NETIoT: a versatile IoT platform integrating sensors and applications

Tudor Rogojanu*, Mihai Ghita*, Valeriu Stanciu*,
Radu-Ioan Ciobanu*†, Radu-Corneliu Marin*†, Florin Pop*†, Ciprian Dobre*†

*University Politehnica of Bucharest, Bucharest, Romania
E-mails: {tudor.rogojanu, mihai.ghita, valeriu.stanciu}@cti.pub.ro
†SmartRDI - Smart Research, Development and Innovation, Bucharest, Romania
E-mails: {radu.ciobanu, radu.marin, florin.pop, ciprian.dobre}@smartrdi.net

***Abstract*—In the landscape of IoT platforms there are many solutions working at various levels for collecting data and supporting specialized set of IoT services. In general, the responsibility for building the application to use the data as such collected is left external, on the hand of a developer working with a set of API made available by the IoT platform. In this paper we present an alternative approach, where the design of the platform is centered around the applications being supported by a particular set of data coming from IoT devices. With NETIoT, we support modular applications (similar to how Web mash-ups are constructed) that integrate a collection of reach functions, from data collections in IoT technology, to queuing and processing, to integrated data visualization.**

***Keywords*-IoT; platform; PaaS; microservices; monitoring; data handling.**

I. Introduction

In our days, the Internet of Things (IoT) has become for many of us a new image of how the future will be in the upcoming years, since, each of the past few years, the number of devices that can have Internet access has been consistently increasing. More and more devices and sensors now have access to the Internet, can be controlled remotely and offer a wide variety of data types that can be used to make amazing things.

IoT Platform-as-a-Service (PaaS) providers ensure that all data collected by sensors or other similar devices is received and sent to other services where it can be stored, viewed, analyzed and used to generate a response for other devices, in a highly available, scalable and secure way. The providers also offer software developments kits (SDKs) that help developers to easily and quickly connect hardware devices to their platform. In this paper we present the design considerations for building a PaaS platform that puts together applications with IoT data being collected, offering guarantees for scalability and reliability.

The paper is structured as follows. In Section II we make an analysis of today's IoT platforms. This is followed in Section III by an overview of NETIoT, and in Section IV by several design details. In Section V we illustrate a case study. Finally, Section VI concludes the paper.

II. State of the Art

Nowadays, there are many suppliers on the market that provide IoT PaaS services, such as Amazon, Microsoft Azure, Google Cloud Platform, or IBM Watson Internet of Things, but those bring some big disadvantages. All clients that want to use their services must have knowledge to create their own service that analyzes received data and use proper hardware that will send required data.

Amazon, Azure, IBM and Google use the following components: message broker, rule engine module, security and identity module, a module that knows the state of sensors or connected devices. Each provider supports bidirectional communication between hardware device and platform, but are implemented differently. Amazon uses message queue to send messages to a device that is subscribed to a certain topic. On the other hand, Azure provides two endpoints that are used to send and receive data. All mentioned platforms uses HTTP and MQTT protocols. Also another important aspect of IoT platforms are given by the SDK languages support. IBM Watson IoT and Azure offer SDKs for Java, C#, Python, NodeJS and C, AWS offer support only for C and NodeJS.

Kaa IoT (http://www.kaaproject.org) is highly flexible, multi-purpose, 100% open-source middleware platform that offers similar features like AWS IoT, Azure Iot. The main advantages of Kaa IoT is given by the possibility to be hosted in private, hybrid or public cloud. Kaa provide possibility to store data on Apache Cassandra and MongoDB, depending on client needs.

III. NETIoT

To overcome all disadvantages of IoT PaaS providers nowadays, we propose and develop a new type of service that can provide PaaS, but also SaaS (Software-as-a-Service). The main goal is to ease end-users to use IoT, through the NETIoT platform. Compared to other IoT PaaS providers, our solution provides the following features:

- *hardware devices* (such as sensors and device gateways) that are already compatible with our solution and are easy to configure and install in the right location with client needs;

- already *built services* that can be used in specific use cases with our hardware devices;
- the *ability to add new services* that will respond to other uses.

NETIoT's architecture is depicted in Fig. 1, and is presented in full detail in Sect. IV. Every hardware device (e.g., sensors, device gateways) that the end-user wants to use is registered in the NETIoT system with a unique ID. All data sent by the sensors must first be received using the MQTT protocol by a device gateway which forwards all data to a cloud gateway. Because NETIoT provides multiple services for multiple use-cases, every end-user must define a list of sensors that the NETIoT platform will manage and select which services they want to use and which sensors will provide all data needed for each service. Also, every sensor can be used by multiple services.

On early release, NETIoT will provide services that will respond to several use cases:

1) Air pollution monitoring - data is collected from mobile sensors about pollution at city-level scale.
2) Building monitoring - data is collected from sensors inside a building with comfort-related readings.
3) Smart agriculture - illustrate a case study for monitoring IoT data for craft and live stocks.

Compared to other existing solutions (such as the ones presented in Sect. II), the main goal of NETIoT is to ensure high availability, scalability and also further development. To meet these requirements, NETIoT's model was implemented using a microservices-based architecture, which has the following advantages:

- every microservice can be independently developed from others
- microservices they are usually faster and less expensive to develop than regular monolithic services
- each component has its own database, depending of service needs
- microservices are quick to build and deploy
- a microservices-based architecture ensures high availability and scalability for the system.

Each microservice developed will be packaged in a Docker container [1]. This ensures a faster launch of microservices in production, a better use of hardware resources, and better isolation from the host machine's operating system. All containers are managed by Kubernetes [2], an open-source orchestrator for cluster of containers. By using a microservices-based architecture and Kubernetes for managing clusters of containers, NETIoT provides two main features: the ability to add new services for new uses, and the possibility of offering customers the opportunity to create their own services that can better meet their own needs.

Another key factor in a scalable and high-availability system is communication, thus NETIoT uses the HTTP and MQTT [3] protocols. All data transmitted from sensors to device gateways employs MQTT, whereas for communication from the device gateways to our service HTTPS is employed. Furthermore, communication between microservices is asynchronous, event-based and uses the CQRS (Command Query Responsibility Segregation) pattern [4].

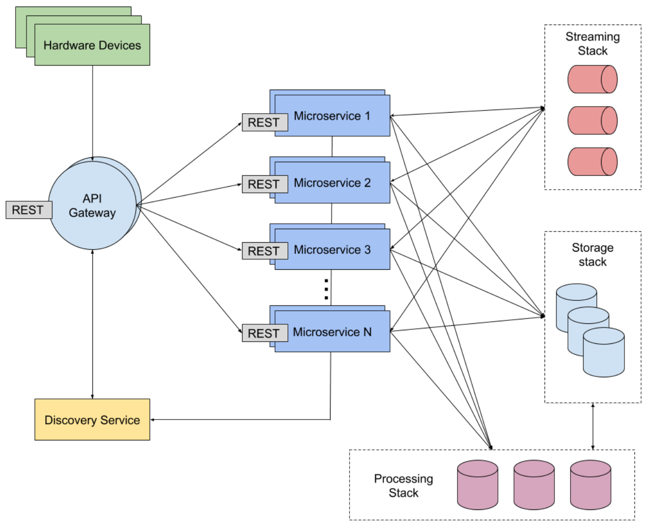

Figure 1. NETIoT architecture.

Because NETIoT can store high volumes of data, its storage layer must have the ability to scale, perform, and offer continuous uptime. The best database in the market that can meet these requirements is Apache Cassandra, because it was designed to scale almost linearly, to replicate data automatically in one or more data centers, and to deal with failover situations.

After data is received and stored in a database, it must be processed and analyzed. NETIoT can perform two types of processing: batch processing and real-time processing. Real-time processing can be used to detect the occurrence of important events, whereas batch processing can be used to analyze all received data within a given time frame from multiple sensors and predict a possible future evolution.

For both types of processing (batch and real-time), NETIoT will use Apache Spark [5], a fast and general use engine for large-scale data processing. The main advantage of Spark is given by the ability to do all processing in-memory and the support for sophisticated analytics and real-time stream processing using multiple languages like Java, Scala, Python.

IV. Detailed description of NETIoT

Through NETIoT, we wish to create an IoT framework that is scalable, extensible, has a high software and hardware fault tolerance and availability. It allows the connection of an unlimited number of sensors, as well as the virtualization and interpretation of the data they send. For the first version of the platform, data will be collected from LoRa-enabled

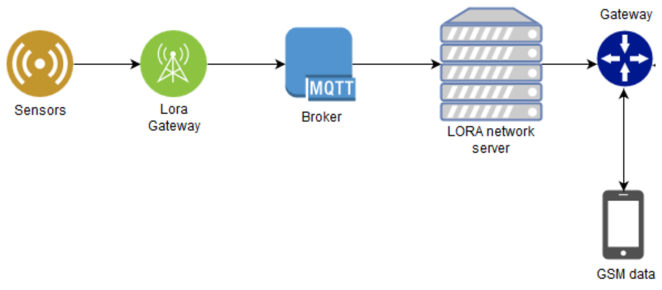

Figure 2. NETIoT hardware devices architecture.

sensors or from mobile devices such as smartphones, which have their own sensors. In order to monitor their sensors, clients will only have to register them in the system and configure the type of application that they wish to use for monitoring. In the remainder of this section, we present a detailed description of the IoT platform, by analyzing and describing each of its components, as depicted in Fig. 1.

A. Hardware devices

The hardware devices refer to the sensors used to collect data and send it to the NETIoT system. For the initial version, NETIoT supports two types of sensors, as stated above. Standalone sensors are able to send data to the system themselves, without the use of a third party. Current sensors employed in NETIoT are used for monitoring air quality, buildings, sound, temperature and humidity, etc. They send data through the LoRaWAN protocol to a LoRa gateway (Fig. 2), using the standard LoRa message format [6], which specifies that a message contains a header (starting with a 1-bit, followed by the length of the message and a parity bit) and the sensor values (containing a byte for the operation code and a maximum of four bytes for the actual values).

Then, the LoRa device gateway collects data from sensors and sends them further towards the LoRa network server using an MQTT broker. It also has the possibility to send back configuration data to the sensors. The MQTT broker sends messages from the LoRa gateway to the LoRa network server. For this component, NETIoT employs rabbitMQ. Finally, the LoRa network server receives data sent by the LoRa gateway, ensures that there are no duplicate data entries, and then sends the remaining data to the device gateway using REST. It should be mentioned that the network server does not perform other data analysis (such as verifying if the sensors that the data come from are registered in the system), all subsequent operations being performed at the API gateway and microservice level. An example of data sent to the API gateway and stored in the database is shown below. In this example, a JSON array is sent with information for multiple devices. *lat* and *lon* represent the geographical coordinates of the device gateway sending the data, *timestamp* is the time when the data were received at the LoRa network server or the mobile device, *devAddr* and *devEUI* are device addresses, while *payload* is the actual useful information sent by the device.

```
{
  "devices" : [
  {
    "lat" : "45.862",
    "lon" : "26.642",
    "timestamp" : "2018-02-13 T 10:30 UTC",
    "devAddr" : "DD15AF9E",
    "devEUI" : "0004A30B001C1C79",
    "payload" : "00FF00FF00FF"
  },
  {
    "lat" : "45.861",
    "lon" : "26.342",
    "timestamp" : "2018-02-13 T 10:31 UTC",
    "devAddr" : "DD15AF0E",
    "devEUI" : "0004A30B011C1C79",
    "payload" : "FF00FF00FF00FFFF"
  }]
}
```

The second type of sensors are not able to send data through LoRaWAN directly to the system, but will send their data to mobile devices, which will act as collectors, and then send the data to the system themselves. It should be mentioned here that sensors actually belonging to the mobile devices (such as GPS, barometer, thermometer, etc.) are also part of this category, since the mobile device has direct access to the data they collect. Furthermore, the data format is the same as for the standalone sensors, so everything that reaches the system has the same format and is efficient in terms of size.

For collecting data from non-standalone sensors, a mobile application is used, which is able to run on both Android as well as iOS. Prior to using this application with NETIoT for data collection, users need to generate a virtual DevEUI (which is a 64-bit end-device identifier that represents a LoRaWAN address) for each sensor (standalone sensors have their own DevEUIs which only need to be written in the application when registering, whereas for the other sensors the application generates custom DevEUIs). The mobile application will allow users to authenticate, to select virtual DevEUIs for which the user wants to send data, to detect sensors, to read or view data from the sensors, and to send data collected from the sensors to the NETIoT system.

B. Device gateway

The device gateway is the only access point in the system, and its role is to route messages to the API gateway. As seen in Fig. 1, all hardware devices (either the LoRa network server or smartphones) connect directly to it.

C. API gateway

The API gateway is the connection point between the "outside world" (i.e., the data collectors) and the applications. It has the following roles:

- user registration;

- authentication using JSON Web Tokens: 1) the client sends a user and password; 2) the API gateway sends them to the AUTH microservice, which is a specialized authorization component; 3) the gateway receives a token containing the user's ID and role, which is then send to the client; and 4) for each subsequent request, the client includes the token, which the API gateway checks using the AUTH microservice;
- sensor data storage by communicating with the storage stack;
- user data retrieval (account information, available applications, available sensors, etc.);
- application parameters retrieval (requests types of parameters and display mode from the application);
- application retrieval (requests types of output and display mode from the application);
- sensor data retrieval (the client performs periodic application requests to read sensor data, and the API gateway returns the data from the storage stack);
- user interface, consisting of: 1) editing account data; 2) adding standalone sensors (by defining names, UUID, type, description, etc.); 3) adding virtual sensors (for the mobile application); 4) selecting applications (defining associated sensors and parameters); 5) visualizing per-application data; and 6) visualizing data from any application defined in a customizable client dashboard;
- admin interface, consisting of: 1) adding, retrieving, deleting, modifying users; 2) adding applications; 3) user statistics; and 4) sensors and applications statistics.

D. Applications and microservices

In NETIot, an application is a set of stateless microservices (there can even be a single one) that use data from various sensors to perform computations on them, display them, etc. An application defines the following things:

- general application parameters (for example, the telephone number where an SMS alert is sent when certain conditions are met);
- per-sensor application parameters (for example, the temperature limit that needs to be reached to trigger an alert);
- minimum and maximum limits for the number of sensors in the application and their types;
- other application IDs that this current application might depend on;
- data visualization format.

Each component of an application is a separate microservice running as a Docker container. As as example, let us assume a simple temperature monitoring application. It will have several components, such as a visualization module, a data analysis module (which reads the temperature from the database and triggers an alert when a certain threshold is reached, a notification module (which sends an SMS when the alert is triggered), etc.

Since they are stateless, applications perform periodic requests to the storage stack for reading data from the sensors or from other applications. The storage stack verifies all the sensors associated with the application and returns the desired data, which is serialized using Protocol Buffers [7]. If the application generates data itself, it will send them to the storage stack along with the user ID and the application's virtual DevEUI (which is automatically generated whenever an administrator adds a new application to the system, and is unique per application). For complex data processing and computation, applications can employ processing stack, while the streaming stack is useful when data streaming is required. Furthermore, the same microservice (i.e., application module) can be employed for multiple applications. In order for the API gateway to be aware of the available microservices and their state, a discovery service is employed.

E. Stacks

As stated before, there are three stacks in NETIoT: storage, processing and streaming. The storage stack contains several databases used for storing various types of information, from sensor data to application information. It has the role of serializing and deserializing the information, retrieving data for applications, etc. A Cassandra database is employed for sensor data, whereas a PostgreSQL database is used for other types of information (application data, user data, etc.). The processing stack contains components for ease of data processing, such as Apache Stark, while the streaming stack has helpful tools for continuous data.

V. USE CASE

Air pollution is the biggest health risk in Europe and its effects are substantial. Cardiac disease and stroke are the most common cause of premature deaths attributable to air pollution and are responsible for 80% of the cases [8]. In addition to causing premature death, air pollution increases the incidence of a wide range of diseases (e.g., respiratory and cardiovascular disease and cancer), with both long-term and short-term effects on health, including below established levels by the World Health Organization [9]. Various reports show that air pollution has also been associated with impacts on fertility, pregnancy, neonates and children.

To monitor air pollution, the use of static sensors within the city cannot lead to a correct representation of different degrees of pollution in different places within the cities. From this perspective, portable technologies for capturing air pollution particles, like Air Patrol, Atmotube, or Quantifly, prove to show higher efficiency and accuracy in air quality monitoring. However, these technologies still need to show they deal with the associated monitoring problems:

- air quality is defined by a plethora of compounds to be monitored simultaneously;

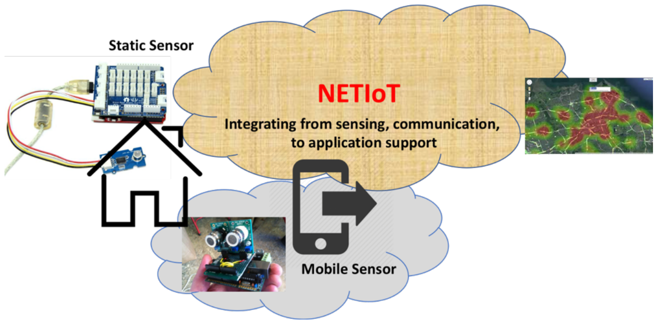
Static Sensor
NETIoT
Integrating from sensing, communication, to application support
Mobile Sensor

Figure 3. The concept of an integrated IoT platform to support the example of air pollution monitoring application.

- for data transmission, collection technologies should use energy-efficient protocols and ensure compatibility with new versions of M2M [10] protocols using existing broadband-mostly networks [11];
- adequate end-to-end security and data protection levels;
- data storage and processing are required for data storage and management [12].

Air quality status for the case of Romania, and especially Bucharest, has been addressed in various studies [13], all indicating a level of some pollutants above the maximum limit regulated by the European Union. Bucharest is the city where most months of life (22) are lost due to exposure to particulate matter (PM2.5), according to a study conducted for 25 major capitals and cities in the EU [13]. Such results are based on the values measured at the eight measurement points in Bucharest of the air quality monitoring network. These are few in number (relative to the extent of the city) and technologically outdated, being in the process of replacement (without the need for more monitoring stations).

Through NETIoT, we can make the implementation of such an air pollution monitoring application feasible quite easily. In our case, we can integrate one pollution sensor (static sensor) next to a user's house, and a pollution sensor to be connected by Bluetooth to the phone for communication purposes (mobile sensor), as depicted in Fig. 3.

With such an application, air pollution sensors collect information which is further used for various services. For example, the user runs an app on his smartphone that alerts him whenever he walks in an area where some air pollution component is monitored to pass air quality thresholds, or the current average is higher compared to the values witnessed for the same area the days before. Furthermore, the user can be notified when the mobile air pollution sensor depletes its battery (to get a chance to charge or replace the battery). In this example, we assume that the user wants to receive an alert (through an SMS on his phone) whenever a certain pollution threshold is reached (at either sensor).

To illustrate the preparatory phase (before entering NETIoT), the LoRa-enabled static sensor sends its data as shown below, a format which was presented in Sect. IV.

```
{
  "lat" : "45.862",
  "lon" : "26.642",
  "timestamp" : "1519397052394",
  "devAddr" : "DD15AF9E",
  "devEUI" : "0004A30B001C1C79",
  "payload" : "00FF00FF00FF"
}
```

To be registered in NETIoT, the user accesses the API gateway, where first he registers as a valid user. Next, he registers the DevEUI of the device he wants to add and maps it to one of the existing applications. Sensor types needed for the execution are defined in the application's metadata and are dynamically obtained by the discovery service. Whenever a new application is being deployed, this service automatically reads and works with its set of defined capabilities (among which are the sensors needed as input). A similar logic applies when registering the mobile sensor in NETIoT, with the difference that we set the air pollution sensors together with a battery sensor.

The user is presented with the set of discovered applications that can work with the type and number of defined sensors. Let us assume the user chooses to link his sensors with the "Air pollution for my house" and "Battery alert" applications. For "Air pollution for my house", both sensors are further associated. Now the API gateway sends a request to the application to obtain info on the selected parameters. The result will be a list of parameters which are sensor-based (e.g., the user needs to select which is a fixed sensor) or application-based (e.g., the phone number where alerts are to be send). NETIoT further generates a unique application ID, to which the two sensors will be associated (together with the metadata, such as the alerting phone number). For the "Battery alert" application, only the mobile sensor is associated. The settings parameters are again sensor-oriented (e.g., battery capacity or the percentage threshold above which to generate an alert) and per-application.

On the mobile application, the user logs with the same account used to register in NETIoT's API gateway. In the settings menu, the user has a list of DevEUIs generated for his account. To each such DevEUI, he can associate the hardware address of the Bluetooth sensors. The idea is to simultaneously use multiple Bluetooth-based sensors, so the application will scan for Bluetooth devices and list all the available ones. The sensed data is coded in the LoRa format (due to the limited payload fingerprint) and sent directly to the main NETIoT gateway. Furthermore, the application allows the visualization of the sensed values.

Once the sensors start collecting data, they send it to the device gateway, then they reach the API gateway, which then stores them in the storage stack, only after verifying if the DevEUI of the data is registered in the system and associated to an application. If this is the case, the data are decoded in the JSON format presented in Sect. IV and are saved in the database, both in a raw and in a JSON format, together with the time when the API gateway received the data.

Periodically, the application requests from the storage stack the values from its associated sensors. For the non-mobile sensor, the data retrieved belong to the previous day, whereas from the mobile sensor the user requests the last recorded value (both being represented in the same format). Then, the application compares the values with the threshold. If the values are above the threshold, an SMS is sent to the phone number specified by the user when configuring the application, and the alert is saved in the database. For the "Battery alert" application, the current battery level is retrieved from the database and will be compared to the threshold set on configuration, and alert will similarly be sent when the limit is reached.

To visualise the data, each application is logically composed of a *backend* (developed starting from a Java Spring template) and a *frontend* (where the react framework is used). The frontend is built as a single *.js* file, that is served as User Interface back to the client. This UI is dynamically loaded, as the client sees a list of available/discovered application, can select each of these, enter the visualization mode (where we serve the *.js* frontend), and dynamically invoke from the frontend other functions of the backend. An application is composed as a mashup, and it can include charts, gauges (the main function being called is *getWidgets*, that returns all forms of visualisation available, all being dynamically loaded in the client's UI).

To give an example, the air pollution application defines the following widgets: a graph comparing two or more air pollution parameters (particles being monitored), a heatmap dynamically constructed based on the values being captured by the mobile sensors, and a table with alerts for passing defined alerting thresholds for the monitored parameters. Similarly, for the battery alert application we define two widgets: a gauge with the last monitored battery level, and a table with alerts being detected for deployed battery levels.

On the backend, an application is composed of i) a communication module with the I/O server, defining all read/write functionality with potential helper functions (like coding/decoding the dataset), ii) the Web service modules (with functions for *getParametersInfo()* or *getVisualizations())*, and iii) the module for calling advanced computations (like computing the average of values for a period).

VI. Conclusion

In the landscape of IoT platforms there are today solutions working at various levels for collecting data and supporting specialized set of IoT services. We presented NETIoT, a SaaS IoT platform designed around the applications being supported by a particular set of IoT-collected data. Compared to other solutions, NETIoT ensures high availability, scalability and also ease of building new applications starting from already-existing sensors, due to its microservices-based architecture.

Acknowledgment

The research presented in this paper is supported by the following projects: MONROE - Toff (H2020 No 644399), NETIO - TELMONAER/ForestMon (53/05.09.2016, cod SMIS2014+ 105976), SPERO (PN-III-P2-2.1-SOL-2016-03-0046, 3Sol/2017) and ROBIN (PN-III-P1-1.2-PCCDI-2017-0734). We would like to thank the reviewers for their time and expertise, constructive comments and valuable insight.

References

[1] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.

[2] E. A. Brewer, "Kubernetes and the path to cloud native," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*. ACM, 2015, pp. 167–167.

[3] U. Hunkeler, H. L. Truong, A. Stanford-Clark *et al.*, "Mqtt-sa publish/subscribe protocol for wireless sensor networks," in *Communication systems software and middleware and workshops, 2008. 3rd conf. on*. IEEE, 2008, pp. 791–798.

[4] D. Betts, J. Dominguez, G. Melnik *et al.*, "Exploring cqrs and event sourcing: A journey into high scalability, availability, and maintainability with windows azure," 2013.

[5] X. Meng, J. Bradley, B. Yavuz *et al.*, "Mllib: Machine learning in apache spark," *The Journal of Machine Learning Research*, vol. 17, no. 1, pp. 1235–1241, 2016.

[6] M. Westenberg. (2016) Lora message format. [Online]. Available: http://bit.ly/2FcioT6

[7] K. Varda, "Protocol buffers: Googles data interchange format," *Google Open Source Blog*, vol. 72, 2008.

[8] C. Sims, C. Hull, E. Stark, and R. Barbour, "Key learnings from ten years of monitoring and management interventions at the bluff point and studland bay wind farms: Results of a review," in *Wind and Wildlife*. Springer, 2015, pp. 125–144.

[9] P. Kumar, A. N. Skouloudis, and M. Bell, "Real-time sensors for indoor air monitoring and challenges ahead in deploying them to urban buildings," *Science of the Total Environment*, vol. 560, pp. 150–159, 2016.

[10] M. K. Patil and S. Lahudkar, "A survey of mac layer issues and application layer protocols for machine-to-machine communications," 2016.

[11] F. Dressler, S. Ripperger *et al.*, "From radio telemetry to ultra-low-power sensor networks: tracking bats in the wild," *IEEE Comm. Magazine*, vol. 54, no. 1, pp. 129–135, 2016.

[12] A. S. Jones, J. S. Horsburgh, S. L. Reeder *et al.*, "A data management and publication workflow for a large-scale, heterogeneous sensor network," *Environmental monitoring and assessment*, vol. 187, no. 6, p. 348, 2015.

[13] E. E. Agency. (2016) Air quality in europe - 2016 report. [Online]. Available: https://www.eea.europa.eu/publications/air-quality-in-europe-2016