IoT Manager: an open-source IoT framework for smart cities

IoT Manager: an open-source IoT framework for smart cities

Luca Calderoni[a], Antonio Magnani[a,*], Dario Maio[a]

[a]*Department of Computer Science and Engineering, University of Bologna, Italy*

Abstract

Recent surveys concerning Internet of Things confirm that there are 20 billion connected devices and counting all around the world. As we assist to the convergence of the IoT and the cloud computing paradigms, sensor networks are being deployed everywhere and grow both in number and significance. One of the main concerns is thus to provide the community with versatile and resilient frameworks capable to store and rearrange data collected by these sensors. However, the world largest information technology companies tend to release products in a as a service fashion, avoiding to reveal the know-how concerning design and implementation details. As a consequence, a common trend for academic institutions is to use these mainstream IoT platforms as 'black boxes'. In this paper we discuss some of the most commonly adopted IoT platforms and we present IoT Manager, a general framework designed for sensor networks management which was entirely developed within the University of Bologna. Through this case study, we provide the scientific community with a detailed implementation strategy concerning our specific IoT solution. Our results are supported from a LGPL realese of the IoT Manager client in order to serve as a test bed both for research and teaching purposes.

Keywords: Internet of Things, IoT platform, Sensor networks, Software engineering, ICT architectures

1. Introduction

The Internet of Things (IoT) is now a consolidated reality of our daily lives: from cars to fitness sensors, from air conditioning systems to cameras, it is increasingly common to stumble upon devices that can communicate data with each other [1]. This vision can scale from the domestic domain to urban and regional scenarios, where sensor networks have become a common feature [2, 3, 4]. The advantages of adopting such technologies are clear, but the continuous spread of sensor networks has generated various and inconsistent environments. From the one hand, this condition poses several security threats [5, 6], from the other hand, the integration of uncorrelated sensor networks could be anything but simple [7]. We can imagine these sensor networks as pieces of a puzzle: in some cases their integration will be trivial while sometimes it could be extremely complex. Suppose we want to make the data produced by different architectures accessible in an agile way through a single compact solution. For example, imagine an integration between data derived from Santander's network[1], data produced by the new *Array Of Things* in Chicago[2] and those from the Smart Citizen platform[3]. In such a scenario we would inevitably face a number of problems, both due to the different nature of the nodes of the networks and to the different technologies and architectures adopted. The examples of sensor networks mentioned above offer a variety of sensors, as well as communication and storage protocols that are not shared. The absence of a clear design methodology that is widely adopted also makes this task rather difficult. With this principle in mind, several major have released IoT platforms that address some of the needs mentioned above. Most of these solutions are offered in a ready-to-use fashion, lacking transparency and providing limited technical information along with high-level architectures and generic communication flows [8]. This is clearly understandable in relation to business models: it would be unreasonable for a major to reveal relevant technological details and design choices adopted. Therefore, using such platforms implies a de-

*Corresponding author

Email addresses: luca.calderoni@unibo.it (Luca Calderoni), antonio.magnani@unibo.it (Antonio Magnani), dario.maio@unibo.it (Dario Maio)

[1] http://maps.smartsantander.eu/

[2] https://arrayofthings.github.io/

[3] https://smartcitizen.me/

pendency on a sort of black-box. The goal of this paper is to discuss some of the most promising IoT platforms while proposing a completely home made solution relying on open source technologies. This approach allows us to discuss design and implementation details at each layer of the stack our platform is built upon, enabling researchers and practitioners to fully understand what lies behind a IoT solution. Other academic institutions have felt the need to propose IoT platforms that could offer an under-the-hood view, for example Castellani et al. [9] have proposed a solution focused on indoor environments. Conversely, our proposal is mainly designed for outdoor environments and in particular for sensor networks distributed both in urban and suburban areas. Therefore, particular importance will be posed on the integration and interoperability between the different networks. In this context, we want to offer the possibility to monitor information from the various sensors currently considered (see Section 3.1). As proposed in [10, 11], we want to offer an IoT testbed that is useful for both academic teaching and research activities. Above all, we want to propose a platform where the citizen acts as an active component, for instance adding one or more nodes to the network in an agile way [12], or monitoring specific areas of interest through a dedicated client application.

2. IoT Platforms Comparison

Since the term IoT was coined in 1999 by Kevin Ashton during a presentation at Procter & Gamble [13], the basic idea behind IoT solutions has been widely explored by both the academic world and the ICT community. The IoT domain can be intuitively discussed as follows: let us consider a number of distributed sensors or gadgets (i.e., *"things"*) lying in an unpredictable vast environment (a house, a large urban area or a greater region). These things are able to gather a massive amount of raw data and translate them into relevant information. Typically, this ecosystem reacts proactively, minimizing (or at least trying to minimize) human involvement.

Although straightforward this scenario may appear, it hides a number of open questions. Which kind of architecture should be adopted? Which requirements are the most meaningful among others? Which communication standards should be adopted in order to enable device interoperability? What kind of API should be implemented to easily allow a sensor (or a sensor network as a whole) to join the ecosystem?

In [14], the authors propose an interesting comparison aimed at highlighting common architectural aspects of several IoT platforms and infer a reference architecture. Conversely, a comprehensive description and comparison of the main requirements (both functional and non-functional) of a IoT platform is discussed in [15].

Many platforms and solutions were proposed within the IoT domain. Each of them was designed with a business model in mind and thus holds specific features: in this work we adopt the taxonomy proposed in [16], where IoT platforms are discussed in relation of the corresponding application area.

Device Management Platforms, as defined by the Open Mobile Alliance Device Management, must guarantee the provisioning and onboarding of the devices, including remote parameterisation and real-time configuration. Again, they should allow remote firmware updates as well as a real time monitoring concerning devices faults and errors [17]. Therefore, these platforms enable a quick deployment of individual or entire groups of devices, and allow to define taxonomies and hierarchies upon them. They also allow to define access policies for different types of devices. One of the key aspects this kind of solution tend stress is the optimization of network resources. Device Management Platforms are becoming increasingly important and have consequently drawned the attention of many companies, such as Amazon, which at the end of 2017 has released the new *Amazon IoT Device Management* platform [18].

Application Development Platforms are aimed at fastening the implementation process of ICT services addressing the IoT domain. End-user applications are developed through automatic code generation and combined with a number of predefined API. One of the best-known toolkits is Temboo [19], which allows parametrization, events management and automatic code generation for a number of heterogeneous devices.

Application Enablement Platforms, as the name suggests, allow IoT architectures to integrate with preexisting external services and applications. Therefore, these solutions operate between the hardware layer consisting of sensors and actuators, and the end-user application layer. They often act as an integration middleware: devices communicate directly with the platform through transport protocols such as HTTP/S or MQTT and encapsulate data using classical data-exchange formats (XML, JSON). The integration middleware rearrange this information and delivers it to end-user applications.

The solution we discuss in the following falls into the latter category. For ease of reading, we point out that Application Enablement Platforms are often referred to as *IoT middleware*, *middleware* or *IoT middleware plat-*

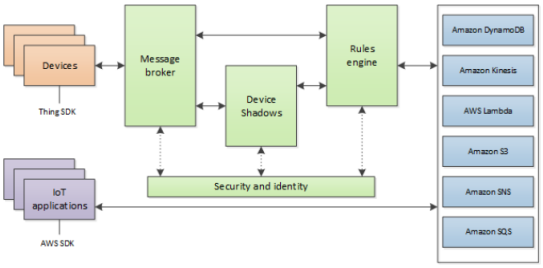

Figure 1: AWS IoT Core architecture [20].

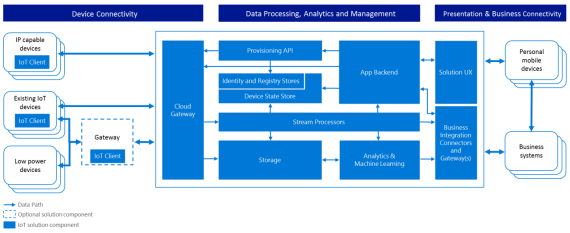

Figure 2: Microsoft IoT reference architecture [21].

form: we use these definitions interchangeably. Before introducing our solution, we discuss some of the most prominent platforms belonging to this latter category.

2.1. AWS IoT Core

Amazon Web Services (AWS) IoT Core [20] is the middleware proposed by Amazon. It consists in a cloud solution relying on a Platform as a Service (PaaS) business model. Scalability and interoperability are the most relevant features of this solution: Amazon ensures that a single IoT Core instance can support billions of devices, allowing the exchange of tens of billions of messages between AWS endpoints. The main role of AWS IoT Core is therefore to provide a reliable connection between "things" and the AWS cloud. In order to achieve this, the well-known HTTP, MQTT and WebSockets protocols are used and all communications are secured through TLS and X.509 certificates.

The platform architecture (see Figure 1) consists of four leading modules (*message broker, device shadows, rules engine, security and identity*) plus a fifth component (the *device gateway*) which is not represented in figure. This latter module connects devices to the message broker. Specifically, it exposes an incoming interface implementing the aforementioned protocols and acts as an intermediary to the message broker. The *message broker* is a publish/subscribe service that allows all devices to receive or send messages related to a specific topic they have previously registered to (e.g., Sensor/Humidity/LivingRoom). A device communicates its own status to the platform publishing a message under a proper topic. The *device shadow* service enable virtualization and persistence of each device in the cloud, allowing maintenance of the last known device state even when it is no longer online. When an object is properly connected, the status of its shadow can be updated consistently with respect to the physical device. Conversely, when the communication fails, it is still possible to interact with the device relying on its shadow. The *rules engine* module implements the business logic of the platform, making it possible to collect and process raw data. As the name suggests, the user is allowed to define rules that orchestrate the distribution of messages among other objects or AWS services. Finally, all these components interact with the *security and identity* module which is responsible of providing reciprocal authentication and encryption at all communication levels of the stack. Therefore, a two-way communication without identity assessment will never occur. This middleware holds all the benefits provided by Amazon Web Services, but, as predictable, several implementation details remain unknown.

2.2. Microsoft Azure IoT Suite

Azure IoT Suite [21] is the cloud platform developed by Microsoft. As per the AWS IoT Core, the business model is PaaS. One of the main advantages offered by this platform is the ability for users to install preconfigured solutions designed to fit common IoT scenarios. These solutions are released for free. As an example, Azure IoT Suite is equipped with a weather forecasts setting which enables data collection as well as information transmission to the middleware and its analysis through the Azure Machine Learning module. Each of these preconfigured solutions involves different devices and rely on several modules among those offered as a service by Azure and Azure IoT Hub, which are indeed the real middleware. Figure 2 shows the reference architecture of an IoT system according to Microsoft's vision: within the blue rectangle it is represented an ensemble of cloud components needed to support an IoT solution. Azure IoT Hub plays the leading role as *Cloud Gateway* technology.

Azure IoT Hub enables connection between millions of devices and a cloud based back-end, supporting bi-directional communication for AMQP, MQTT and HTTPS protocols. Features of this hub include *twin devices*, a similar solution to the AWS Device Shadow. A *twin device* consists in a JSON document in which information concerning the status of the paired device is stored. For each connected device, Azure maintains a

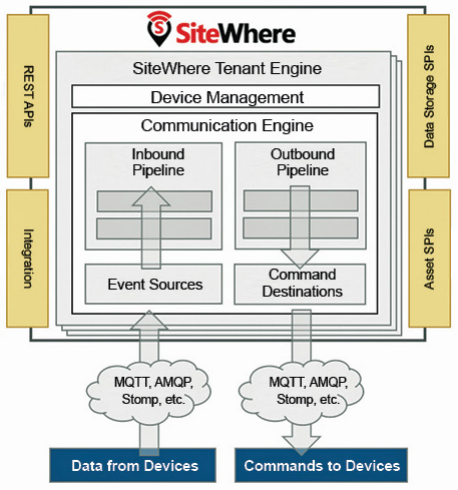

Figure 3: SiteWhere architecture [22].

twin whose information can be used by the device itself or by other applications, in order to perform device configurations or to query it for data. This feature is very helpful for batch operations. Regarding communication security, Azure Hub IoT grant access to each hub endpoint through a token-based authorization mechanism or through X.509 certificates. Such authorisations may restrict access to the hub and to some specific functionality.

2.3. SiteWhere

SiteWhere [22] differs from previously discussed middlewares primarily for its business model. It is indeed an open-source IoT platform, developed and maintained by SiteWhere. This solution is licensed under CPAL-1.0 (Common Public Attribution License Version 1.0). To be more accurate, two variants of this platform were released: a free for use *Community Edition*, and an *Enterprise Edition*, which consists in an extended paid-for version of the first. The latter solution need to be purchased directly from SiteWhere. Several are the requirements to deploy a SiteWhere instance: an Apache Tomcat web server should be configured as well as a MongoDB repository. Java and HiveMQ (a MQTT broker) are also required.

The SiteWhere server represents the central node through which it is allowed to manage both components and REST services. This solution is designed as a multi-tenant system in which tenants are responsible for most of the processing logic. Within each server, one or more tenant engines are bootstrapped, each running as a different IoT application. In order to keep the information separate, each tenant is coupled with its own data

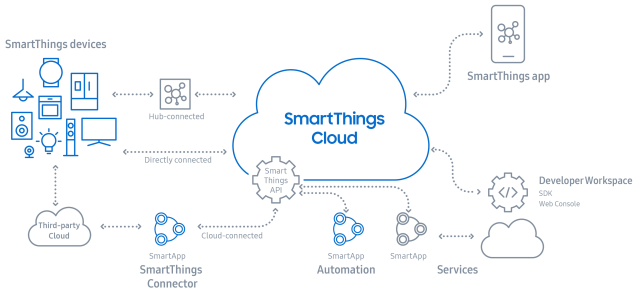

Figure 4: Samsung SmartThings architecture [23].

store. As depicted in Figure 3, every tenant also features a processing pipeline that can be customized without affecting other pipelines. Sensors send data through a gateway which operates between tenants and devices. SiteWhere supports MQTT, AMQP and REST communications.

2.4. Samsung SmartThings

SmartThings [23] is an IoT applications ecosystem. SmartThings project started in 2012 through a Kickstarter campaign. The basic idea was to realize a solution for smart domestic environments through a hub connected to a set of "things" (e.g., temperature and humidity sensors, smoke and CO alarms). As the project was started, it was coupled with a smartphone app able to communicate with the remote hub. 2014 represents a milestone for SmartThings as it was acquired by Samsung Electronics. The initial architecture has evolved considerably to become a genuinely cloud-centric platform. Indeed, as depicted in Figure 4, it is now possible to connect devices to the cloud back-end following three different strategies, even without the aid of a *direct hub connection*. Another type of connection is the *cloud-connected* one, that makes possible to implement an indirect communication channel between (cloud-based) third-party devices and the SmartThings cloud. While the presence of a hub that acts as a gateway between devices and the cloud is recommended, several operations could be performed locally, without the need to query the back-end. In this specific case, SmartThings refers to these devices as *hub-connected* and they rely on ZigBee or Z-Wave communication protocols. In SmartThings applications, objects are usually organized and grouped according to the room they are in. The *room* concept is therefore a key aspect for SmartThings clients.

SmartThings includes the concept of *automation* which allows the user to interact with the ecosystem without any manual intervention. With respect to au-

tomation, two are the possible strategies to adopt: the first relies on WebHook, the second on AWS Lambda functions. For instance, it is possible to define an automation strategy designed to adjust light intensity within a particular room according to weather changes.

This cloud solution also supports encrypted communications between all connected components through the SSL/TLS protocol [24]. Although the architecture offered by SmartThings is solidly aimed at the domestic environment or, more generally, at the *smart building* concept, its features make it possible to adopt it in broader contexts. In particular, thanks to *cloud-connections*, it would be possible to hook up a sensor network.

3. IoT Manager

In [25], Calderoni et al. proposed a general ICT architecture designed to manage several subsystems in urban contexts. *IoT Manager* represents an evolution of this model and implements its main features. In this section, we want to clearly explain how our platform was designed and implemented, in order to provide the scientific community and practioners with a tangible example of a fully open IoT stack. As pointed out previously, an IoT Player does not frequently reveal details about its solution. In addition to this, it is increasingly common to see platforms that are not supported by exhaustive details about the connection of sensors. Discussion typically focuses on the IoT middleware layer, its infrastructure and the services offered. However, this leads to neglect some relevant details concerning on the one hand the physical component that has to communicate with the middleware, and on the other hand the possible application component. In this section we want to face this discussion in its entirety: through a top-down approach we will analyze *IoT Manager* not only discussing the role of integration middleware but also describing the physical and application layers of the stack. Indeed, this will allow us to describe the sensors used in our case study (see Section 4), make a comparison between our proposed middleware and those introduced in Section 2 and illustrate a client application connected to the platform.

From a high-level architectural point of view (see Figure 5 for reference), this system is composed of three layers, similarly as discussed in [26].

The **sensing layer** consists of a number of heterogeneous sensor networks. These networks can be distributed anywhere in the globe and their purpose is to collect raw data. In addition, the platform is fully geo-referenced, allowing application-level filtering based on

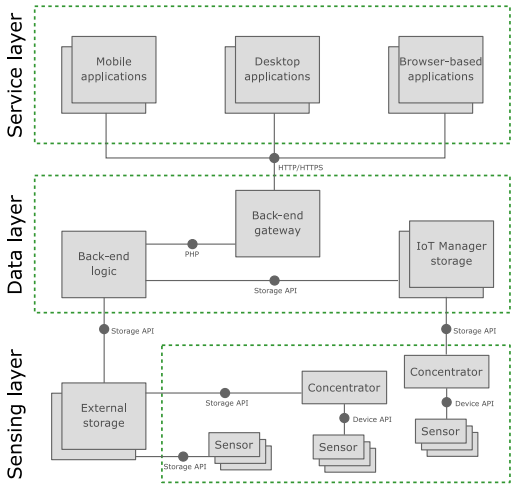

Figure 5: A high-level diagram showing IoT Manager architecture.

sensors effective location. Range queries may be also addressed in relation with the user's current position (as an example consider a user who wants to check the air quality in his surroundings). The geo-referencing of the sensors does not offer guarantees regarding the logical division of the sensors of interest. Therefore, IoT Manager is designed to natively support sensors with a two-tier taxonomy. In detail, in the sensing layer a network node can be treated either as a *simple sensor* or as a *concentrator*. In this second case, the purpose is to represent a logical set of different simple sensors. Thanks to the two-level taxonomy, the *back-end gateway* allows for requests which only address the set of simple sensors connected to a given concentrator. Raw data from sensors and concentrators are sent to the middleware via APIs that depend on the storage engine adopted. IoT Manager currently has its own internal storage, but through a set of predefined APIs it is possible to integrate data from sensor networks whose storage is external to the back-end. This is another key aspect of our solution: it is possible for third parties, such as a citizen, to connect their own sensor or sensor network. Within Section 3.1 we describe some sensors which are already handled by IoT Manager and we also detail the procedure used to build one of this devices from scratch.

The **data layer** represents the back-end of the system and is responsible for two main features: on the one hand, it serves as a repository for all of the sensed information, on the other hand, it provides several API which may be called by client applications in order to query those data and retrieve them in a properly arranged format. As we have seen, this level plays the key role of maintaining compatibility between the various subsys-

tems. It also provides the application layer with an efficient and transparent way to access data. The role of integration middleware is covered by the *back-end logic* module (see Figure 5) that represents the more sophisticated component of the system.

Specifically, this component is able to retrieve raw data produced by sensors and concentrators using a set of predefined APIs which allow it to query different storage engines. Furthermore, information recovery is empowered both for internal and external storages. Albeit raw data might be retrieved from a wide range of different repositories, the back-end logic can revise these records in order with the goal for them to conform to a particular format, in accordance with the *back-end gateway* dispositions. The back-end gateway is another key component of this layer. It exposes HTTP/HTTPS APIs to enable communication with client applications. It is also responsible for requests translation (in a set of jobs handled by the back-end logic component) and for final response formatting (JSON). An in-depth discussion about the back-end gateway and the back-end logic is provided in Section 3.2.

The **service layer** offers users a wide range of possible client applications that communicate with the back-end gateway through appropriate APIs. These APIs are currently based on HTTP and HTTPs protocols, which makes integration with desired user application quite simple. Within Section 3.3 we provide a detailed design of one of these client applications, which has been developed for Android mobile devices. Clients are subject to a specific access policy and handle geo-referenced data.

3.1. Sensing layer: some examples

Our solution deals with different types of sensors, one of which consists in a low-cost weather station. This prototype relies on a UDOO Neo Extended board. This board is equipped with a NXP i.MX 6SoloX processor with two different core: an ARM Cortex-A9 and a Cortex-M4 (an Arduino UNO-compatible platform). In addition, it is provided with 1GB RAM, a Bluetooth 4.0 receiver, a Wi-Fi module and 9 integrated sensors (3-Axis accelerometer, magnetometer, gyroscope) which were not considered in our case study. Finally, there is an I^2C (Inter Integrated Circuit) connector used to plug sensor modules (UDOO bricks). One of the main features concerning UDOO bricks is the ability to work through a cascade configuration: it is allowed to connect several sensor modules using the sole I^2C interface on the board. Of course, it is also allowed to connect sensors directly to the Arduino socket provided by the board [27].

In our experiment, we used three different sensor modules: a *Barometer brick* (based on MPL3115) that is able to sense pressure (*hPa*) and temperature ($^{\circ}C$), a *Light brick* (based on TSL2561T) that returns illuminance (*Lux*) ambient values and a *Humidity brick* (based on SI7006-A20) providing relative humidity percentage.

We have developed a simple bash script which allowed us to read data from the barometer. Conversely, as part of our implementation relies on external libraries, other bricks were handled through an Arduino sketch. Data received from each sensor are collected by the UDOO operating system and then sent to an external storage via HTTP/S API. In order to comply with IoT Manager specifications, the payload also includes some mandatory information (*sensor identifier*, *sensor name*, *subsystem identifier*, *status*, *latitude* and *longitude*). These fields are introduced in Section 3.2.

Besides weather stations, the sensing layer is composed of a number of other devices such as *ArLu* and *Lamps*. An ArLu, representing a lighting cabinet, acts as a *concentrator* and is logically connected to a set of simple sensors (Lamps) allowing a full lighting system management. This two-level taxonomy enables a logical partition even when ArLu and Lamps are not physically connected one each other.

3.2. Data layer: the back-end logic

When a client application queries the back-end for data, the data layer acts as outlined in Figure 6.

The client application delivers a request over a HTTP/HTTPS post channel. The web server, implementing the back-end gateway, handles this request and, first of all, checks for user authentication. This operation is performed by the *AuthManager* class, a specific software component which addresses authentication queries to the central IoT Manager storage. Thanks to a complete integration with prepared statements, this module preserves the framework from being affected by SQL injection. On authentication granted, the back-end gateway instantiates a *JobManager*: this module checks for the type of the handled request and instantiates in turn a *Mapper* object in order to retrieve data. The set of job types supported by our framework along with each request parameter are reported in Table 1.

While jobs 3, 4, 5 and 7 depend on meta data and affect IoT Manager storage only, jobs 1, 2 and 6 may also affect a number of external storages. In fact, as previously discussed (see Figure 5 for reference), our framework is able to retrieve raw data both from its own storage and from a number of external sources. As we

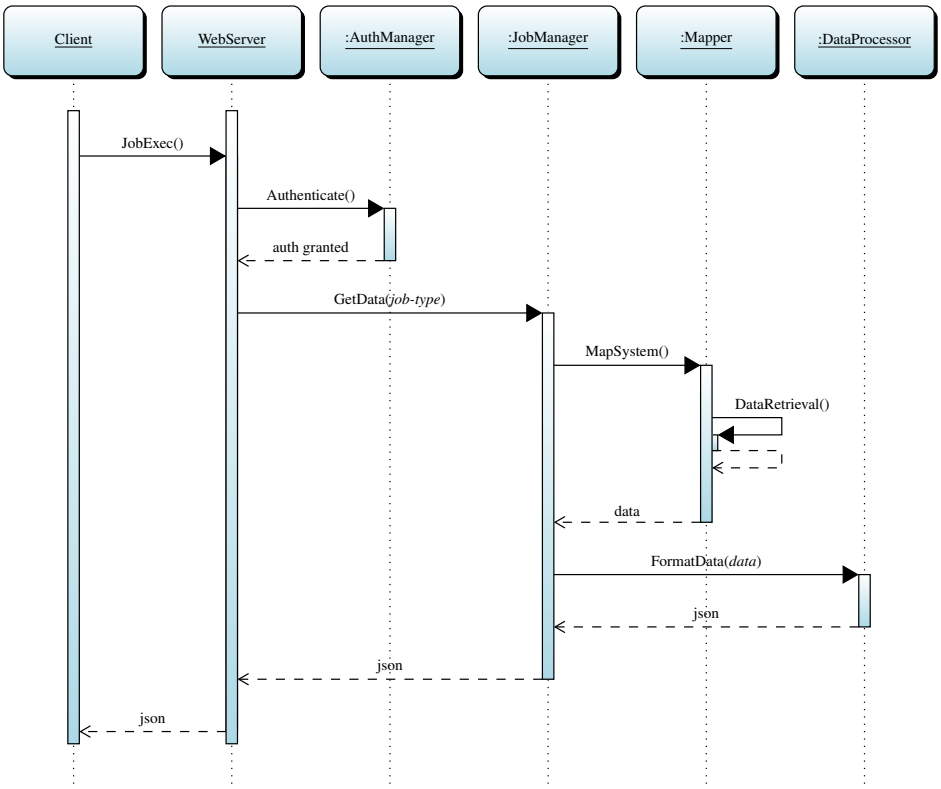

Figure 6: IoT Manager requests processing from the back-end perspective.

may notice, each of these jobs is completely transparent with respect to the calling application concerning real data location. Thanks to a set of back-end APIs, the Mapper object connects to each subsystem and retrieves each relevant record. Desired records are thus collected by the JobManager object and prepared for being returned to the calling application by the *DataProcessor* (see Figure 6 for reference). The latter class is responsible for data formatting in compliance with the service contract through JSON notation.

Our back-end logic thus relies on several APIs for data retrieval. It is important here to point out that each retrieved record may belong to a separate subsystem, each holding specific features. As a consequence, data may contain a large number of heterogeneous attributes. This is the reason why we defined a restricted set of attributes which subsystems need to exhibit as a mandatory requirement for them to be connected to IoT Manager. Specifically, these attributes shall represent a *sensor identifier* (unique within its own subsystem), a *sensor name* (or description), the *identifier of the subsystem* they belong to, a *status* information and a couple of fields specifying the *longitude* and *latitude* coordinates of the sensor. It is meaningful to note that these data do not need to be stored under a single or predefined column name. For each external source, the Mapper queries IoT Manager meta data in order to know which column or columns contain each mandatory information and which names represent those columns within the external storage schema. This mapping feature provided by the back-end logic allows for a proper implementation of jobs 1 and 6 which, as should be noticed, produce a list of compliant information derived from heterogeneous subsystems. This allows client applications to easily handle sensor lists throughout each part of the user interface where sensor-specific details are not required. Conversely, when a calling application would require something specific about a single sensor, a different mapping principle applies. This is indeed the case of job 2. The back-end logic access the aforementioned meta data and search for column mapping concerning sensor and subsystem identifiers. Through the proper connection API, the Mapper queries target storage for each data related to the sensor and blindly collect them. Sensor-specific data are then JSON formatted and returned through the HTTP service in a key-value fashion. The calling application is thus responsible for data interpretation. In order to build a proper user in-

Table 1: IoT Manager input parameters and job types derived from the HTTP service contract exposed by the back-end gateway.

Parameter	Description
user, pwd	Username and password for authentication.
filter	Susbsystem identifier (0: all subsystems).
id	Single sensor or single city identifier, depending on the job.
minLon, maxLon	Longitude bounding values.
minLat, maxLat	Latitude bounding values.
job	Job identifier, as outlined below.
Job	Description
1	Returns a list of sensors lying within a specific bounding box specified by the calling application. Depending on the *filter* parameter, it is possible to address this request to a specific subsystem (a specific set of sensors) or to each subsystem.
return	[*id, name, subsystem, longitude, latitude, status*]:list
2	Returns a single sensor and all of its related information in a key-value fashion. The identity of the sensor is provided in the request through the couple *subsystem, id*.
return	[*attribute name, value*]:list
3	Returns the list of subsystems handled by IoT Manager.
return	[*subsystem, name*]:list
4	Returns the list of known cities in the back-end atlas.
return	[*city, name*]:list
5	Returns a single city and all of its related information.
return	*city, nation, name, longitude, latitude, gmt*
6	Returns the list of sensors connected to a specific concentrator (uniquely identified by *subsystem, id*).
return	[*id, name, subsystem, longitude, latitude, status*]:list
7	Returns a key-value list exposing a semantic description of each attribute for each specific subsystem.
return	[*attribute name, description*]:list

terface and to correctly show meaningful data, end-user application developers may rely on job 7, which provide the client with a human readable description of each returned field. Finally, a couple of words about georeferencing. IoT Manager natively supports positional data. Mobile services built against the IoT Manager framework may use GPS coordinates to enrich their queries with bounding box information. However, when a client application is not aware of its location, or when the hardware it is executed on is not equipped with any form of location sensing device, job 4 and 5 may be used to simulate user's position as derived from the framework atlas.

3.3. Service layer: a client application

As previously discussed (see Figure 5 for reference), the service layer is an ensemble of applications designed to interact with IoT Manager data layer. Within this section we explore this layer through a real application which was designed and implemented by our research

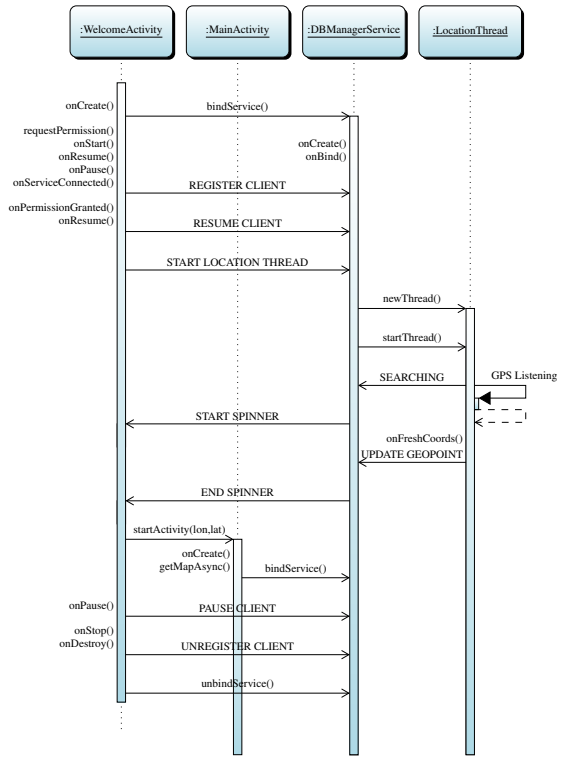

Figure 7: Launch sequence (Android API level ≥ 23).

team. The service we are about to discuss consists of a mobile application built over Android OS.

The main aim of this application is to sense location information through GPS and network hardware and to display sensors which lie within a given distance with respect to the device itself. The user is also allowed to displace his position using one of those provided by the back-end atlas. As this application is intended to be used in the IoT domain, it is designed with multithreading and asynchronism in mind. Each client *activity* relies on a shared Android service in order to obtain positional data as well as each kind of external data to be downloaded through IoT Manager HTTP API. The launch sequence of our application is described in Figure 7.

As we may see, a *welcome activity* is initially started along with a *service* running on a separate thread. The activity first checks for user permission concerning GPS and Network and, on permission granted, asks the service for location coordinates. The service then starts a dedicated thread which implements several primitives provided by Android OS able to deal with GPS and Network sensing. When a fresh position is sensed, the location thread sends a message to the service, which in

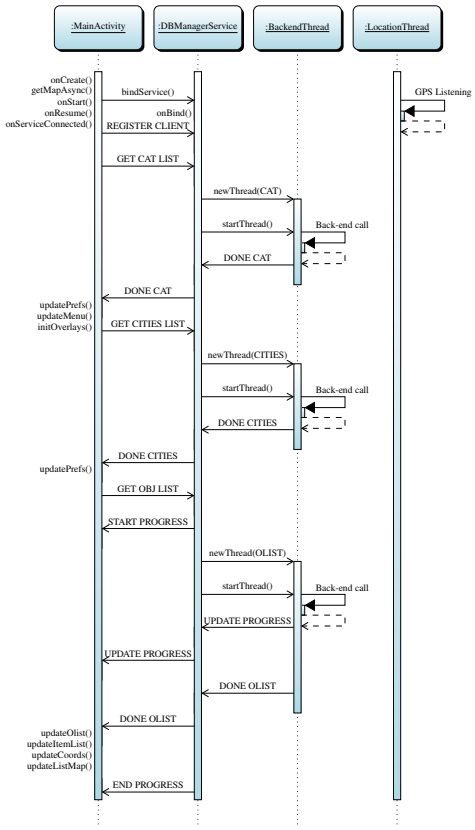

Figure 8: Main activity starting sequence. Here we assume login information has been already filled in the application preferences and no specific city from the back-end atlas was selected instead.

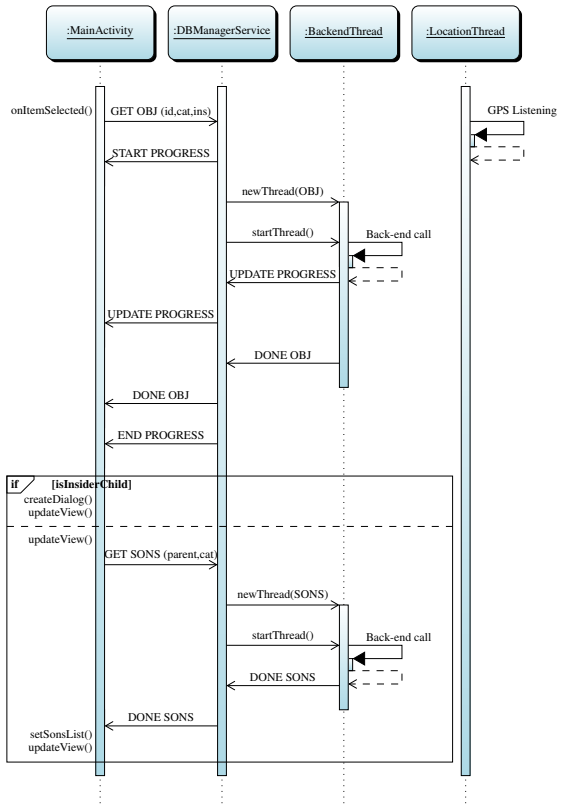

Figure 9: Sensor details request. The collected information replaces the map layout container or, when the request arises from that container itself (*isInsiderChild* = *true*), it is shown in a dedicated dialog.

turn sends these new coordinates to each connected activity. As the welcome activity receives coordinates, the program control passes to the *main activity* which immediately binds to the service. The main activity first checks for authentication information within the application preferences. Figure 8 shows its starting sequence assuming these credentials were already provided by the user.

While the Android service and the location service keep on running on their own threads, this sequence diagram shows a new type of thread which has been designed to handle back-end calls. During its starting sequence, main activity asks the service for a number of external data. For each task, the service instantiates a single thread implementing the IoT Manager communication service and propagates the request to the endpoint through HTTP/S. Specifically, it first asks for the complete list of subsystems handled by the back-end (Table 1, job n.3). Then it requests the list of cities stored in the back-end atlas, used to populate a specific combo box within application preferences (Table 1, job n.4). Finally, it asks for a list of sensors (belonging to any subsystem) which lie within a predefined range from the user (Table 1, job n.1).

As we may see, each back-end call is handled by a specific thread and does not affect the application responsiveness at all. Please note that each *back-end call* depicted in Figure 8 may be exploded with the sequence diagram provided in Figure 6. When these calls are completed and information is returned to the calling activity, the application GUI is updated with sensors data. A sorted list of sensors (with respect to the distance to the user) is populated on the left, while a map showing an overlay icon for each sensor is proposed on the right. It is meaningful to point out that, at this stage, no detailed sensor information is required. In order to populate list and map it is enough to know few basic information as those returned by job 1 or 6 (see Table 1 for reference). Consequently, our GUI is subsystem-independent and is able to deal with heterogeneous sensors with no need for specific personalization. Processing of sensors list and map relies on a specific class

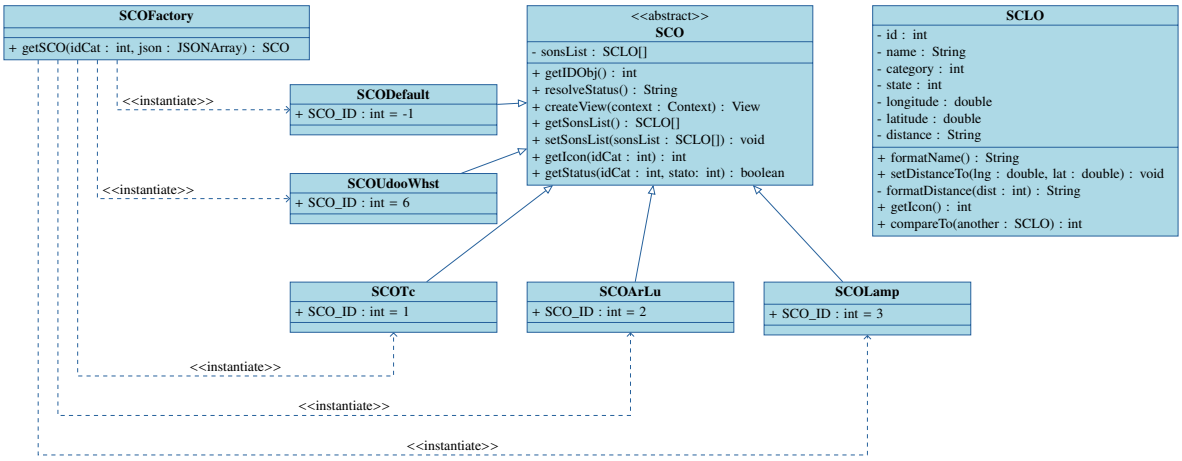

Figure 10: The Android client class factory, the abstract class representing a single sensor and some of its specializations.

called *SCLO* (see Figure 10). The role of this class is to store basic sensor information for those devices included in the current bounding box. Such information constitutes the instances objects of the *SCLO* class. As we may see in the class diagram depicted in Figure 10, the SCLO class expose a number of methods. Among them, it is meaningful to underline those dealing with distance evaluation with respect to the user's position. These methods and fields are relevant as they enable location-based filtering and sorting. Again, it is important to note that the abstract class *SCO* includes the *sonsList* field, consisting of an array of *SCLO*. It also contains the related overloads of the *getSonsList* method. As we may see, each instance of the *SCO* class keep all the information related to sensor's sons in a compact and interoperable fashion via the *SCLO* class.

When the user clicks or taps on a specific sensor, a request for sensor's details is propagated to the back-end, as depicted in Figure 9.

This sequence implements the call for job n.2 (see Table 1 for reference). When the download process terminates and the information is delivered to the main activity, the GUI is properly updated. Again, as the given sensor could be a concentrator, another back-end call (job n.6) is propagated in order to show the list of related sensors. Conversely as per the sequence proposed in Figure 8, the information to be shown is sensor-specific and, thus, a specific layout needs to be designed to arrange it. Our Android client is conveniently designed to this purpose and it is provided with a *class factory* which instantiates the proper object on a subsystem basis. A simple class diagram showing some specializations of the abstract class implementing a single sensor is provided in Figure 10.

Each sensor class need to specialize an abstract method *createView()*. This method should contain those instructions used to render a proper layout for the sensor. Consequently, when we need to show some sensor-specific detail within the GUI, it is sufficient for us to call this method on the object representing the given sensor, without any other knowledge about its features.

4. Case study

The open source framework IoT Manager was firstly designed and introduced to reflect the needs of several partners of the University of Bologna in a smart city scenario. As each partner possessed a different, separate sensor network, the main goal was to allow these networks to join the middleware without any ad hoc intervention. This challenge represented an excellent case study for both industrial and research purposes, and positively contributed to the platform implementation process.

Currently, IoT Manager involves four different types of sensors: in addition to the already mentioned ArLu, Lamp and Weather Station (see Section 3.1) a sensor called *Traffic Controller* (TC) is also handled. This sensor is based on a smart camera that continuously monitors a road section using some virtual spires placed on

Table 2: Geographical distribution and quantification of the various types of sensors currently involved in our case study.

Sensor	Quantity	Distribution
ArLu	$\simeq 50$	Europe
Lamp	$\simeq 500$	Europe
Traffic Controller	$\simeq 30$	Europe and Morocco
Weather Station	$\simeq 10$	Italy

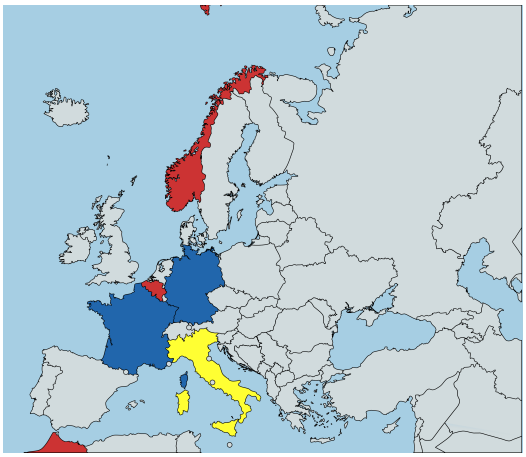

Figure 11: Distribution of the various types of sensors that are part of the IoT Manager sensing layer. In red, those countries in which TC sensors are deployed. In dark blue, those countries involving ArLu and Lamp. Italy (yellow) is the only nation where all of the currently handled sensors have been deployed.

the lanes. The TC is responsible for counting, classifying and estimating the speed of vehicles crossing the virtual coils that are placed in strategic points of the roadway. Although the number of sensors is not very high, they are widely spread across the European continent (see Table 2 and Figure 11). Data collected by these sensors were derived from an agglomeration of corporate databases and research outcomes as the result of a number of collaborations between the University of Bologna and other institutions.

Each among the aforementioned sensors belongs to a different network involved in some kind of outdoor urban sensing. Specifically, the Traffic Controllers sensor network provides traffic monitoring information upon several major arterial roads in different European countries. Conversely, the network comprising ArLus and Lamps is used for public lighting management and is mainly deployed in Italy, France, and Germany. Finally, weather stations are part of a prototype network deployed in Italy solely, and they are designed for air quality and weather conditions monitoring. As discussed above, these sensor networks were already operative, and belong to different companies. Therefore, they are part of different and separate systems and they store raw data on separate remote data bases. Thanks to IoT Manager, we were able to harmonize these networks transparently. While they still collect data in each respective storage system, IoT Manager is able to retrieve each data on the middleware and offers a unified application for an easier sensor network management.

It is finally important to stress that IoT Manager is designed with research and teaching purposes in mind. We released an open distribution of the client application introduced in Section 3 on *GitHub*[4]. This approach allows students and researchers to synchronize their IDE with IoT Manager's repository and to develop their own IoT solutions against the framework.

5. Discussion and future improvements

IoT Manager's goal is twofold: first, to provide researchers and practitioners with a full-stack platform that enables rapid deployment of prototype IoT solutions; second, to provide guidance at all architectural levels for the production of open-source IoT layers/platforms. Commercial solutions presented in Section 2 offer a typically partial or compartmentalized view. A rather evident lack is the absence of operational details concerning the application layer. A full-stack solution, as IoT Manager represents, could be useful for research groups to understand how to build an IoT platform from scratch and to quickly hook up sensor networks. IoT Manager also help the designer in the customization of the client application which needs to be implemented according to the requirements of specific application contexts. This feature is usually provided by *Application Development Platform* (which fall outside of the scope of this work) while it is rarely adopted by *Application Enablement Platform*, which focus is posed on the middleware (see Section 2 for details).

Therefore, as stressed before, the main features of IoT Manager are *(i)* its interoperability and *(ii)* its full-stack architecture. Concerning *(i)*, as we have seen in the Section 4, the proposed framework allows the rapid coupling of entire networks of sensors, even for those which already operate. The only constraint is the existence of the six mandatory information (*sensor identifier*, *sensor name*, *subsystem identifier*, *status*, *latitude*, and *longitude*), as discussed in Section 3.2. In specific application contexts, this feature makes IoT Manager's interoperability more agile than its commercial counterparts, which often require the creation of an ad hoc digital twin (e.g., AWS IoT Core) for each connected device. In relation to *(ii)*, IoT Manager is combined with a complete end-user application framework which enables to quickly define the taxonomy of the different types of sensors involved in a project. The client application manages this taxonomy with a class factory design pattern. This feature allows the rapid rendering of customized graphical interfaces, potentially relying on

[4] https://github.com/smartcitylabunibo

those which are already provided on GitHub. Besides, although IoT Manager is designed for smart city contexts, the presence of hierarchies makes it possible to adopt it in several other contexts, as the home automation one or, more generally, in smart buildings. Again, it is allowable to use sensor hierarchies to define groups of sensors belonging to the same place. For instance, the introduction of a hierarchical subdivision by rooms may reflect the sensors partitioning provided by Samsung SmartThings.

Our research and teaching team is constantly working on IoT Manager platform. Several modules were implemented during the recent years in order to expand the data layer capabilities as well as to extend the set of subsystems handled by the framework. Several efforts have been also carried out in order to improve the service layer. As one of the main concern of IoT Manager is interoperability, we will devote our attention to the platform's APIs. Two are the main challenges with respect to this subject: first, a wider set of communication protocols should be exposed by the back-end gateway. As an example, several IoT platforms accept connection from MQTT or WebSockets protocols, which are not handled by our middleware at the moment. Second, the back-end mapper should be provided with a wider set of external storage engine APIs. This condition would indeed lead to an easier connection of pre-existing subsystems. Specific attention should be posed on NoSQL databases and column-based storage engines. We are currently working on an additional module located between the *Sensing Layer* and the *Data Layer* (see Figure 5) in order to enhance our three-layered stack. The mission of this module is to act as a dispatcher between sensors and the back-end allowing a two-way message exchange. The dispatcher should be combined with appropriate APIs for sensors connection. Implementing this component would enable a publish/subscribe paradigm similarly as discussed for AWS IoT Core (see Section 2.1) and represents one of the most insightful challenges of the IoT Manager project.

6. Conclusion

In this paper we introduced IoT Manager, a full stack IoT platform relying on open source technologies. We discussed our platform in accordance with several mainstream IoT middlewares provided by well-known companies. Throughout the first part of this work we emphasised several common patterns which may be found in commercial platforms while, in the second one, we discussed our own solution with respect to these reference architectures. As a lot of research and teaching projects within this field rely on hidden details which private companies do not tend to unveil, our main aim was to provide the scientific community with a tangible implementation of such a solution, along with a detailed description of our design strategies at each level of the stack.

References

[1] K. S. Yeo, M. C. Chian, T. C. W. Ng, A. Do, Internet of things: Trends, challenges and applications, in: 2014 International Symposium on Integrated Circuits (ISIC), Singapore, December 10-12, 2014, IEEE, 2014, pp. 568–571.

[2] L. Atzori, A. Iera, G. Morabito, The internet of things: A survey, Computer Networks 54 (15) (2010) 2787–2805.

[3] P. Bellavista, G. Cardone, A. Corradi, L. Foschini, Convergence of manet and wsn in iot urban scenarios, IEEE Sensors Journal 13 (10) (2013) 3558–3567.

[4] J. Jin, J. Gubbi, S. Marusic, M. Palaniswami, An information framework for creating a smart city through internet of things, IEEE Internet of Things Journal 1 (2) (2014) 112–121.

[5] M. Conti, A. Dehghantanha, K. Franke, S. Watson, Internet of things security and forensics: Challenges and opportunities, Future Generation Comp. Syst. 78 (2018) 544–546.

[6] P. Palmieri, L. Calderoni, D. Maio, Private inter-network routing for wireless sensor networks and the internet of things, in: Proceedings of the Computing Frontiers Conference, CF'17, Siena, Italy, May 15-17, 2017, ACM, 2017, pp. 396–401.

[7] D. Partynski, S. G. M. Koo, Integration of smart sensor networks into internet of things: Challenges and applications, in: 2013 IEEE International Conference on Green Computing and Communications (GreenCom) and IEEE Internet of Things (iThings) and IEEE Cyber, Physical and Social Computing (CPSCom), Beijing, China, August 20-23, 2013, IEEE, 2013, pp. 1162–1167.

[8] P. P. Ray, A survey of iot cloud platforms, Future Computing and Informatics Journal 1 (1) (2016) 35 – 46.

[9] A. P. Castellani, N. Bui, P. Casari, M. Rossi, Z. Shelby, M. Zorzi, Architecture and protocols for the internet of things: A case study, in: Eigth Annual IEEE International Conference on Pervasive Computing and Communications, PerCom 2010, March 29 - April 2, 2010, Mannheim, Germany, Workshop Proceedings, IEEE, 2010, pp. 678–683.

[10] S. Latré, P. Leroux, T. Coenen, B. Braem, P. Ballon, P. Demeester, City of things: An integrated and multi-technology testbed for iot smart city experiments, in: IEEE International Smart Cities Conference, ISC2 2016, Trento, Italy, September 12-15, 2016, IEEE, 2016, pp. 1–8.

[11] A. L. Chan, G. G. Chua, D. Z. L. Chua, S. Guo, P. M. C. Lim, M. Mak, W. S. Ng, Practical experience with smart cities platform design, in: 4th IEEE World Forum on Internet of Things, WF-IoT 2018, Singapore, February 5-8, 2018, IEEE, 2018, pp. 470–475.

[12] J. Gubbi, R. Buyya, S. Marusic, M. Palaniswami, Internet of things (iot): A vision, architectural elements, and future directions, Future Generation Comp. Syst. 29 (7) (2013) 1645–1660.

[13] K. Ashton, et al., That 'internet of things' thing, RFID journal 22 (7) (2009) 97–114.

[14] J. Guth, U. Breitenbücher, M. Falkenthal, F. Leymann, L. Reinfurt, Comparison of iot platform architectures: A field study based on a reference architecture, in: 2016 Cloudification of the Internet of Things, CIoT 2016, Paris, France, November 23-25, 2016, IEEE, 2016, pp. 1–6.

[15] M. A. Razzaque, M. Milojevic-Jevric, A. Palade, S. Clarke, Middleware for internet of things: A survey, IEEE Internet of Things Journal 3 (1) (2016) 70–95.
[16] M. A. A. da Cruz, J. J. P. C. Rodrigues, J. Al-Muhtadi, V. Korotaev, V. H. C. Albuquerque, A reference model for internet of things middleware, IEEE Internet of Things Journal (2018) To appear.
[17] Open Mobile Alliance, Oma device management tree and description, Tech. rep., Open Mobile Alliance Ltd. (2012).
[18] Amazon web service, "aws iot device management", https://aws.amazon.com/iot-device-management/, accessed: 2018-10-02.
[19] Temboo inc, "temboo", https://temboo.com/, accessed: 2018-10-02.
[20] Amazon web service, "aws iot core", https://aws.amazon.com/iot-core/, accessed: 2018-10-02.
[21] Microsoft azure iot, "azure iot suite", https://www.microsoft.com/en-us/internet-of-things/azure-iot-suite/, accessed: 2018-10-02.
[22] Sitewhere, "sitewhere - the open platform for the internet of things", https://www.sitewhere.org/, accessed: 2018-10-02.
[23] Samsung smartthings, "smartthings - the smartthings ecosystem", https://smartthings.developer.samsung.com/develop/index.html, accessed: 2019-03-07.
[24] M. Ammar, G. Russello, B. Crispo, Internet of things: A survey on the security of iot frameworks, J. Inf. Sec. Appl. 38 (2018) 8–27.
[25] L. Calderoni, D. Maio, S. Rovis, Deploying a network of smart cameras for traffic monitoring on a "city kernel", Expert Syst. Appl. 41 (2) (2014) 502–507.
[26] L. Calderoni, D. Maio, P. Palmieri, Location-aware mobile services for a smart city: Design, implementation and deployment, JTAER 7 (3).
[27] G. Borrello, E. Salvato, G. Gugliandolo, Z. Marinkovic, N. Donato, Udoo-based environmental monitoring system, in: A. D. Gloria (Ed.), Applications in Electronics Pervading Industry, Environment and Society - APPLEPIES 2015, Rome, Italy, May 5-6, 2015, Vol. 409 of Lecture Notes in Electrical Engineering, Springer, 2015, pp. 175–180.

Luca Calderoni received a Ph.D. degree in computer science from the University of Bologna, Italy, in 2015. He is currently a Post-doctoral Researcher with the Department of Computer Science and Engineering of the University of Bologna, in Cesena, Italy. His research activity focuses on security, privacy, probabilistic data structures and urban infrastructures. He has published on location privacy, border controls, secure and privacy-preserving tracking and monitoring technologies, location-aware applications and urban ICT infrastructures.

Antonio Magnani received the Master's degree in Computer Science and Engineering from the University of Bologna, Italy in 2016. He is currently working toward the Ph.D. degree in Computer Science and Engineering at University of Bologna, Italy. His research interests include computer vision solutions for Ambient Intelligence, Internet of Things and Smart Cities.

Dario Maio received a Master's degree in electronic engineering from the University of Bologna, Italy in 1975. He is a Full Professor of Information Systems with the Department of Computer Science and Engineering, University of Bologna. He is a member of IEEE, ACM and IAPR. He was the Chair of the Cesena Campus (2001-2007), and is the Director of the BioLab and the Coordinator of the Smart City Lab with the University of Bologna. He has published more than 200 research papers investigating various aspects of computer science including distributed computer systems, computer performance evaluation, database design, information systems, neural networks, autonomous agents, pattern recognition, and biometric systems.