics or surveillance. In this last example, virtualisation is achieved by controlling the path of a drone to periodically drive it over the locations chosen by tenants and taking a picture.

Fig. 3 depicts a general schema of how a ThingVisor receives data coming from one or more real things (or other sources, e.g. the web) and process it in its native format to produce new data items of the virtual thing. These items are produced in a *neutral* data format that can be translated to the data formats in use by different brokers.

The VirIoT platform provides tenants with virtual IoT systems, dubbed *Virtual Silos*, which are isolated environments dedicated to a specific tenant for running his applications. A tenant can add data coming from the platform's virtual things to his virtual silo. Besides, he can also connect his real things to his virtual silo. Collectively, such data comprises the tenant data domain which is exposed to the external world through a broker technology of choice. For instance, let us assume that Bob is a tenant who wants to develop a watering system for his house, and he is familiar with the FIWARE Orion Broker. Bob can create a virtual silo embedding such a broker, connect his own thermometers and watering devices (actuators) to the broker, and he can "rent" a virtual hydrometer for measuring air humidity outside his house, just because he does not own a real one. Data from the rented hydrometer reaches Bob's broker in the silo, together with data from his sensors. So, Bob only sees his dedicated data set and broker, by accessing his silo, and the platform thereby provides for data and service isolation.

Similarly to cloud computing, we envisage two possible high-level usage scenarios of the VirIoT platform, *public* and *private*. In the public scenario, we foresee three distinct types of stakeholders: i) providers of real IoT systems offering their data in different formats, ii) providers of one or more competing for VirIoT platforms, which use this data to set up virtual things, iii) IoT application developers renting IoT virtual silos. This use case is going to be crucial in large-scale environments, such as smart cities, whereby the city owns several arrays of sensors and sells the raw data streams to a VirIoT provider, which acts as the intermediary between the vast amount of raw resources and the applications. Designers of smart city applications can instantiate silos as a service in the platform, gaining access to perhaps thousands of selected virtual things and

a brokering environment of choice, without caring about infrastructural and data heterogeneity problems. Regarding data heterogeneity, we note that many IoT platforms cope with it by transforming external heterogeneous data items into an internal format (e.g. through proxies) and then by exposing that format to the final user, through a specific API. VirIoT makes a step forward: the data model and the API are a choice of the user rather than a platform one.

In the private scenario, the same actor owns both the infrastructure and the applications. She can use VirIoT both to enclose each IoT application in a small isolated environment (i.e. a virtual silo) and to support safe innovation, by decoupling the newly designed IoT applications from IoT services that are already in production. For instance, a company operating a smart harbour system may have a robust solution in place, where the existing application exploits various real sensors through a production broker. A novel version or an enhancement can be safely tested in a virtual silo, before final deployment in the production environment. Security-wise, a choice can be made as to what to expose to attacks from the outside. In short, a private approach to IoT virtualisation offers the same advantages a private cloud is nowadays offering to companies deploying their servers in virtual vs bare-metal.

IV. System Architecture

Fig. 4 shows a preliminary architecture for the VirIoT platform. This architecture follows a micro-services design; hence each component is an autonomous subsystem exposing network interfaces. Linux containers (e.g. Docker) have been considered as the preferred component packaging tools, possibly supported by a container orchestration tool such as Kubernetes (k8s).

For external communications, the platform exposes an HTTP REST interface for the administrator and the tenants. Internal communications use a topic-based pub/sub system whose topics are reported in table I. There are control and data topics. Control topics are used by components to receive (`c_in`) or send (`c_out`) control messages. Data topics are used to convey the data items of virtual things.

On the left of fig. 4 there are the ThingVisors, each uniquely identified by a name (`TViD`). A ThingVisor generates data items of one or more virtual things, and each virtual thing is uniquely identified by a name (`vThingID`). The architecture is agnostic to the technology used to develop a ThingVisor since it runs within an own container (or k8s pod). However, it is necessary that it communicates with the other components through vThing and ThingVisor topics (again, see table I).

On the right of fig. 4 there are virtual silos (vSilos), which are used by tenants: e.g. Bob, Hana, and Lucas. Each silo is identified by a unique name (`vSiloID`). There could be different types of virtual silos, which differ in terms of broker type, scaling property, storage model, etc. We call *flavour* a specific configuration of a virtual silo, therefore a virtual silo is an instance of a given flavour. In figure 4, Bob and Hana have their own virtual silos (#a and #b, respectively) whose flavours are the same and include a oneM2M broker to which their applications connect. Lucas uses a virtual silo of a different flavour, instead, which exports the IoT data of his virtual/real things via simple MQTT topics. This is a kind of *raw* virtual silo, which can be in turn connected to an upstream IoT platform such as Node-Red or Google/Azure/Amazon IoT cloud services, according to the application design and deployment strategies.

Each virtual silo includes an internal controller that is used to configure it (e.g. for adding or removing instances of virtual things) and also to relay the data items of selected virtual things from the ThingVisor to the silo's broker. Again, the architecture is agnostic to the technology used to develop a virtual silo, as each silo runs in an own container (or k8s pod).

The master-controller manages the deployment of new components in the systems as well as their configuration, following requests coming from administrator and tenants. System state information, about virtual silos, virtual things, ThingVisor, etc. is stored in a System DB. Specific object stores maintain container images of silo flavours and ThingVisors.

We now describe some basic procedures. The administrator can request to add a new silo flavour or ThingVisor, in order to extend the VirIoT platform's capabilities. Consequently, the master-controller inserts the new container image of the silo/ThingVisor into the proper stores and updates the System DB. In the case of ThingVisors, it is then the underlying container platform that runs instances of them. As soon as it is up and running, a ThingVisor starts to publish data items of the virtual things it handles to the related data topics.

When a tenant requests creation of a virtual silo, the master-controller fetches and runs an instance of the image of the requested flavour, providing the tenant with an IP address and port where she can contact the broker running inside the virtual silo. Subsequently, a tenant can request virtual things to be added to her virtual silo. The master-controller, in turn, relays this request to the virtual silo controller through its input control topic. Consequently, the silo controller registers the necessary metadata in the silo's broker and becomes a subscriber of the virtual thing data topic, thereby starting to receive related data items. These data items are translated from the neutral format to the data model used by the silo's broker, and then they are eventually pushed to the broker. Moreover, the master-controller is the one that stores all configuration information of the virtual silos in the System DB.

We have made a preliminary open-source implementation, which uses Docker as a container manager. We devised three silo flavours, offering Mobius oneM2M [4],

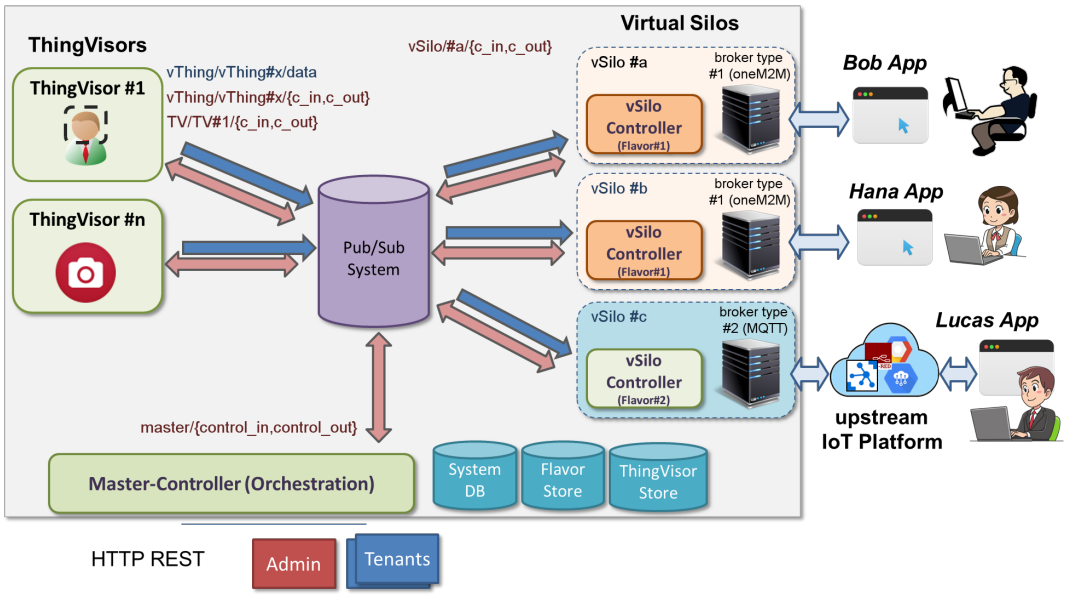

Figure 4: System Architecture

Table I: System Topics

Topic	Naming Scheme	Description
vThing (Data)	`vThing/<vThingID>/data`	Used by a ThingVisor to publish data items of a virtual thing.
vThing (Control)	`vThing/<vThingID>/{c_in,c_out}`	Used by a ThingVisor to send (`c_out`) and receive (`c_in`) control information related to an handled virtual thing (e.g. change data source, add face to match, motion threshold update, change virtual camera position, etc.)
ThingVisor (Control)	`TV/<TViD>/{c_in,c_out}`	Used by a ThingVisor to send (`c_out`) or receive (`c_in`) control messages related to the whole ThingVisor (e.g. pause, remove, activate vThing, etc.)
vSilo (Control)	`vSilo/<vSiloID>/{c_in,c_out}`	Used by the vSilo controller to send (`c_out`) or receive (`c_in`) control messages related to the specific virtual silo (e.g. add vThing, remove vThing, etc.)
Master (Control)	`master/{c_in,c_out}`	Used by the master-control to send (`c_out`) or receive (`c_in`) control messages related to the system configuration

FIWARE Orion and MQTT Mosquitto as possible brokers. We also developed some configurable ThingVisors: two of them merely fetch data items from FIWARE/oneM2M IoT domains and replay them to related virtual things topics; another ThingVisor fetches weather data of different cities from OpenWeather, parses it and publishes temperature, pressure and other weather information as data items of different virtual things. Regarding the neutral format, we used the latest NSGI-LD ETSI specification [8].

V. ThingVisor Design Technologies

A flexible IoT service development platform may undeniably help the implementation and deployment of ThingVisors. To this end, solutions providing service function chaining are valuable candidates. We can devise a ThingVisor as formed by an ordered set of tasks, composing in this way a service function chain where the last task is publishing produced data on VirIoT system topics. In what follows we present two platforms, FogFlow and ICN, that can be used as midleware between the Root Data Domain and ThingVisors.

A. *FogFlow*

FogFlow [9] is an IoT edge computing framework that, aimed at smart city environments, which uses an NGSI-centric programming model to allow users to develop and deploy IoT services over cloud and edges easily.

Each service is made by a set of tasks that receive and send ”flows” of data, either from IoT sources or from other tasks. FogFlow provides a graphical user interface for the definition of IoT tasks called *fog-functions*. The platform also offers a discovery function that makes it possible to create a federation among the NGSI IoT brokers of the Root Data Domain. A fog-function can request IoT data to the internal Orion Broker, and the system will relay this request to the proper IoT Broker of the Root Data Domain. Moreover, the federation may also be extended to non-NGSI brokers/devices living within the Root Data Domain, through a specific adapter. A FogFlow orchestrator takes care of deploying fog-functions to optimised locations that offer Cloud/Edge/Fog computing functionality (on the