Virtual Resources for the Internet of Things

Andrea Azzarà
Scuola Superiore Sant'Anna, Italy
a.azzara@sssup.it

Luca Mottola
Politecnico di Milano, Italy and SICS Swedish ICT
luca.mottola@polimi.it

Abstract*—We present VIRTUAL RESOURCES: a software architecture to resolve the tension between effective development and efficient operation of Internet of Things (IoT) applications. Emerging IoT architectures exhibit recurring traits: resource-limited sensors and actuators with RESTFUL interfaces at *one* end; full-fledged Cloud-hosted applications at the *opposite* end. The application logic resides entirely at the latter, creating performance issues such as excessive energy consumption and high latencies. To ameliorate these, VIRTUAL RESOURCES allows developers to push a slice of the application logic to intermediate IoT devices, creating a continuum between physical resources and Cloud-hosted applications. With VIRTUAL RESOURCES, for example, developers can push processing of sensed data to IoT devices close to the physical sensors, reducing the data to transmit and thus saving energy. We describe the key concepts of VIRTUAL RESOURCES and their realization in a CoAP prototype atop resource-constrained devices. Experimental results from cycle-accurate emulation indicate that VIRTUAL RESOURCES enable better performance than Cloud-centric architectures, while retaining the RESTFUL interaction pattern. For example, energy consumption in representative scenarios improves up to 40% and control loop latencies reduce up to 60%.

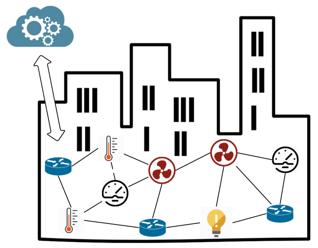
I. INTRODUCTION

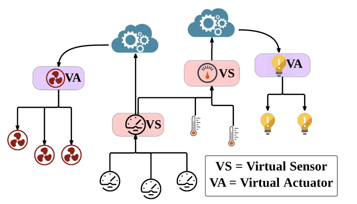
The Internet of Things is expected to play a key role in a range of domains, from factory automation to healthcare [1]. These applications are enabled by embedded sensors and actuators able to exchange data with the larger Internet.

Applications and devices. Building automation is a paradigmatic IoT scenario. The system's goal is to intelligently control electrical appliances such as lights and HVAC (Heating, Ventilation, and Air Conditioning) to reduce energy expenditures, while retaining the user comfort. To this end, applications gather data from sensors; for example, to measure the appliances' generated loads, and use these data to affect the environment by controlling the relevant actuators.

In most scenarios, sensed data are not used in raw form. Rather, they are typically processed through multiple stages until a higher-level information is obtained that is useful to take intelligent decisions [2]. For example, the appliances' generated loads are aggregated based on their location; either public spaces or private offices, as these demand different control strategies. The commands sent to actuators also depend on their characteristics. For example, some appliances are not critical to a building's operation, such as those providing accessory services to the inhabitants. These may be safely dimmed or switched off to save energy.

Adopting IoT technology for these applications can break the isolation of systems. For example, to perform load balancing, different buildings may be jointly controlled by matching their energy consumption against that of their neighborhood. Moreover, offering the ability to sense or to actuate through standard-compliant networks enables a better re-use of the


(a) Cloud-centric IoT. *The application logic resides entirely at the Cloud, creating performance issues.*



(b) Logical view of VIRTUAL RESOURCES applied to building automation. *Parts of the application logic execute inside the network, improving performance.*

deployed devices, which may be seamlessly accessed by multiple applications.

To lower costs and to ease installations, the devices typically run on batteries and communicate wirelessly. As a result, they can only feature 8- or 16-bit microcontrollers, few kB of RAM, and low-bandwidth radios [3]. Further, because of the limited communication range affordable with little energy, they often form *multi-hop* networks to ensure overall connectivity.

These observations do not apply solely to building automation. For example, smart-city applications such as intelligent lighting, traffic control, and structural monitoring exhibit similar traits [1]. In this context, for example, IoT technology may allow different public institutions to share the sensing infrastructure, offering a multitude of advanced services.

Current practice. Despite the lack of established solutions to architect IoT software, specific trends are distinctly emerging. One approach is the so called "Cloud-centric IoT" [4], depicted in Figure 1(a). Sensors and actuators expose application-agnostic elementary functionality through RESTFUL interfaces, whereas the application logic is entirely deployed in the Cloud. By doing so, applications benefit from virtually-unlimited computing resources, whereas developers enjoy the range of development tools and integrated services made available by Cloud providers such as Xively (xively.com), ThingSpeak (thingspeak.com), and OpenSense (open.sen.se).

The Cloud-centric architecture, however, also comes with disadvantages. For example, in building automation, the application logic is unlikely interested in the individual readings of each and every power meter. Rather, only the sum of those readings is relevant to determine the energy consumption. Using a Cloud-centric architecture, the application needs to probe every sensor one by one and then compute the sum at the Cloud. Similarly, the application may decide to dim the lights in public areas to reduce a building's energy consumption. Using a Cloud-centric architecture, this requires a separate call to every individual light controller. These functions likely increase the traffic within the resource-constrained network. As radio communication is energy-draining on IoT devices, this may severely impacts their lifetime. Network latencies grow as well, as data need to travel from the Cloud across a complex infrastructure before reaching the IoT devices in the field.

Contribution. To remedy these issues while retaining RESTFUL interactions, we present VIRTUAL RESOURCES: an IoT architecture that contrasts the separation between physical devices and Cloud-hosted applications. Using VIRTUAL RESOURCES, developers can relocate slices of the application to any intermediate IoT device, such as the message routers inside the wireless network. This occurs through the concepts of *virtual sensor* and *virtual actuator*, as in Figure 1(b).

Using a *virtual sensor*, developers can acquire data from a set of physical sensors, process the data according to a programmer-provided function, and export the results through a RESTFUL interface akin—or even identical—to a physical device. For example, developers may define a "total-energy-public-areas" sensor built to provide the sum of the readings of all power meters in public areas. They can then deploy the virtual sensor on any IoT device between the physical sensors and the Cloud. The Cloud-hosted application interacts only with the virtual sensor to obtain the data.

Similarly, using a *virtual actuator*, developers can offer a single entry point for the Cloud-hosted application to control the lights in public areas, through a RESTFUL interface similar to a physical actuator. Developers may define a "lights-public-areas" actuator to drive all lights in public areas at once, and deploy the virtual resource on any IoT device. The Cloud-hosted application performs a single call to the *virtual actuator* instead of a varying number of calls depending on how many light controllers are deployed in public areas.

Using VIRTUAL RESOURCES thus provides key benefits:

- Network resources are better utilized. The ability to push a slice of the application processing to intermediate IoT devices enables, for example, a reduction in the amount of data coming from sensors, which helps prolong the system's lifetime. Similarly, lower network traffic is less likely to generate congestion within the IoT network, improving on the application's perceived latency.
- The Cloud-hosted application becomes simpler. For example, driving one actuator for all lights in public areas is easier than a varying number of individual actuators. Notably, the set of resources representing inputs (outputs) for a *virtual sensor* (*actuator*) is evaluated only at run-time. Therefore, changes in such sets are dealt with transparently w.r.t. the Cloud-hosted application.
- Developers can separate out low-level concerns and move them to other devices, fostering a better separation of concerns. For example, the manufacturer the IoT devices may provide a library of virtual resources useful in paradigmatic scenarios, used by domain-experts when implementing the high-level application logic at the Cloud.

To make our contribution concrete, we create a prototype based on CoAP, targeting extremely resource-constrained devices. As the resulting performance is also a function of what node is chosen to run a virtual resource, our prototype includes custom heuristics to determine where to deploy a virtual resource in a generic multi-hop wireless network.

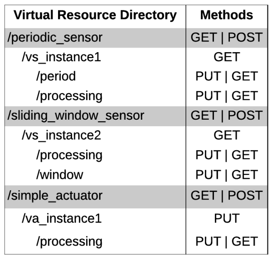
Prior art and road-map. Although service-oriented architectures are being widely applied to abstract the "things" [5], very few approaches enable to relocate slices of the application logic inside the IoT network. Efforts close to ours are, for example, those of Mitton et al. [6], who present a concept of sensor virtualization in a smart-city use case. However, they do not provide the ability of moving parts of the application logic inside the IoT network. Similar observations apply to the concept of "physical mashup" [7], and to systems such as IBM's Node-RED (nodered.org), where sensor virtualization components can indeed be created, which however execute on a machine outside the IoT network. We did present a notion of virtual sensor and virtual actuator for stand-alone wireless embedded networks in earlier work [8]. However, we did not tackle the problem of Internet integration while the design was entirely customized, and thus not standard-compliant.

The remainder of the paper unfolds as follows. Section II illustrates our design of VIRTUAL RESOURCES. Section III describes the concrete use of VIRTUAL RESOURCES through the programming support we provide. We illustrate the underlying implementation in Section IV. Our quantitative evaluation, reported in Section V, demonstrates performance improvements in energy consumption and application latencies.

II. DESIGN

We design the concrete realization of VIRTUAL RESOURCES in a way that manipulating its elements is accomplished using RESTFUL interactions as well.

Our design revolves around three concepts, hierarchically organized: *i) templates*, *ii) instances*, and *iii) configuration* resources. The templates abstractly specify the services offered by a virtual resource, fostering re-use. The instances are the operational counterpart of templates and are derived from them, that is, every instance is a sub-resource of the template it is derived from. A generic instance can be configured by means of configuration resources, located in the next level of the hierarchy. This hierarchical organization is reflected in a virtual resource *directory*, exemplified in Figure 1, whose goal is to provide an entry point for the creation, configuration, and use of virtual resources.



Virtual Resource Directory	Methods
/periodic_sensor	GET | POST
/vs_instance1	GET
/period	PUT | GET
/processing	PUT | GET
/sliding_window_sensor	GET | POST
/vs_instance2	GET
/processing	PUT | GET
/window	PUT | GET
/simple_actuator	GET | POST
/va_instance1	PUT
/processing	PUT | GET

Fig. 1. Virtual resource directory example.

Virtual resource templates. The templates specify three aspects: *i)* the resources of interest associated with the virtual one, for example, what actuators are the target output of a *virtual actuator*; *ii)* the operations offered by a virtual resource's interface and its configuration sub-resources; *iii)* the distributed behavior of the virtual resource, for example, whether a *virtual sensor* pulls data from the input resources or the latter periodically push data.

As for point *i)*, the input/output resources concurring to create a virtual one are defined by specifying a list of desired attributes to be found in the resources of interest. Such a list is taken as a Boolean constraint to be matched against application-level attributes of other resources. For example, a *virtual sensor* may be defined to consider as input all exiting resources whose `type` attribute equals `power_meter`, and whose `location` attribute matches `public_areas`.

The template also defines the configuration resources of all corresponding instances, as described in point *ii)*, which allow to tune their behavior. For example, a configuration resource may control the period at which a *virtual sensor* offers a new reading. Every instance always features at least one configuration resource that binds the *processing function* applied to input data or to output commands. For example, the processing function for a *virtual sensor* may perform a form of aggregation over the input data. For a *virtual actuator*, the processing function may perform the command translation necessary to interact with the output resources.

As for point *iii)*, the distributed behavior of the virtual resource corresponds to an underlying implementation of distributed functionality, typically carried out by a programmer with knowledge of communication protocols. The encoding of what interaction pattern to employ; for example, push vs. pull in the case of a *virtual sensor*, is also part of this implementation. Here again is an opportunity for better separation of concerns. In traditional architectures, these aspects are intertwined with the application's processing. VIRTUAL RESOURCES cleanly separate this from the high-level application logic, allowing different developers to take care of distinct functionality.

From templates to instances. At run-time, the templates are published on the *virtual resource directory*. This operation makes them available to create virtual resource instances.

The virtual resource directory of Figure 1 contains three templates, namely a *periodic-sensor*, a *sliding-window-sensor*, and a *simple-actuator*. Every template must support the GET and POST methods. The GET method returns a description of the template itself and of the interfaces of derived instances. This enables run-time discovery of the available virtual resources. The POST method performs the actual creation of the virtual instance, creating a new sub-resource of the template. The operation returns the URI of the new resource. The required configuration resources are automatically created as sub-resources of the instance.

Every template can host an arbitrary number of instance sub-resources. The instances offer the methods defined in the corresponding template. The example in Figure 1 includes three instances, `vs_instance1`, `vs_instance2`, and `va_instance1`, namely, two virtual sensors and one virtual actuator. Instances of virtual sensors generally support a GET method to return the (virtual) sensor reading, whereas instances of virtual actuators typically support a PUT method to update the target actuators. After the creation of an instance, the application can further configure its behavior by updating the configuration sub-resources, using PUT methods. Virtual resources also support a DELETE method to destroy an instance, which makes it disappear from the directory.

Once the instances are created and possibly configured, the application can use their services through their RESTFUL interfaces, exactly like if these were bound to physical devices. In doing so, the binding between a virtual resource and its input/output resources is *evaluated dynamically*, only when a service is requested on an instance. That means that the actual input or output set of a virtual resource may freely change at run-time; for example, as new nodes join or some fail because of battery depletion, and yet the main application keeps operating transparently to these dynamics.

Although we hitherto considered that the input/output data of virtual resources be physical devices, this is not a requisite. A virtual resource may use another virtual resource as input/output, creating hierarchies. This is as simple as creating a template that matches the attributes of other virtual resources. Moreover, virtual resources may act as RESTFUL proxies for legacy technology or non-IP networks, masking the system's heterogeneity. For example, a *virtual actuator* may output commands on an industry-strength wired bus [9] by means of a proper translation. This would additionally offer a standard-compliant means to integrate non-IoT networks.

```
1 metersPublic = resource_set(Type='power_meter', location='public')
2 VsPeriodic(Input=metersPublic, name='per_power_meter_public')
3 lightsPublic = resource_set(Type='light_control', location='public')
4 VaSimple(Output=lightsPublic, name='lights_control_public')
```

Fig. 2. Publishing virtual resource templates onto the directory.

```
1 metersTempl = Resource.get(name='per_power_meter_public')
2 vs = metersTempl.POST('vs')
3 vs.fun.PUT(sumCode)
4 vs.period.PUT('60')
5 lightsTempl = Resource.get(name='lights_control_public')
6 va = lightsTempl.POST('va')
7 va.fun.PUT(translationCode)
```

Fig. 3. Example instantiation of virtual resources and configuration.

III. PROGRAMMING

We use CoAP [10] at application level to enable service-oriented interactions with resource-constrained devices. It also supports discovery of services and resources. We implement the virtual resource directory as a CoAP server. Virtual resources are represented as CoAP resources. These support the RESTFUL operations described earlier, making it possible for applications to manipulate virtual resources through CoAP methods.

PyoT [11], an IoT programming system based on Python, supports the execution of virtual resources. PyoT abstracts sensors and actuators as a set of software objects qualified by application-level attributes; for example, the type and the location, which are instrumental to define the input/output resources. PyoT objects can also be used interactively using a shell, which provides a complementary means to manipulate virtual resources, in addition to explicit CoAP calls.

Instantiating virtual resources. Figure 2 shows a PyoT program to publish two templates onto the directory. The code defines a set of input resources for a *virtual sensor* in line 1, using a built-in `resource_set()` method. This takes as input a list of key-value pairs that determine the resources of interest. In this example, these resources include the sensors operating as "power meters" in "public areas".

The set of resources identified in `metersPublic` is used as a parameter to `VsPeriodic` in line 2, which publishes a *periodic-sensor* template on the directory. Our PyoT implementation of the *periodic-sensor* asynchronously collects data from the input resources. When the virtual sensor receives a GET request, it returns the value obtained by applying the processing function on the data of the last time period. The code unfolds similarly in line 3 and 4 for the *simple-actuator* that targets the "light controllers" in "public areas".

Figure 3 exemplifies the instantiation. In line 1, we obtain a reference to the template of the *periodic-sensor* published earlier. This occurs by querying a generic `Resource` object. Alternatively, one may query the directory to discover the available resources dynamically. In line 2, we create the actual instance of virtual resource, passing the name of the new object as a parameter of the POST operation. The newly-created instance is configured in line 3 and 4 by updating the available configuration sub-resources: `fun` for the processing function, and `period` for the sampling period. Similarly, in lines 5 to 7, a *virtual actuator* named `va` is instantiated and configured.

Tying things together. Figure 4 shows an example application running on the Cloud that uses the created virtual resources. The application uses the instances named `vs` and `va`—corresponding to those created in Figure 3—to retrieve

```python
def control_app():
  while True:
    meter_values = vs.GET()
    new_output = control_fun(meter_values)
    va.PUT(new_output)
```

Fig. 4. An example control application using virtual resources.

```python
Input = get_input_list()
setpoint = get_setpoint()
if len(Input) > 0:
  newSetpoint = setpoint / len(Input)
  set_actuator(newSetpoint)
```

Fig. 5. Processing function example for a *virtual actuator*.

values from the virtual sensor and to send commands to the virtual actuator. Figure 4 makes it apparent that the Cloud-hosted application is extremely simplified using VIRTUAL RESOURCES. The code only includes the high-level logic, whereas the lower-level details are delegated to the virtual instances and hidden from the developers.

To manipulate input and output data for virtual resources, developers implement the processing functions using a specific Python API. Figure 5 shows an example executing a command translation on the *simple-actuator*. Functions `get_actuator_list()`, `get_setpoint()` and `set_actuator()` are part of the API we provide. They serve to retrieve the list of active output resources, to read the command sent to the virtual actuator, and to send a new command to the output resources, respectively. Since the binding with input/output resources is dynamic, their number, as returned by `len()` in Figure 5, may change at run-time.

IV. RUN-TIME SUPPORT

Our prototype targets WisMote devices, a platform representative of IoT resource-constrained devices equipped with an MSP430 microcontroller, 16 kB of RAM, and 256 kB of program memory. The nodes run a network stack that includes CoAP, 6LoWPAN [12], and IEEE 802.15.4. To dynamically change the running functionality, these devices normally require a complete replacement of the deployed binaries. We customize T-Res [13], a system enabling in-network processing in CoAP networks, to dynamically allocate processing functions to arbitrary nodes in the IoT network.

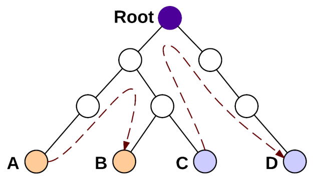
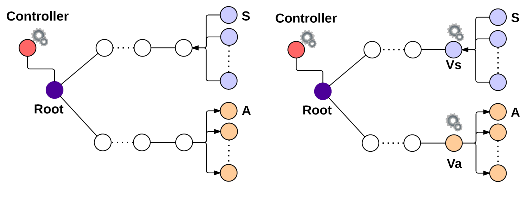


Fig. 6. RPL graph and example routing between nodes.

The performance of a system using VIRTUAL RESOURCES is mainly influenced by what device executes the processing of a virtual resource. This is essentially due to the way the underlying routing protocols operate. The standardized solution for IPv6-enabled IoT devices is RPL [14], which superimposes a Directed Acyclic Graph (DAG) atop the multi-hop physical topology, as shown in Figure 6. The DAG is rooted at a single device, typically the border router that bridges to and from the larger Internet.

When two devices communicate using RPL, messages travel only along the links in the DAG. If source and destination share an ancestor other than the root, this can shortcut the path, as is the case when transmitting from A to B in Figure 6. Otherwise, the message needs to travel up to the root before being forwarded downwards to the destination, as is the case when transmitting from C to D in Figure 6. RPL is indeed optimized for scenarios where most of the traffic goes through the root. However, this is not necessarily the case using



(a) Cloud-centric. (b) VIRTUAL RESOURCES.

Fig. 7. Application setup used throughout the evaluation.

VIRTUAL RESOURCES, which instead generate peer-to-peer traffic among arbitrary nodes; for example, between a virtual sensor and its input resources. If the involved devices happen to be located like C and D in Figure 6, the potential savings in network traffic may vanish.

We thus develop a simple heuristic to drive the placement of virtual resource instances onto the physical devices, applied when RPL has converged to a stable state. This does happen in many scenarios akin to those we target [14]. We take as input the RPL graph and the position of the physical resources. We then determine the positioning of the virtual resources[1], based on a few rules: *i)* virtual sensors (actuators) are positioned on the first common ancestor found by walking the graph from the input (output) resources to the root; *ii)* if multiple such ancestors exists, we take the one farthest from the root; *iii)* if common ancestors other than the root do not exist, the virtual resource is placed on an available node closest to the root.

Rule *i)* ensures that data coming from sensors is processed, and thus reduced in size, as soon as possible on the way to the Cloud, or viceversa that data from the Cloud and addressed to actuators is demultiplexed as close as possible to the output devices. Rule *ii)* break ties by attempting to amplify the effects of rule *i)* on a higher number of hops from/to the root. Finally, rule *iii)* is a fall-back solution in case the common ancestor is indeed the root. This is however seldom usable because the border routing functionality is often taking most of the available resources, leaving little room for running a virtual resource instance as well.

In Section V, we quantitatively measure the effects of applying these rules. However, this is in fact an instance of the well-known task allocation problem [15], where a body of work already exists [16]. We plan to leverage the existing literature to design a provably optimal solution.

V. EVALUATION

We aim at understanding the benefits and performance of VIRTUAL RESOURCES over the Cloud-centric architecture. To this end, Section V-A describes the settings enabling the comparison, whereas Section V-B reports on the results.

A. Setting and Metrics

We realize two functionally-equivalent versions of the building automation application. One design employs a Cloud-centric architecture; the other uses VIRTUAL RESOURCES. We instantiate a virtual sensor that uses a *sliding-window-sensor* template to compute an average over a window of sensor values asynchronously collected from the physical resources. We also create a virtual actuator to collectively drive the actuators.

Besides qualitatively comparing the implementations, we run experiments using Cooja/MSPSim [17]: a cycle-accurate

[1]The heuristic may also re-evaluate the positioning of existing virtual resources in case the underlying RPL graph changes.

wireless simulator. We simulate 21 nodes, representative of IoT installations in medium-size buildings [2]. These include a border router at the RPL root, a variable number of physical sensors and actuators, a variable number of intermediate nodes as message routers, and a control node that runs the Cloud-hosted application. The latter is written in C/Contiki [18], which is directly supported by Cooja/MSPSim, as running the application in an actual Cloud would require taking measures across a hybrid system, unnecessarily complicating the setup.

We first run simulations on top of a controlled network topology, akin to Figure 7 for either the Cloud-centric or the VIRTUAL RESOURCES design. Physical resources are placed at the leaves of the RPL graph, and virtual resources are placed on a node that is both a common ancestor of, and closest to the physical resources. This somehow represents a favorable case for VIRTUAL RESOURCES, which is however not that unrealistic. Indeed, it is reasonable that devices used as input/output of virtual resources be also physically co-located; for example, all power meters on the same floor, and that it would be possible to place the virtual resource on a nearby device. Nevertheless, this setting is solely instrumental to compare the trends—not the absolute performance—of VIRTUAL RESOURCES against the Cloud-centric architecture, *without* the bias due to the physical topologies.

Next, we perform experiments with arbitrary topologies, to understand the performance in settings that represent a *worst case* for VIRTUAL RESOURCES. The underlying physical topology is randomly generated, with the only constraint of ensuring overall (multi-hop) connectivity, and there is initially no control over the placement of virtual and physical resources. This means that, for example, two sensors that may be related at the application level, such as a power meter and the light controllers in the same room, may be placed totally apart in the RPL graph. We then apply the heuristic of Section IV to reposition the virtual resources, and measure the improvements.

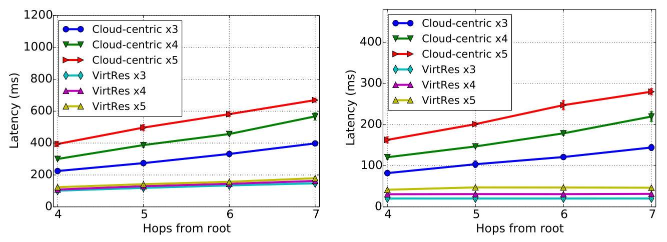
To assess the performance, we measure: *i)* the *control loop latency*; *ii)* the *inter-actuator latency*; and *iii)* the *energy consumption*. The control loop latency is the time between the start of the control loop at the controller and the time when a command is last received by an actuator. The inter-actuator latency measures the time between the first and the last command reception at the actuators, giving an indication of how uniformly the nodes affect the environment.

CoAP messages are sent reliably, that is, packets are retransmitted if an acknowledgment is not received within a timeout. This suffices to complete all 500 iterations of the control loop we test in every setting.

B. Results

Qualitative comparison. Both implementations of the application include a setup phase. In the Cloud-centric design, this consists in discovering what physical sensors and actuators are available to execute the control loop. This processing is mandatory and must be executed periodically. Indeed, the set of available sensors and actuators may change over time; for example, because some devices deplete their batteries, and the application must be made aware of these changes. Generally, using a Cloud-centric architecture, the functionality to maintain a catalogue of active devices is necessarily part of the application's processing.

Using VIRTUAL RESOURCES, the setup includes the processing to create and to configure the virtual instances.



(a) Control loop latency. (b) Inter-actuator latency.

Fig. 8. Controlled topologies with 5 seconds period and no radio duty-cycling. *Control loop and inter-actuator latencies decrease using* VIRTUAL RESOURCES, *while the system scales more gracefully.*

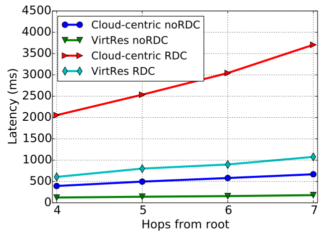
The setup may not be necessary, for example, if another application has already defined the same virtual resources and published them on the directory. In the worst case, this processing is performed only once, as managing the changes in the input/output resources is delegated to the VIRTUAL RESOURCES run-time support. This spares developers from including this functionality in the application processing.

Besides the observations above, interacting with two (fixed) virtual resources turns out simpler than accessing a varying number of physical resources. This reflects, for example, in the code complexity, which we indicatively measure in LLOC. Using VIRTUAL RESOURCES, the application processing amounts to 96 LLOC for the control loop, plus additional 24 LLOC optionally required to instantiate and configure the virtual resources. The latter serve to create resources that become part of the directory and may be shared with other applications. In contrast, the Cloud-centric implementation requires 167 LLOC that only serve a single application.

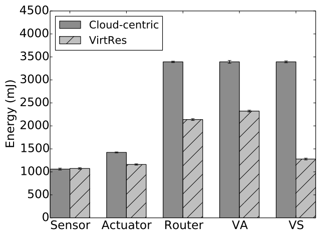
Controlled topologies. We run the system with a 5 second control period. This greatly overestimates the dynamics of building automation systems, which typically control phenomena with slow dynamics, such as temperature. Control loop periods are therefore in the range of minutes. We intentionally push on this dimension to stress the systems. Nevertheless, the trends and observations we discuss next do apply—sometimes even with greater evidence—also when setting larger control periods. We momentarily disable radio duty-cycle to obtain measures of control loop latency and inter-actuator latency not affected by the added delays of energy saving mechanisms.

Figure 8 illustrates a sample of our results to understand the trends at stake. Figure 8(a) demonstrates that using VIRTUAL RESOURCES significantly decreases the control loop latencies. The improvements grow as the hop-distance between controller and physical resources increases. As this happens, the beneficial effect of the processing at the virtual resources progressively amplifies. The system also shows better scalability as the number of physical resources grows: the curves corresponding to a different number of physical sensors, from 3 to 5 in the chart, grow more slowly using VIRTUAL RESOURCES. Similar observations apply to Figure 8(b), with the only difference that inter-actuator latencies remain constant using VIRTUAL RESOURCES, because the *virtual actuator* is always one hop away from the physical ones.

In contrast, Figure 9 studies how the performance changes in a more realistic configuration that uses radio duty-cycling for saving energy, in a sample setting with 5 physical resources. Figure 9(a) shows that the improvements in control loop latency amplify in favor of VIRTUAL RESOURCES. This

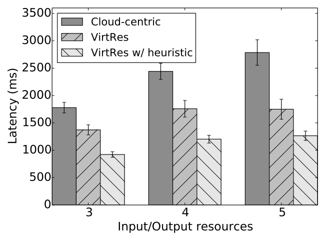


(a) Control loop latency.

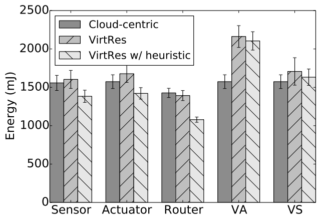


(b) Per-round energy consumption.

Fig. 9. Controlled topologies with 5 seconds period, using 5 physical resources. *The gains in latencies amplify in favor of* VIRTUAL RESOURCES*, which also saves energy at message routers and at the controller node.*



(a) Control loop latency.



(b) Energy consumption

Fig. 10. Random topologies with 5 seconds period and radio duty-cycling. *Control loop latencies are more than halved with the heuristic placement; energy consumption reduces as well.*

is essentially because the per-hop delay grows when using radio duty-cycling; therefore, the impact of reduced traffic becomes more significant. Figure 9(b) illustrates the gains in energy consumption depending on the processing at a node. Intermediate nodes operating as message routers forward fewer messages using VIRTUAL RESOURCES, so they consume less energy. The nodes running virtual sensors or virtual actuators, compared in Figure 9(b) to the figure when operating as message routers in the Cloud-centric design, also improve their energy efficiency. Physical resources consume about the same energy in that their processing is similar in the two settings.

Random topologies. Figure 10 summarizes the results obtained from more than 70 different randomly-generated wireless topologies. As already mentioned, these represent a worst-case for VIRTUAL RESOURCES, as the placement of virtual and physical resources is initially random.

The control loop latency, shown in Figure 10(a), improves for VIRTUAL RESOURCES even when considering the fully random placement. After applying the heuristic of Section IV to re-position the virtual resources, the results further improve. Ultimately, the control loop latency is more than halved. Similar considerations also apply to the inter-actuator latency in the same settings—not shown here for brevity—albeit the improvements are not as high. In both cases, the gains are enabled by the traffic reduction enabled by VIRTUAL RESOURCES, while still maintaining the RESTFUL interactions.

Figure 10(b) demonstrates the energy improvements. As before, nodes operating as message routers reduce the energy expenditures because of the lower traffic. Different than before, however, also the physical sensors and actuators consume less energy when using VIRTUAL RESOURCES with the heuristic placement. As their location is randomly decided, it may happen that they are *not* placed at the leafs of the RPL graph. In these cases, they may need to double as message routers, hence the improvements of the latter apply to them too.

VI. CONCLUSION

We presented the VIRTUAL RESOURCES architecture, which contrasts the traditional Cloud-centric designs by giving developers the ability to push slices of the application logic down to the IoT network. VIRTUAL RESOURCES provide several benefits to developers, including better utilization of network resources that results in higher energy efficiency and lower latencies, a simplification of the application logic at the Cloud, and better separation of concerns throughout the development process. We designed a RESTFUL interface to manipulate virtual resources and a CoAP-based prototype providing dedicated programming support. Our results indicate that VIRTUAL RESOURCES achieves a 40% improvements in energy consumption and a 60% improvement in control loop latency, while retaining RESTFUL interactions.

Acknowledgements. This work was partly supported by the Swedish Innovation Agency VINNOVA and the Technological Cluster Projects "Zero-energy Buildings in Smart Urban Districts" (EEB), "ICT Solutions to Support Logistics and Transport Processes" (ITS), and "Smart Living Technologies" (SHELL) of the Italian Ministry for University and Research.

REFERENCES

[1] L. Atzori *et al.*, "The Internet of Things: A survey," *Computer networks*, vol. 54, no. 15, 2010.
[2] K. Whitehouse, "The rise of the intelligent building," *IQT Quarterly*, no. 3, 2013.
[3] M. Johnson *et al.*, "A comparative review of wireless sensor network mote technologies," in *Sensors*, 2009.
[4] M. Kovatsch *et al.*, "Moving application logic from the firmware to the cloud: Towards the thin server architecture for the Internet of Things," in *IMIS*, 2012.
[5] T. Teixeira *et al.*, "Service oriented middleware for the Internet of Things: A perspective," in *Towards a Service-Based Internet*. Springer, 2011.
[6] N. Mitton *et al.*, "Combining Cloud and sensors in a smart city environment," *Journal on Wireless Communications and Networking*, vol. 2012, no. 1, 2012.
[7] D. Guinard *et al.*, "A resource oriented architecture for the Web of Things," in *Internet of Things (IOT)*, 2010.
[8] P. Ciciriello *et al.*, "Building virtual sensors and actuators over Logical Neighborhoods," in *MidSens*, 2006.
[9] "ISO 17458—FlexRay Communications System," goo.gl/kZhgYY.
[10] Z. Shelby, "Embedded web services," *IEEE Wireless Communications*, vol. 17, no. 6, 2010.
[11] A. Azzarà *et al.*, "PyoT, a macroprogramming framework for the Internet of Things," in *SIES*, 2014.
[12] Z. Shelby and C. Bormann, *6LoWPAN: the wireless embedded internet*. John Wiley & Sons, 2011, vol. 43.
[13] D. Alessandrelli *et al.*, "T-Res: Enabling Reconfigurable In-network Processing in IoT-based WSNs," in *DCOSS*, 2013.
[14] O. Iova *et al.*, "Stability and efficiency of RPL under realistic conditions in wireless sensor networks," in *PIMRC*, 2013.
[15] S. Salcedo-Sanz *et al.*, "Hybrid meta-heuristics algorithms for task assignment in heterogeneous computing systems," *Computers & operations research*, vol. 33, no. 3, 2006.
[16] B. J. Bonfils and P. Bonnet, "Adaptive and decentralized operator placement for in-network query processing," in *IPSN*, 2003.
[17] J. Eriksson *et al.*, "COOJA/MSPSim: Interoperability testing for wireless sensor networks," in *SIMUTools*, 2009.
[18] A. Dunkels *et al.*, "Contiki: A lightweight and flexible operating system for tiny networked sensors," in *LCN*, 2004.