Mastering interactions with Internet of Things platforms through the IoTVar middleware

Pedro Victor Borges, Chantal Taconet, Sophie Chabridon, Denis Conan, Thais Batista, Everton Cavalcante, Cesar Batista

proceedings

MDPI

Proceedings

Mastering Interactions with Internet of Things Platforms through the IoTVar Middleware †

Pedro Victor Borges [1,2], Chantal Taconet [2,*], Sophie Chabridon [2], Denis Conan [2], Thais Batista [1], Everton Cavalcante [1] and Cesar Batista [1]

[1] Department of Informatics and Applied Mathematics, Federal University of Rio Grande do Norte, Natal 59064-741, Brazil; pedro.borges@telecom-sudparis.eu (P.V.B.); thais@dimap.ufrn.br (T.B.); everton@dimap.ufrn.br (E.C.); cesarbatista@ppgsc.ufrn.br (C.B.)

[2] SAMOVAR, CNRS, Télécom SudParis/Institut Polytechnique de Paris, 91000 Évry, France; sophie.chabridon@telecom-sudparis.eu (S.C.); denis.conan@telecom-sudparis.eu (D.C.)

* Correspondence: chantal.taconet@telecom-sudparis.eu

† Presented at the 13th International Conference on Ubiquitous Computing and Ambient Intelligence UCAmI 2019, Toledo, Spain, 2–5 December 2019.

Published: 21 November 2019

Abstract: The rising popularity of the Internet of Things (IoT) has led to a plethora of highly heterogeneous, geographically dispersed devices. In recent years, IoT platforms have been used to provide a variety of services to applications such as device discovery, context management, and data analysis. However, the lack of standardization currently means that each IoT platform comes with its own abstractions, APIs, and interactions. As a consequence, programming the interactions between an application and an IoT platform is often time consuming, error prone, and depends on the developers' level of knowledge about the IoT platform. To address these issues, we propose offering to application developers on the client side the possibility to declare variables that are automatically mapped to sensors and whose values are transparently updated with sensor observations. For this purpose, we introduce IoTVar, a middleware between IoT applications and platforms. In IoTVar, all the necessary interactions with IoT platforms are managed by proxies. This paper presents IoTVar integrated with the FIWARE platform, which is used for developing IoT Future Internet applications. We also report results of some experiments performed to evaluate IoTVar, showing IoTVar reduces the effort required to declare and manage IoT variables and its impact in terms of CPU, memory, and energy.

Keywords: Internet of Things; IoT platform; middleware

1. Introduction

The use of the Internet of Things (IoT) in both industry and society is becoming widespread [1]. New services and applications (these applications will be hereinafter referred to as *IoT applications*.) are proposed in several domains such as smart cities, medical assistance, logistics, and smart homes. As a consequence, complex systems encompassing IoT applications, platforms, and physical and virtual devices geographically dispersed arise.

A common architecture of IoT include a number of devices with capabilities for sending data as well as being acted upon receiving data. The devices communicate either directly with an IoT platform or by using gateways. They register to a platform and publish data as soon as they are ready. Objects such as sensors are often represented as virtualized entities within the IoT platform. Both IoT platforms and gateways are connected to the Internet and they make object data available to end-user IoT applications. In this paper, we focus on the interactions between IoT applications and IoT platforms. This architecture is illustrated by the Figure 1.

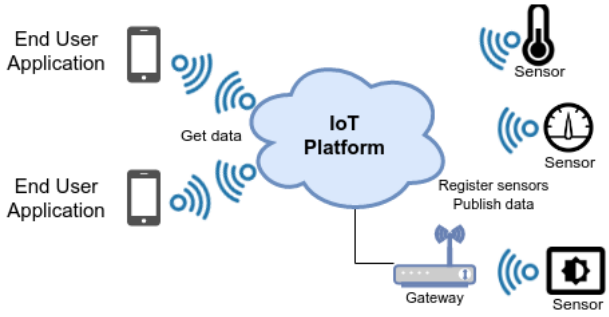


Figure 1. General Internet of Things (IoT) architecture.

IoT systems are characterized by a high heterogeneity as a result of the diversity of hardware and software technologies bringing new challenges to the middleware community [2]. Middleware is used in the contemporary IoT systems as they provide (i) abstractions for accessing volatile physical devices and managing the data produced by the devices, (ii) virtualization and aggregation functions of many devices into IoT platforms, and (iii) interoperability patterns for managing software entities [3,4].

IoT platforms facilitate communication and data flow between IoT applications and devices. These platforms provide interfaces and typically support different data transmission methods, communication protocols, and computational capabilities. They also include a variety of middleware services to ease application development, such as device discovery, context management, and data analysis [5]. As an example, to display the current temperature at the Eiffel Tower in Paris, application developers have to program the interactions with an IoT platform that shows up several virtual entities that provide the updated temperature data around the area. They need to adapt their code to the API of the IoT platform with a dedicated interaction mode (e.g., request/reply or publish/subscribe) and may have to programmatically select the sensors. Other tasks include (un)marshalling data and manipulating sensor identifiers and metadata, such as quality attributes of sensor data. Furthermore, they may have to tune the frequency of interactions to limit the usage of computational resources. These issues are exacerbated when managing sensor devices directly or through gateways, without IoT platforms.

IoT platforms can doubtless be used by IoT applications, but application developers still face many barriers. All the aforementioned development tasks are time consuming, error prone, and depend on the developers' level of knowledge of the IoT platform and on how easy it is to get started with the targeted IoT platform. In this paper, we aim at demonstrating that the usage of IoT platforms can be abstracted with middleware on the application client side to leverage even more the IoT platform model by facilitating the provision of services. We propose the concept of *IoT variables*, which was previously introduced in a previous work [6]. An IoT variable is a variable that has an observation attribute representing a value measured in the environment by a sensor that is automatically updated. In our proposition, IoT variables are managed by the *IoTVar* middleware. IoTVar provides proxies that manage all the interactions with IoT platforms. As a main contribution, IoTVar enables developers to easily discover devices and transparently update context data at low development cost in terms of lines of code.

This paper presents the IoTVar middleware (https://fusionforge.int-evry.fr/www/iotvar/) and details its architecture. For illustration purposes, we present the IoTVar handler for FIWARE [7], an European initiative intended to provide an interoperable middleware platform made up of several generic components to leverage the development of smart cities, IoT, and Future Internet applications. We use a client-side application to compare its interaction directly with FIWARE and through the IoTVar middleware. We assessed how IoTVar can reduce development effort in terms of lines of code and the extra cost of the middleware with respect to CPU, memory, and energy.

The remainder of this paper is organized as follows. Section 2 briefly discusses related work. Section 3 introduces IoTVar. Section 4 presents the integration of IoTVar with the FIWARE platform.

Section 5 presents the results of the evaluation of IoTVar. Section 6 contains the conclusion with directions to ongoing and future work.

2. Related Work

Due to the lack of standardization of IoT protocols throughout their layers (device, network, middleware, and application layers), a variety of platforms have been proposed in the recent years [8]. Those platforms are mostly complex systems that seek to provide developers with a common way for building IoT applications, but often coupled to their syntax and semantics. Although such platforms are generally robust, they create what is called "vertical-silos" [9], i.e., domain-oriented systems offering a limited set of smart objects and hence becoming insufficient for cross-domain applications such as smart cities. Furthermore, such a strong coupling is a major barrier to developers who need to consume resources allocated in more than one platform, thereby forcing them to learn platform-specific concepts such as data formats and interaction patterns. To tackle these problems, many approaches are being proposed such as (i) promoting interoperability and standardized APIs for IoT platforms, (ii) developing server-side adapters, and (iii) deploying client-side proxies.

To achieve interoperability among many platforms, the BIG-IoT [10] and the INTER-IoT [11] projects propose APIs to be implemented by platforms to provide cross-platform access. Each one promotes interoperability among heterogeneous IoT platforms, granting applications access to a high-level API for communication with IoT devices from different platforms. These projects are close to the bIoTope project [12] as it provides standardized open APIs to enable publication, consumption, and composition of heterogeneous information sources and services between the today's IoT vertical silos. Through standards, the bIoTope high-level API grants the interconnection among different systems and allows IoT data/services to securely join the bIoTope ecosystem. From the application point of view, these projects aim at proposing a standardized API on the platform side, but they do not manage the interactions between the application and the IoT platforms. In turn, IoTVar is situated at the client side to manage interactions between the IoT application and IoT platforms and it handles data model translation to provide data from different IoT platforms to applications. Nonetheless, an IoTVar extension should be provided for each targeted platform.

The VICINITY project [13] proposes a virtual neighborhood concept to connect owners of IoT infrastructures. It enables users without technical background to connect different smart objects and control devices through a Web-based operator console. Guest IoT infrastructures and VICINITY-enabled services are connected to an interoperability gateway through the VICINITY gateway API. As with IoTVar, other platforms can be integrated to VICINITY. They use the concept of adapters at the server side, resulting from implementations of the VICINITY gateway API. However, IoTVar concerns providing these adaptations at the client side. IoTVar does not consider hardware and network layers such as gateways as it contemplates the application layer through data format translation from the underlying IoT platforms.

Obtaining up-to-date values from IoT platforms may be complex and specific proxy objects help to manage these remote interactions. This proxy design pattern appeared in the 1980's with remote procedure calls (RPCs) [14] and distributed objects [15]. Since then, proxies are used to simplify the implementation of complex distributed systems. Examples of distributed objects are the CORBA objects [16], which have remote proxies to manage the communication with remote objects. The CORBA group objects [17] handles groups of remote proxies and the group proxy manages the group communication. Another example is a proxy for replicated objects such as Creason [18], in which the replicated object proxy manages object replication. In IoTVar, we use the proxy pattern to create IoT variables as proxies that manage the interactions with IoT platforms with the purpose of discovering and updating environment variables captured by sensors.

3. IoTVar

In this section, we briefly present IoTVar and show its usage through an example in Section 3.1. Next, we describe IoTVar proxies in Section 3.2.

3.1. IoTVar by Example

IoTVar is a middleware that provides developers with the possibility to declare IoT variables, easily discover some corresponding data-producer objects, and transparently dealing with updates on these data from IoT platforms. IoTVar offer an abstraction level to interact with virtualized sensors and hence minimize the number of lines of code to be written by the client application developer to obtain up-to-date sensor data.

The usage of IoTVar is shown in Listings 1 to 3 with code in the Java programming language to display the up-to-date temperature in the vicinity of the Eiffel Tower. Listing 1 shows the declaration of the IoT variable for a platform (Lines 1–9) and the registration of a listener for this variable (Line 10). To declare an IoT variable, the developer provides IoTVar, for example, with the *id* (Line 2) and *type* (Line 3) of the searched-for sensor (`Temperature` at Line 4), the *location* of the Eiffel Tower (Line 5), the required refresh time as *quality parameter* (Line 6), the *size of a local history* list of values (Line 7), additional *filters* to be passed to the platform to filter out data (Line 7), the *URL* of the platform (Line 7), the *interaction strategy* with the IoT platform (Line 8), and the *class* to be used for unmarshalling purposes (Line 9). The location is defined by *latitude*, *longitude*, and *radius* in meters specifying the maximum distance between the sensor to be selected and the Eiffel Tower.

Listing 1. Declaration of FIWAREIoT variable with id and type.

```
1  IoTVariableFiware<Integer> temperatureEiffelTower =
2    new IoTVariableFiware<>("temperature_eiffel_tower_310", // ID
3      "LM35", // Type
4      "Temperature", // Attribute
5      new Location("location", 48.6223426, 2.4404356, 100.0),
6      new RefreshTime(10, TimeUnit.SECONDS),
7      10, null, orionConfiguration, // History size, filters and plat. config.
8      HandlerStrategy.SYNC // Or PubSub
9      Integer.class);
```

IoTVar automatically activates a listener each time the temperature observation is updated, either when a refresh has been requested (in the synchronous strategy) or when a notification has been received (in the publish-subscribe strategy). Listing 2 shows the basic interface for a listener with the `onUpdate` and `updateIssue` methods. The former is called each time a new observation is provided by IoTVar whereas the latter is called when the update cannot comply with the specified constraints. When fetching for data, IoTVar might undergo network or platform errors as well as lack of sensor availability. This leads to call the `updateIssue` method to notify the application with a message regarding what has happened.

Listing 2. Interface of a listener.

```
1  public interface IoTVarObserver {
2    void onUpdate(Observation newObservation);
3    void updateIssue(String issue);
4  }
```

Listing 3 presents the code of a simple listener class `TextDisplay`, which implements the `IoTVarObserver` interface. The listener class defines an implementation for the `onUpdate` and `updateIssue` methods (Lines 3–8). In this example, the method simply logs the observation (Line 4). The listener class also defines the `updateIssue` method (Lines 6–8).

Listing 3. Declaration of a listener.

```
1 public class TextDisplay<Integer> implements IoTVarObserver {
2   private static final Logger logger = LogManager.getLogger(TextDisplay.class);
3   public void onUpdate(Observation newObservation) {
4     logger.info(Thread.currentThread().getName() + " " + newObservation);
5   }
6   public void updateIssue(String issue) {
7     logger.info(issue);
8   }
9 }
```

3.2. IoTVar Proxies

An IoT variable is a proxy to an entity of an IoT platform. The proxy activates the handler in charge of communicating with the IoT platform. When the proxy receives an updated observation, it unmarshalls the received data, updates the observation attribute of the corresponding IoT variable, and calls the onUpdate method of the registered listener (see Listing 3).

The proxy transparently manages all interactions between the application and the IoT platform. According to the possibilities of the IoT platform, IoTVar may propose one or both of the two interaction patterns, namely the synchronous and/or publish-subscribe ones.

When using the synchronous call pattern, the application periodically requests for updates based on the specified refresh time. A pool of threads is in charge of periodically requesting updates of the variable. The `onUpdate` method is called when receiving the reply.

In the publish-subscribe pattern, the application registers at the IoT platform for updates and the platform notifies about updates. Some platforms may limit the frequency of the notifications according to the IoT variable refresh time. If the platform provides such a capability, then IoTVar registers an additional parameter for the minimum time between two updates. When IoTVar specifies a refresh time of ten seconds, the IoT platform notifies IoTVar only after ten seconds, even if the entity is updated every second at the IoT platform. The `onUpdate` method is called when IoTVar is notified. As some platforms may filter and notify only significant changes, a variable whose linked entity does not significantly change is not updated.

4. Integrating IoTVar with the FIWARE Platform

IoTVar provides resources to ease the integration with IoT platforms, which can be done by extending a common `IoTVariable` class for each platform. In this section, we show the integration of IoTVar with the FIWARE platform. We first introduce FIWARE in Section 4.1. Next, Section 4.2 describes the integration of IoTVar and FIWARE. At last, we present examples of `IoTVariableFiware` declarations that handle specific searches in Section 4.3.

4.1. Presentation of the FIWARE IoT Platform

To illustrate our proposal, we have chosen FIWARE, a popular platform supported by the European Community. FIWARE is a generic, open-source solution and it provides many extensible, reusable, interoperable components to make it easy system development in different application domains, the so-called *Generic Enablers (GEs)*. FIWARE encompasses GEs for different purposes, such as context entity management, device management, historical data storage, entity event processing, security, creation of dashboards, etc.

The main FIWARE GE is the *Orion Context Broker* (or simply *Orion*), which is responsible for managing context entities and subscriptions of parties interested in its state changes. Orion follows the Next Generation Service Interfaces (NGSI) specification model [19] to standardize information exchange and allow for interoperability among components. In NGSI, the information is structured in a generic way through entities that can be used to represent both physical and virtual elements such as

a building, a car, sensors, actuators, etc. As depicted in Figure 2, an NGSI *Entity* has an identifier, a type, and a list of attributes. Attributes have a name, a type, a value, and a list of metadata, which have the same data structure as the attributes.

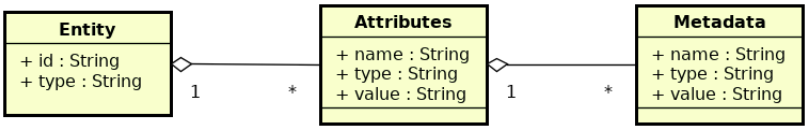


Figure 2. Next Generation Service Interfaces (NGSI) entity model.

Listing 4 presents the representation of an entity and its attributes in Orion. The entity is a sensor near the Eiffel Tower with recorded temperature (Line 4), location (Line 14), and temperature resolution (Line 18) as attributes. All the attributes are specified as additional properties in JSON and support metadata declaration.

Listing 4. Representation of temperature sensor data in Orion.

```
1  {
2   "id": "temperature_eiffel_tower_310",
3   "type": "LM35",
4   "temperature": {
5    "value": 23.3,
6    "type": "Float",
7    "metadata": {
8     "Unit": {
9      "value": "Celsius",
10     "type": "String"
11    }
12   }
13  },
14  "location": {
15   "value": "48.6223426, 2.4404356",
16   "type": "geo:point"
17    },
18  "temperature_resolution": {
19   "value": "0.1",
20   "type": "Float"
21  }
22 }
```

Orion allows managing of the entire life cycle of context information, including updates, queries, registrations, and subscriptions. Working with it, there are information producers (such as IoT devices) and information consumers (such as IoT applications and users). Producers create entity elements and manage them through updates. Application developers can search for entities by making simple HTTP GET requests with parameters to narrow the search and to specify the quality of the data. In addition, application developers can subscribe to be notified when data have changed, for example.

4.2. IoTVar Extension Architecture

The general architecture of the integration of IoTVar with FIWARE is shown in Figure 3. IoTVar provides common protocol handlers components for interacting with IoT platforms. For example, the *MQTT Handler* is a prototypical example of a common communication component: it contains all required code for handling the interactions with any platform that provides MQTT communication. Another example of common component is the *HTTP Synchronous Handler*. Platforms requiring to use these components only need to extend it by having two small additional components as each platform handles sent and received data in a different way.

The communication with the FIWARE platform can take place using either the *HTTP Synchronous handler* or the *HTTP Pub-Sub Handler*. The FIWARE's dedicated components are used to marshall the body of requests and unmarshall the body of responses. These components use the dedicated data unmarshaller to transform the received data into application objects. For example, the FIWARE unmarshaller receives an NGSI data model from one handler and extracts the desired observation, e.g., transform it into an `Integer` object that represents the temperature.

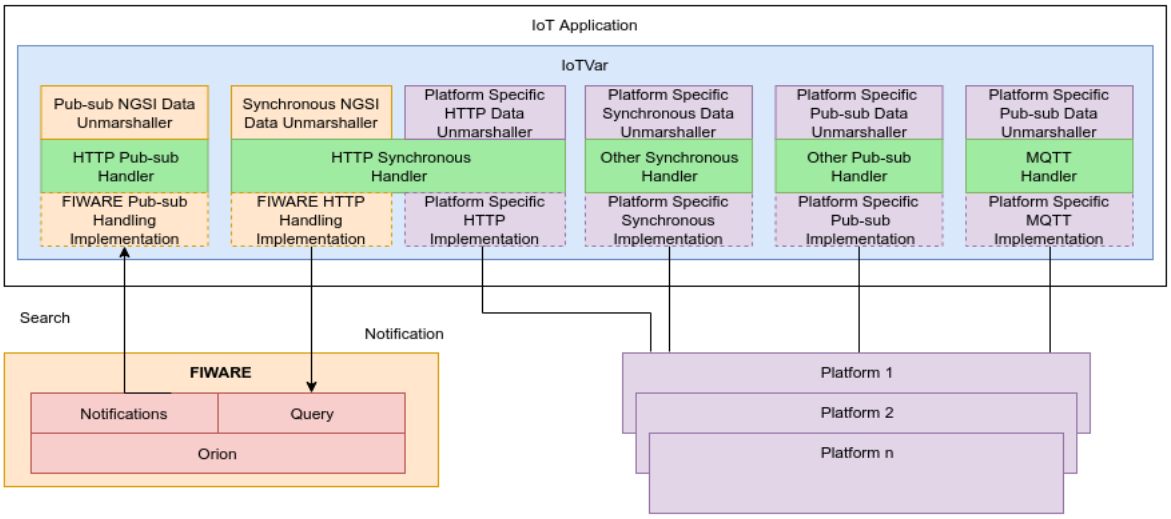


Figure 3. Integration architecture of IoTVar with the FIWARE platform.

4.3. Declaration of FIWARE IoT Variables

FIWARE provides many features to search for entities. The following examples show how IoTVar benefits from these possibilities.

Searching for sensors in FIWARE is done by giving either an *id* and *type* of an entity or a pattern of *id* and *type*, along with an attribute in both cases. Listing 1 previously shown in Section 3 is an example of how a search can be done with sensor *id* and *type* (Lines 2–3). In this case, a request is made to Orion aiming at retrieving the sensor.

`IoTVariableFiware` is an extension of the `IoTVariable` class. Listing 5 shows how the search is done with the location of the sensor, so that the *id* and *type* are given as patterns (https://bit.ly/2IQrjKg) (Lines 2–3). They also need an attribute to be searched for (Line 4) while the other parameters are similar to those used in Listing 1. In this case, multiple entities registered at Orion could be retrieved, but the support made by IoTVar is that each searched attribute is one variable in the code and it is linked to one entity in Orion. When making the request to the Orion's API, special parameters are used to ensure that only one entity is returned, thus reducing the size of the message returned by Orion.

Listing 5. Declaration of FIWARE IoT variable with location and type.

```
1 IoTVariableFiware<Integer> temperatureEiffelTower =
2   new IoTVariableFiware<>(".*", // Regular Expression
3     ".*", // Regular Expression
4     "Temperature",
5     new Location("location", 48.6223426, 2.4404356, 100.0),
6     new RefreshTime(10, TimeUnit.SECONDS),
7     10, null, orionConfiguration,
8     HandlerStrategy.SYNC,
9     Integer.class);
```

IoTVar also allows using filters to search for devices containing attributes with values, as shown in Listing 6 (Line 13). These filters are created by using the `OrionFilter` class (Lines 3 and 6) and giving an attribute, a value, and a logical operator defined by the class `OrionSQLOPerator`, which includes logical operators to compare the value given with the value stored in Orion. For example, consider a search for temperature sensors. There may be many of those sensors, but if the user wants one with resolution of 0.1°C and the entity in Orion has this attribute (e.g., temperature resolution), then a filter can be used by providing `temperature resolution > 0.1` to filter the sensors complying with this criterion. Listing 6 (Lines 2–3) shows the search for a sensor with this logic as long as Orion support such filtering expressions.

In terms of codification, the developer who wants to interact directly with Orion to provide functions similar to the ones provided by IoTVar needs to understand a significant number of parameters of the Orion API. The developer also needs to understand the used data model towards properly processing both the information to be sent to Orion and the received data. Furthermore, data need to be sent through HTTP requests that need to be implemented. If the publish-subscribe pattern is used, then the developer must also implement a local HTTP server to listen notifications from Orion. All of these tasks are abstracted away from developers by proxies provided by IoTVar.

Listing 6. Declaration of FIWARE IoT variable using filters.

```
ArrayList<OrionFilter> filterList = new ArrayList<OrionFilter>();
OrionFilter f1 =
  new OrionFilter("temperature_resolution", "0.1", OrionSQLOPerator.LESS_EQUAL);
filterList.add(f1);
OrionFilter f2 =
  new OrionFilter("temperature", "20", OrionSQLOPerator.GREATER);
filterList.add(f2);

IoTVariableFiware<Integer> var3 =
  new IoTVariableFiware<>(".*", ".*", "Temperature",
    new Location("location", 48.6223426, 2.4404356, 100.0),
    new RefreshTime(1, TimeUnit.SECONDS), 10,
    filterList, orionConfiguration, HandlerStrategy.PUBSUB
    Integer.class);
```

5. Evaluation

In this section, we report a quantitative evaluation of IoTVar through the integration with the FIWARE platform. Section 5.1 describes the evaluation IoTVar from the point of view of application developers. Section 5.2 presents a performance assessment of IoTVar regarding the usage of memory, the CPU, and the energy.

5.1. Development Effort Assessment

Regarding the number of lines of code, application developers who want to directly use the Orion API need to code about 450 lines of code for a synchronous interaction and 600 lines of code for a publish-subscribe interaction. On the other hand, the basic setup with IoTVar can be done with only 15 lines of code for both synchronous and publish/subscribe interaction patterns.

It is noteworthy that the use of IoTVar releases application developers from the need of learning the specificities of the Orion API and data model. Nonetheless, IoTVar may not provide support for all the operations available through the IoT platform API. For example, the FIWARE Orion additionally provides metadata filtering (e.g., accuracy of a sensor), more complex geospatial filtering (e.g., polygon area search), and device actuation (e.g., turn a lamp on). These functionalities can all be added to IoTVar after studying the Orion API and implementing the code to abstract them.

5.2. Performance Assessment

For the performance evaluations, we compare the same application with and without IoTVar (i.e., directly accessing the FIWARE platform) aiming at measuring the internal cost of IoTVar in terms of energy, CPU, and memory usage. We collect the measurements by wrapping both the method called by the synchronous handler and the method to handle notifications sent by Orion.

Data for each measure (CPU, memory, and battery) were collected in three minutes. The first minute of the test was not recorded to ensure that the class loading is complete in the Java Virtual Machine (JVM) and hence avoid interferences of such a warm-up time in the results. The last two minutes are the effectively run phase. In addition, an external producer creates from 25 to 200 sensors and then starts updating these sensor observations every one second.

Each IoT application (with or without IoTVar) gets data from 25 to 200 sensors and uses either the synchronous call or the publish-subscribe pattern. In the former, the IoT application with IoTVar creates an IoT variable for each sensor and IoTVar sends one request per second to get data for each declared sensor. In the latter, the IoT application with IoTVar creates an IoT variable for each sensor, IoTVar registers to the corresponding entity, and the IoT platform notifies IoTVar about all the updates.

As depicted in Figure 4, we used two computers. The first one run an instance of the FIWARE Orion and the emulator over an Intel® Core™ i3-4130 3.40 GHz, 8 GB of RAM, and Linux Ubuntu 18.04 as operating system. The second one run the IoT client application over an Intel® Core™ i7-5600U 3.20 GHz, 8 GB of RAM, and Linux Debian 8 as operating system. In addition, it has a lithium battery with 42 Wh of capacity. All the measurements were made in the client machine.

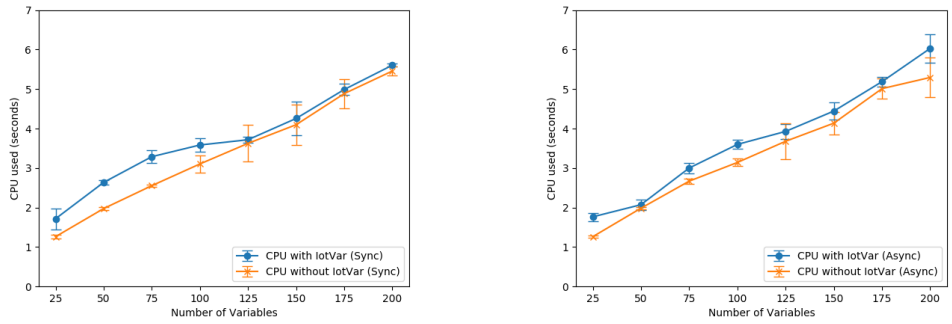
Figure 4. Computational infrastructure used in the experimental setup.

Considering the goal of the evaluation and the previously defined metrics, we have formulated a set of hypotheses to be investigated. The null hypothesis states that there is no statistically significant difference between using IoTVar and not using it. The alternative hypothesis state that there is a statistically significant difference regarding the use of IoTVar, i.e., IoTVar introduces a non neglectful overhead. These hypotheses will be validated through a quantitative analysis using hypothesis testing [20]. A hypothesis test returns a p-value, which can be understood as the probability of the observed result to accept the null hypothesis.

5.2.1. CPU Time

The CPU performance for synchronous and publish-subscribe handling is shown in Figure 5. We can notice that for both synchronous and publish/subscribe handler, IoTVar consumes more CPU, a consequence of increasing the number of declared variables. The extra consumption is due to the fact that IoTVar performs more processing to identify and maintain the variables alive in the code, manage the history of observations, and detect faults and errors from connection failures or platform errors.

Even though the publish-subscribe pattern requires more operations for handling the notification in comparison to the synchronous one, we can notice that there is not much difference between synchronous and publish-subscribe handling. The synchronous response message consists in a simple JSON vector with size of 38 bytes containing the value of the observation. The notification for the publish/subscribe handling consists of a JSON containing the identifier and type of the entity along with the attribute being updated, and the coordinates of the entity (if it exists) with a total size of 380 bytes. Data handling is more CPU consuming in the publish-subscribe mode, but the synchronous one has more usage of network as it needs to make HTTP requests to the server while the publish-subscribe handler simply waits for notifications. It is important to highlight that our experiment considered the refresh time as equal to the sensor emulation period. If the sensor emulation period would be greater than the refresh time, then the publish-subscribe mode would give better results than the synchronous mode.

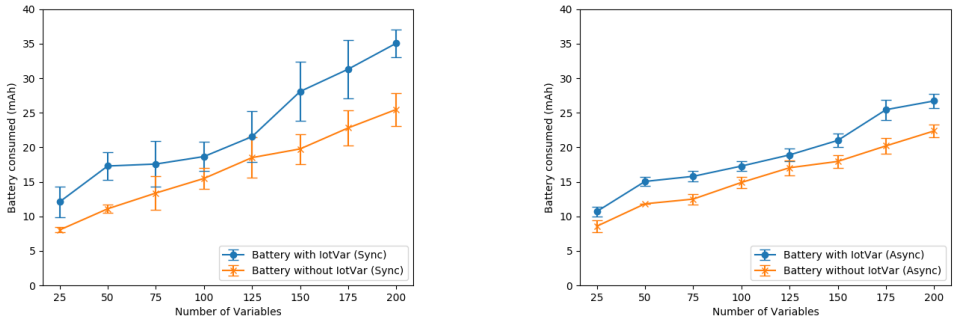


(**a**) (**b**)

Figure 5. CPU usage for (**a**) a synchronous handling and (**b**) a publish-subscribe handling.

5.2.2. Energy Consumption

Figure 6 shows the consumption of battery in mAh for the synchronous and publish/subscribe handling. For the same reasons as the CPU consumption, we can see that the consumption for IoTVar is greater than the one without IoTVar.



(**a**) (**b**)

Figure 6. Battery usage for (**a**) a synchronous handling and (**b**) a publish-subscribe handling.

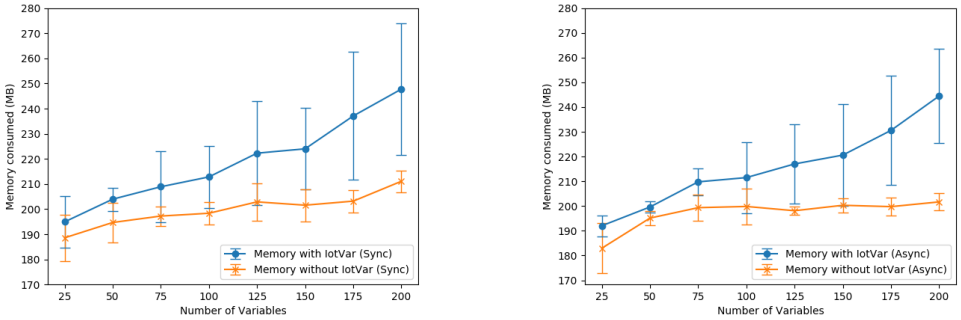
The synchronous handling is always more battery consuming than by using publish-subscribe handling. This is due to the fact that more network resources are used to make the HTTP requests and for processing when using the synchronous handler. In the publish-subscribe handler, we wait for notifications from the platform and perform some processing to transform received data and identify the variable being updated. As shown in Figure 6, the impact of the HTTP requests are higher for the battery consumption.

5.2.3. Memory

The memory usage of the application is shown in Figure 7 for synchronous and publish-subscribe handling. When using IoTVar in both cases, the memory consumption is greater than when not using it. This happens because the number of objects created to handle the updates increases with the number of variables. In contrast, the number of used memory without IoTVar increases by small amounts as the number of variables increases. This is due to fact that it does less processing as it does not create as many objects to handle the processing and does not maintain a history of the observations.

When comparing synchronous and publish-subscribe with respect to memory consumption, we notice that there is small or no difference in values. The amount of memory used in the publish-subscribe mode remains close from the synchronous values, with and without the use of IoTVar. This can be explained by the fact that the amount of objects created by the code does not stray

too far. Both synchronous and publish-subscribe handlers have the same objects with minor changes that do not impact the overall memory usage.



(**a**) (**b**)

Figure 7. Memory usage for (**a**) a synchronous handling and (**b**) a publish-subscribe handling.

5.3. Results

For quantitative analysis purposes, we have performed hypothesis testing to verify the validity of the hypotheses stated for the study (see Section 5.2). To decide about the test to use, we have first performed the Shapiro-Wilk test [21], a powerful statistical test to verify if a sample follows a normal distribution. For a significance level $\alpha = 0.05$, we noticed that the values followed a normal distribution, thus leading to use a parametric hypothesis test.

We have chosen the Student's t-test [22], one of the most used parametric tests to verify if there are differences between two independent samples. The null hypothesis states that the means of the values in the samples are the same, i.e., there is no statistically significant difference between the samples. The t-test returns a p-value that is compared to the adopted significance level $\alpha = 0.05$. If the p-value $< \alpha$, then the null hypothesis is rejected and the alternative hypothesis is accepted, otherwise it is not possible to conclude if there is statistically significant difference between the analyzed samples. In our study, we have adopted a significance level $\alpha = 0.05$ for memory, CPU, and energy usage. When the returned $p-$value is lesser than 0.05, we reject the null hypothesis and conclude that there is a statistically significant difference in terms of using and not using IoTVar.

Table 1 shows the results of the t-test for the synchronous mode with respect to memory, CPU, and energy. Based on the returned p-values, we reject the null hypothesis and conclude that there is a statistically significant difference for memory and energy, that is, the overhead put by the IoTVar abstraction does affect the overall performance. For CPU, we have p-values greater than 0.05 for 125, 150 and 175 variables, thus indicating that there is no statistical difference for the overhead caused by IoTVar in these cases.

Table 1. p-values from the t-test for the synchronous mode.

Metric	**Number of IoTVar Variables**							
	25	**50**	**75**	**100**	**125**	**150**	**175**	**200**
Memory	0.0142	7.45×10^{-7}	5.72×10^{-5}	1.36×10^{-7}	1.13×10^{-5}	2.45×10^{-9}	1.47×10^{-9}	0.041
CPU	4.27×10^{-13}	1.09×10^{-49}	1.08×10^{-32}	2.39×10^{-13}	0.2936	0.1964	0.1519	3.14×10^{-10}
Battery	3.43×10^{-14}	2.41×10^{-23}	5.42×10^{-7}	7.99×10^{-9}	0.0008	1.87×10^{-13}	2.47×10^{-13}	3.82×10^{-24}

Table 2 shows the results of the t-test for the publish-subscribe mode with respect to memory, CPU, and energy. We have a similar verdict as the synchronous testing. All p-values for memory and battery makes us to reject the null hypothesis, what does not happen for 50 and 125 variables with respect to CPU. Therefore, we conclude that there is statistical difference regarding memory and battery for IoTVar.

Table 2. *p*-values from the *t*-test for the publish-subscribe mode.

Metric	Number of IoTVar Variables							
	25	**50**	**75**	**100**	**125**	**150**	**175**	**200**
Memory	0.0034	0.0010	1.32×10^{-5}	0.0088	0.0004	0.0024	6.69×10^{-5}	5.36×10^{-8}
CPU	4.49×10^{-17}	0.09541	2.09×10^{-9}	1.27×10^{-12}	0.0622	0.0038	0.0244	0.0001
Battery	8.87×10^{-8}	7.06×10^{-11}	1.42×10^{-12}	6.08×10^{-9}	4.46×10^{-5}	1.08×10^{-8}	1.76×10^{-10}	1.93×10^{-11}

The results from the *t*-test indicate that application developers who choose to use IoTVar has to keep in mind that IoTVar does impact memory and battery usage. However, Figures 6 and 7 show that the memory and battery usage grow at a steady pace. The values for memory are greater due to the number of Java objects required to maintain the variable with all its functions. Although the battery usage is higher, it is close to the values when not using IoTVar, growing together with the values of the application using IoTVar.

When looking at the *p*-values for CPU, we can notice that there are five cases whose *p*-value is greater than 0.05, approximately 31% of the cases. This means that there is no significant difference in the data. We can conclude that an application using IoTVar may have measured values for CPU that are close to the measurements of an application that does not use IoTVar. Furthermore, the overhead caused by the CPU is not high (see Figure 5) and it always stays above the CPU usage of an application that does not using IoTVar, but having a difference only up to the maximum of approximately one second of CPU usage.

6. Conclusions

The heterogeneity of the overabundance of platforms in IoT presents a significant challenge to find, select, and use IoT resources, e.g., devices, sensors, services, and context data. Therefore, it is important to provide techniques that enable clients to easily discover, retrieve and use data produced by them. Due to the different types of data provided by IoT platforms and the various ways to interact with them, it is valuable to be able to gather this data at a low development cost.

This paper has presented the IoTVar middleware, which provides application developers with a way of interacting with an IoT platform using few lines of code. For this purpose, IoTVar encompasses proxies representing IoT platform virtual entities. These proxies handle the complexity of interacting with the IoT platform in both synchronous and publish-subscribe ways. Additionally, IoTVar offers a bypass for the need of understanding the IoT platform specific API and data model. In this paper, we have described the integration of IoTVar with the FIWARE European platform. The HTTP Synchronous Handler and the Publish-Subscribe Handler come up with the interesting capability of a fine-grained search provided by the FIWARE platform, thus enlarging the use cases for developers.

We have also performed an evaluation of IoTVar measured for the FIWARE platform addressing the balance between the relative cost of IoTVar for both synchronous and publish-subscribe handling of information as well as the benefits for developers. Concerning the cost, the results of statistical tests revealed that memory usage and energy consumption have significant impact on the performance. The CPU performance was accepted in 31% of the cases, thus leading us to conclude that there is no impact for IoTVar users regarding CPU usage in most situations. However, we can see that the values do not stray too far from using and not using IoTVar.

As future work, we intend to integrate IoTVar with other platforms aiming at bringing more options for application developers. We also plan to explore more the Orion API to provide more functionalities to IoTVar users, besides expanding the integration to work with other FIWARE GEs, such as the ones related to security. Finally, we need to deeply investigate the memory, CPU, and battery usage to reduce the overhead caused by IoTVar.

References

1. Zanella, A.; Bui, N.; Castellani, A.; Vangelista, L.; Zorzi, M. Internet of Things for smart cities. *IEEE Internet Things J.* **2014**, *1*, 22–32, doi:10.1109/JIOT.2014.2306328.
2. Stankovic, J. Research directions for the Internet of Things. *IEEE Internet Things J.* **2014**, *1*, 3–9, doi:10.1109/JIOT.2014.2312291.
3. Razzaque, M.; Milojevic-Jevric, M.; Palade, A.; Clarke, S. Middleware for Internet of Things: A survey. *IEEE Internet Things J.* **2016**, *3*, 70–95, doi:10.1109/JIOT.2015.2498900.
4. Ngu, A.; Gutierrez, M.; Metsis, V.; Nepal, S.; Sheng, Q. IoT middleware: A survey on issues and enabling technologies. *IEEE Internet Things J.* **2017**, *4*, 1–20, doi:10.1109/JIOT.2016.2615180.
5. Ray, P. A survey of IoT cloud platforms. *Future Comput. Inform. J.* **2016**, *1*, 35–46, doi:10.1016/j.fcij.2017.02.001.
6. Courtais, C.; Taconet, C.; Conan, D.; Chabridon, S.; Gomes, P.; Cavalcante, E.; Batista, T. *IoTVar* to transparently handle interactions between applications and IoT platforms. In Proceedings of the 4th Workshop on Middleware and Applications for the Internet of Things, Las Vegas, NV, USA, 11–15 December 2017; ACM: New York, NY, USA, 2017; pp. 7–10, doi:10.1145/3152141.3152390.
7. FIWARE. Available online: www.fiware.org (accessed on 13/11/2019).
8. Gazis, V.; Görtz, M.; Huber, M.; Leonardi, A.; Mathioudakis, K.; Wiesmaier, A.; Zeiger, F.; Vasilomanolakis, E. A survey of technologies for the internet of things. In Proceedings of the 2015 International Wireless Communications and Mobile Computing Conference, Dubrovnik, Croatia, 24–28 August 2015; pp. 1090–1095, doi:10.1109/IWCMC.2015.7289234.
9. Ahlgren, B.; Hidell, M.; Ngai, E. Internet of Things for smart cities: Interoperability and open data. *IEEE Internet Comput.* **2016**, *20*, 52–56, doi:10.1109/MIC.2016.124.
10. Bröring, A.; Schmid, S.; Schindhelm, C.; Khelil, A.; Käbisch, S.; Kramer, D.; Le Phuoc, D.; Mitic, J.; Anicic, D.; Teniente, E. Enabling IoT ecosystems through platform interoperability. *IEEE Softw.* **2017**, *34*, 54–61, doi:10.1109/MS.2017.2.
11. Ganzha, M.; Paprzycki, M.; Pawłowski, W.; Szmeja, P.; Wasielewska, K. Semantic interoperability in the Internet of Things: An overview from the INTER-IoT perspective. *J. Netw. Comput. Appl.* **2017**, *81*, 111–124, doi:10.1016/j.jnca.2016.08.007.
12. Robert, J.; Kubler, S.; Kolbe, N.; Cerioni, A.; Gastaud, E.; Främling, K. Open IoT ecosystem for enhanced interoperability in smart cities - Example of Métropole de Lyon. *Sensors* **2017**, *17*, 2849, doi:10.3390/s17122849.
13. Mynzhasova, A.; Radojicic, C.; Heinz, C.; Kölsch, J.; Grimm, C.; Rico, J.; Dickerson, K.; García-Castro, R.; Oravec, V. Drivers, standards and platforms for the IoT: Towards a digital VICINITY. In Proceedings of the 2017 Intelligent Systems Conference, London, UK, 7–8 September 2017; pp. 170–176, doi:10.1109/IntelliSys.2017.8324287.
14. Birrell, A.; Nelson, B. Implementing remote procedure calls. *ACM Trans. Comput.Syst.* **1984**, *2*, 39–59, doi:10.1145/2080.357392.
15. Shapiro, M. Structure and encapsulation in distributed systems: The Proxy Principle. In Proceedings of the 6th International Conference on Distributed Computer Systems, Cambridge, MA, USA, 13–19 May 1986; pp. 198–204.
16. Object Managment Group. Common Object Request Broker Architecture. 2012. Available online: https://www.omg.org/spec/CORBA/ (accessed on 13 November 2019).
17. Felber, P.; Guerraoui, R.; Schiper, A. The implementation of a CORBA Object Group Service. *Theory Pract. Object Syst.* **1998**, *4*, 93–105.
18. Sutra, P.; Rivière, É.; Cotes, C.; Artigas, M.; López, P.; Bernard, E.; Burns, W.; Zamarreno, G. CRESON: Callable and replicated shared objects over NoSQL. In Proceedings of the 37th IEEE International Conference on Distributed Computing Systems, Atlanta, GA, USA, 5–8 June 2017; pp. 115–128, doi:10.1109/ICDCS.2017.239.
19. FIWARE-NGSI v2 Specification. Available online: https://fiware.github.io/specifications/ngsiv2/stable/ (accessed on 13 November 2019).
20. Lehmann, E.L.; Romano, J.P. *Testing Statistical Hypotheses*, 3rd ed.; Springer: New York, NY, USA, 2005, doi:10.1007/0-387-27605-X.

21. Shapiro, S.S.; Wilk, M. An analysis of variance test for normality (complete samples). *Biometrika* **1965**, *52*, 591–611.
22. Kim, T. T test as a parametric statistic. *Korean J. Anesthesiol.* **2015**, *68*, 540–546, doi:10.4097/kjae.2015.68.6.540.