Development of a remote-control system for GNSS sensors using AWS IoT

Arnau Ochoa Bañuelos

Resum– La Internet de les coses (IoT en anglès) s'està convertint en una de les tecnologies més importants i s'espera que seguirà en creixement en el futur. Per tal de desenvolupar un sistema IoT existeix l'opció de desenvolupar tota la infraestructura, però també és una opció utilitzar una de les solucions que algunes empreses proveeixen. Un pot imaginar que la segona opció serà millor en la majoria de casos, ja que serà una solució més ràpida i sovint funcionarà millor. Aquest article presenta com una de les solucions més utilitzades, la que proporciona Amazon Web Services, s'ha utilitzat en un projecte real. Aquesta tecnologia s'ha utilitzat per a desenvolupar un sistema IoT per a controlar remotament els dispositius que s'utilitzen com a sensors GNSS en el projecte Cloud GNSS Rx. Els components d'AWS IoT s'expliquen en aquest document i es detalla com aquests s'han utilitzat per a desenvolupar el sistema. A més, també es detalla com s'ha implementat un *dashboard* web el qual s'utilitza com a *front-end* per a l'usuari.

Paraules clau– Amazon Web Services, Internet of Things, Ruby on Rails, Raspberry Pi, Cloud GNSS Receiver.

Abstract– The Internet of Things (IoT) is becoming one of the main technologies of the present, and it is expected to keep growing in the future. In order to develop an IoT system there is the option of developing all the infrastructure, but it is also an option to use one of the solutions that some companies provide. One can imagine that the second option will be better in most cases as it will be a faster and usually a better solution. This article presents how one of the most used solutions, the one provided by Amazon Web Services, is used in a real project. This technology to develop an IoT system to remotely control devices that are used as GNSS sensors within the Cloud GNSS Rx project. The AWS IoT components are explained on this document and, then, it is detailed how they have been used to develop the system. Furthermore, the implementation of a web dashboard that is used as the front-end for the user is also detailed.

Keywords– Amazon Web Services, Internet of Things, Ruby on Rails, Raspberry Pi, Cloud GNSS Receiver.

1 Introduction

The *Cloud GNSS Rx* is a project developed and implemented by the SPCOMNAV[1] research group at UAB and funded by the European Space Agency. This project consists on the development of a Global Navigation Satellite System (GNSS) receiver on the cloud, which allows a device to capture the samples of the GNSS signals and to send them to the cloud receiver. Once received, the cloud receiver processes the signal samples and returns the desired information (e.g. the position) to the device [1, 2].

This solution is a paradigm shift with respect to the typical GNSS receivers, which implement all the signal processing on the receiver device. This fact involves a high complexity on the design of the device and an also high power consumption [3]. Furthermore, new global positioning systems will soon be completely operational, resulting on the coexistence of 4 different systems, i.e. The American GPS, the European Galileo, the Russian GLONASS and the Chinese Beidou. This will solve some of the current problems in these systems such as the availability and will also improve accuracy. Nonetheless, this evolution will also increase the complexity of the signal processing and, hence,

• Contact email: arnau.ochoa@e-campus.uab.cat
• Specialisation: Information Technology
• Project supervised by: José A. López Salcedo (TES)
• Academic year 2018/19

[1] http://spcomnav.uab.es/

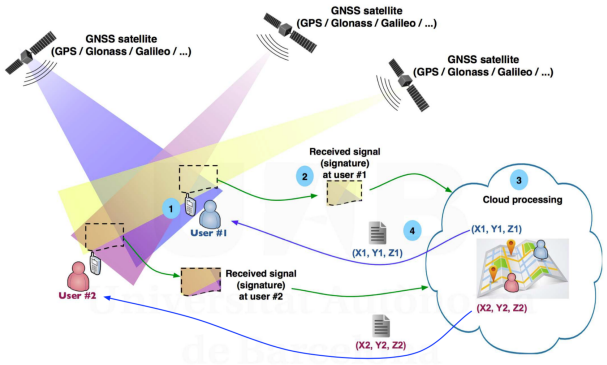

Fig. 1: Illustration of the *Cloud GNSS Rx* paradigm [2].

the computational requirements will be greater.

The development of a GNSS receiver on the cloud aims to solve the previously mentioned problems, since the signal processing is moved from the main device to specifically dedicated machines, which have much bigger computational capacities [4]. This characteristic makes the *Cloud GNSS Rx* a very good solution for many applications which require to compute the position, but it is specially pertinent for Internet of Things (IoT) applications. These kind of applications require low-complexity devices with also low cost and power consumption. Therefore, it is quite difficult to implement a GNSS receiver on IoT devices, specially when a high precision is required. Furthermore, having a GNSS receiver on the cloud also provides another capacity which is very useful for IoT applications: the capability to analyse a group of data obtained from different receivers. This capacity allows to obtain additional information apart from the position of the receivers. For example, the measurements obtained by different receivers that are geographically distributed can be used to locate the source of an interference signal.

Within this context, a typical application would have various devices or sensors which would, more or less continuously, capture the GNSS signals to send them to the cloud receiver. This receiver would return the position of the receivers and the other information required to the devices and/or to a centralised server, which would manage this information, figure 1 illustrates the Cloud GNSS Rx paradigm from a high level. This situation becomes complex when the system incorporates a large number of devices or when these devices are geographically disperse, since the status of the sensors must be possible to track and control. In order to solve this problem, a system has been implemented to allow the visualisation and modification of the state of the devices from a web dashboard. The IoT service provided by Amazon Web Services (AWS) is the technology that has been selected to implement the communication between the devices and the central server. Later on, the objectives of this project are detailed.

This paper is structured as follows. First of all, the objectives and requirements are detailed in section 2. Then, section 3 explains which technologies have been selected to develop the system and why. Later, in section 4, the methodology that has been followed during the development is explained and the planning of this development is detailed. Section 5 describes the AWS IoT components some related services offered by AWS. Later on, in section 7 the resulting system capabilities are shown. Finally, some conclusions are drawn in section 8 and some future work is left open in section 9.

2 Objectives

The main objective of this project is to obtain a communication system between the sensor devices and the administrator, so that the devices can be remotely configured and controlled. This main objective is divided in for specific requirements, which are described next.

- The devices must be remotely controllable and whenever the parameters of a device are modified, this device must maintain these new parameters until a new definition is done.
- It must be possible for the administrator to remotely initiate a capture of GNSS signals and also to program a capture every certain time.
- It must be possible to define the parameters of any device even when the devices are not connected to the Internet. When the devices recover the connection, the new parameters must be automatically updated with the last definition.
- The monitoring and control of the devices' state must be done from a web dashboard.

3 Selected technologies

In order to solve the objectives previously described, it has been decided to utilise a *Platform-as-a-Service* solution, since these solutions allow to notably simplify the implementation of the system [5]. This is because the communication protocols have already been defined, no infrastructure has to be built and the security is partially implemented by the service itself.

The platform that has been selected is the IoT service offered by Amazon Web Services (AWS) [2] since this platform is used on other parts of the project [2], such as in the database or in the cloud computational services. This way, all the services are in the same company which simplifies the management and the communication between modules. Also, this service is one of the most used for implementing IoT systems [6].

For the development of the website, the selected technology has been *Ruby on Rails*, since this web dashboard will be included as a module on the web platform of the Cloud GNSS Rx project, which has been already developed in *Ruby on Rails*.

4 Methodology and planning

The development of this project has been carried out in an iterative manner. Since some of the technologies used were unknown at the beginning of this project, the development of some features has required a previous phase of acknowledgement of the technology. Then, the implementation of

[2] https://aws.amazon.com/iot/

the system has been done by the iterative development of new features, following a *feature-driven development* process. This process, included in the *Agile* methodology, bases the development on the definition of various features. Once the features are defined and planned, these are implemented one after the other, testing each one of them before starting the development of the next feature. Also, the *Git* software is used with the *GitHub* platform in order to keep track of the versions of the software. A *Raspberry Pi Zero W* has been used as the testing device, since the prototype sensors of the *Cloud GNSS Rx* are based on this device.

In order to solve the requirements specified in section 2, the implementation of the system is divided on two main parts which can be clearly differentiated. The first part consists in the development of the communication system between the central AWS IoT server and the various sensors. This part requires the configuration of the server and the implementation of the software which will carry the communication routine and the execution orders on the devices.

The second part of the system development consists in the development of a web dashboard which allows the administrator to visualise and configure the devices' states. Furthermore, the administrator has to be able to add or remove devices from the system using the same web dashboard. This web is needed since the control console offered by Amazon is not designed for the final users but for the developers of the system. Therefore, this is not a user-friendly environment and it is also not very versatile, since it does not allow the control of the selected values. Also, this control dashboard has to be added to the already developed web of the Cloud GNSS Rx.

4.1 Planning

The development of the system previously described has been divided in 6 tasks in order to better organise the implementation over the time. Each of these tasks includes one or various features to be added to the system and their tests. Next, a revision of these tasks is provided in chronological order. The three first tasks are related with the first part of the development while the 3 last tasks are related with the second part of the development, as explained before in this section.

4.1.1 Basic controller

In order to know how the AWS IoT platform and its SDK work and also to understand how the communication between devices and the server work, a basic routine was implemented at the beginning of the development. This basic routine offered a basic communication with which a variable *mode* could be modified from the AWS IoT dashboard. This variable had no effect on the device and it was only used for communication test purposes.

4.1.2 Complete controller

Once the communication was properly working between the server and the device, the complete control program was implemented. On this controller, all the parameters required to control the device were added. Also, these parameters were saved on a file and updated on every new request from the server. Finally, the call to capture signal samples was also implemented.

4.1.3 Error control

The parameters that define the state of the devices may have invalid values, since these parameters are transmitted in JSON format, which has no control of the possible values that a variable can take. Therefore, the validity of these values must be controlled by the receiver. On the other hand, the software that controls the capture of the GNSS signals may also cause some errors. Therefore, a control of these possible errors was implemented, which acts accordingly to the kind of error raised.

4.1.4 Basic website

In order to familiarise with the *Ruby on Rails* environment and its AWS IoT SDK, a basic web was developed first. This basic web showed the name of a single device and its parameters. Also, the parameters of this device could be modified with a basic form. Therefore, the bidirectional communication between the website and the server was tested and, hence, so was the communication between the website and the device.

4.1.5 Complete website

Once the correct operation of the previous task was checked. A more complete website was developed, which showed all the devices saved on the AWS IoT server. Also, navigation between screens was implemented and the interface used to modify the devices' parameters was improved, from a very basic form to a more user-friendly environment. Finally, the control of the possible values was also implemented, so only the valid values could be selected for a given parameter.

4.1.6 Add/delete devices

Later on, the options of adding and removing devices from the system were implemented. In the case of adding a new device, the required credentials can be generated and downloaded. Besides that, whenever a device is deleted from the system, its corresponding credentials are invalidated.

4.1.7 Messages visualisation

The last feature that was implemented was the visualisation on the website of the control messages that the devices send, differentiating the normal messages and the error messages. Also, during this phase, the aesthetic of the website was improved. Even though the final style of this website will be defined when it is joined with the Cloud GNSS Rx website, it is interesting to have a better visualisation for the tests that other members of the project team may carry.

4.1.8 Final tests

Although every one of the previous tasks has included its own tests, a final test task was carried in order to check the right behaviour of the system in all possible situations. Here, functionality and security tests have been carried out,

such as code injection tests on the forms that the website contains.

5 Description of AWS IoT

AWS is a collection of cloud computing platforms called services offered by Amazon. The purpose of these services is to provide solution to individuals or organisations to develop their systems. AWS offers various systems for different applications among which are EC2 (Compute), S3 (Storage), DynamoDB (Database) and many others. Every service can be used on its own or can be joint with other services in order to build a system.

This section details the characteristics of AWS IoT, the service which is used in this project to implement the communication with the devices. This service defines protocols that developers have to use in order to develop their project and it includes an API and many SDK's for different programming languages so that it can be adapted to the different characteristics of the implementation. Next, a description of the main components of this service are detailed in order to better understand how the system works, which is explained later on this paper. In this section, some related AWS IoT services are briefly explained. These other services are not used on this project but it is interesting to know them in the case that further extensions of the system are required.

5.1 Components of AWS IoT

The *Device gateway* is the door that allows the devices for connecting and efficiently communicating with AWS IoT, either to communicate with other devices or to do so with the applications or clients. All this communication is handled by the *Message broker*, which is a mechanism that allows the devices and applications for publishing and receiving messages. It is based on MQTT, which is a machine-to-machine connectivity protocol designed as a lightweight publish/subscribe messaging transport [7]. With this broker, the clients (devices and applications) can subscribe and publish messages to *topics* so that everyone subscribed to one *topic* will receive the messages published to it. Although it is based on MQTT, it is also possible to use HTTP REST in order to publish messages to *topics*. All this communication can be processed with the *Rules engine*. This component provides message processing and integration with other AWS services. Using an SQL-based language one can select data from message payloads, process the data and send the data to other AWS services, such as Amazon S3, Amazon DynamoDB and AWS Lambda. The message broker can also be used to republish messages to other subscribers.

In order to manage all the physical devices from the cloud, there is the *Thing registry*. This component is used to identify the IoT devices in the AWS Cloud, representing them as *things*. Every *thing* can have assigned certificates, MQTT client IDs and up to three custom attributes. There is also the *Group registry* component, so that different *things* can be managed at the same time using groups. These groups can also be grouped so that a hierarchical structure can be built.

In order to guarantee the security and identity of the systems, the *Security and identity service* provides the components to ensure both authorisation and authentication. The authentication provides security to the system validating the identity of the clients that use the service. This is achieved by using digital certificates for all the involved parties. It also ensures the confidentiality in the communication as it uses the TLS protocol. The authorisation determines what an identity can do. It is given by the use of *policies*, which are assigned to a certificate. A *policy* defines the actions that the identity related to the certificate can do, for example `connect`, `subscribe` or `publish`. It also defines the resources that the identity is allowed to use, such as topics, clients or other resources.

The *Thing shadow* is another component provided to represent the state of a device in the cloud. A *shadow* is a JSON document associated to a thing that represents the state of the device. These JSON is initially made up of two inner JSON, the `desired` and the `reported`. The first one represents the state which it is wanted the device to be in. The second represents the actual state of the device. All the shadows are managed with the *Thing shadows service*, which allows having a representation of the states of all the devices. It also provides the usability of *shadows*. When there are differences between `desired` and `reported`, a third inner JSON is automatically generated. This inner JSON is named `delta` and includes the parameters that are different between `desired` and `reported`. When a `delta` is generated, a message is published to a specific topic in order to communicate that modification to the pertinent device. On the the listing 1, an example of a device *shadow* can be seen. This could be the *shadow* of a device used to remotely control the temperature of a pool. There, the three parts of the *shadow* can be observed, namely the `desired`, the `reported` and the `delta`. The last one has been generated because the `mode` and the `desired_temp` have different values on the `desired` and the `reported`. Therefore, an MQTT message has now been published to the Device-Shadow topic with the `delta` portion of the *shadow*. Whenever the sensor resolves the delta, it will publish an MQTT message to the same topic including the `reported` parameters, which will be received by the AWS IoT server.

```
{
    "desired":{
        "mode": "auto",
        "desired_temp": "28",
        "actual_temp": "25"
    }
    "reported":{
        "mode": "off",
        "desired_temp": "26",
        "actual_temp": "25"
    }
    "delta":{
        "mode": "auto",
        "desired_temp": "28"
    }
}
```

Listing 1: Example of a device shadow

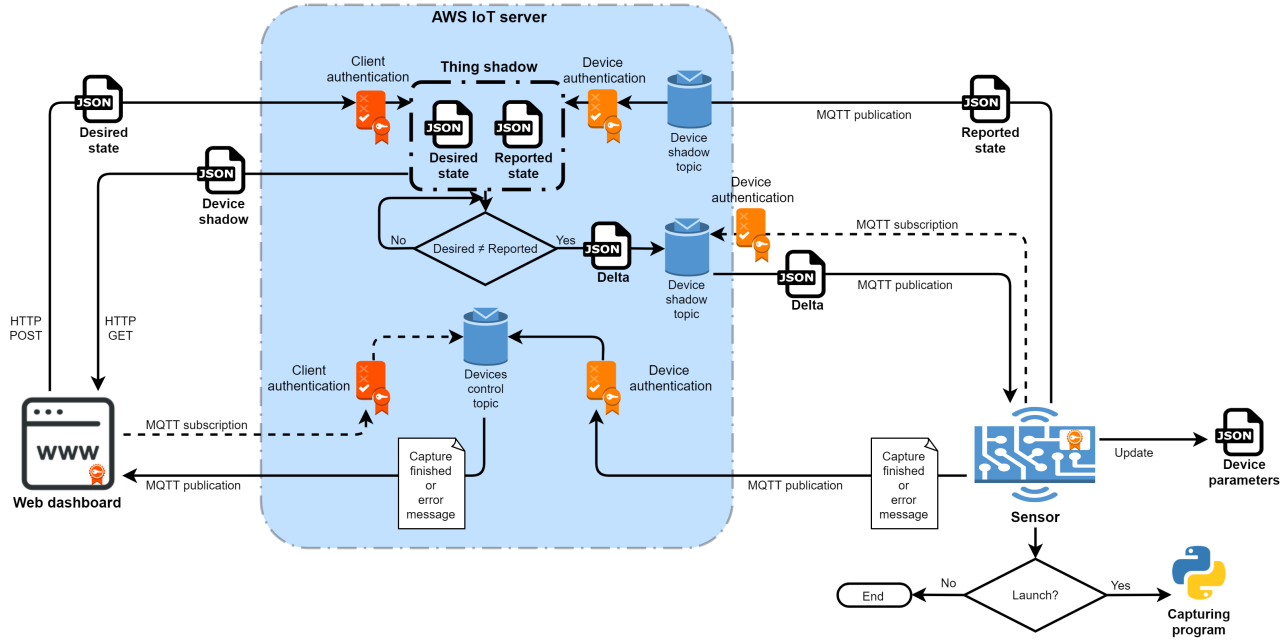

Fig. 2: Representation of the system's architecture

Then, the server will delete the current `delta` and it will update the reported parameters with the last received ones. At this time, if the `desired` and the `reported` portions were not equal, the process would start again.

For some applications, a certain automation of some actions will be needed. In this cases, the *Jobs service* will be useful. Using *jobs* one can define remote operations that will be automatically realised when certain conditions are accomplished. Some examples of applications could be the downloading of firmware or restarting the device.

Finally, for scalability purposes, there is the *Device provisioning service*. With this component devices can be added to the system based on established models that describe the parameters of the thing, the certificate type and the policies.

5.2 Other AWS IoT services

What has been previously described constitutes the main AWS IoT services, over which a complete IoT infrastructure can be developed. Even so, AWS provides two additional services that are very related to AWS IoT and provide supplementary functionalities. This services are named *Amazon IoT Core* and *Amazon IoT Core AWS IoT Device Management*. But around these, more AWS IoT can be used to develop a more complex system. Those services and the functionalities they offer are described next.

5.2.1 AWS Greengrass

AWS Greengrass is a software that can be configured on the IoT devices to add local computation capabilities, messaging data caching, synchronisation and Machine Learning inference capabilities. The software joins all these functionalities with the services previously explained. The Greengrass software, named Greengrass Core, can be installed on devices that run Linux. The devices running Greengrass Core act as a hub that can communicate with other devices that have an AWS IoT Device SDK installed. These devices can be configured to communicate with one another in a Greengrass Group. If the Greengrass Core device is not connected to the cloud, devices in the group can keep on communicating with each other. This service offers some benefits that can be applied on an IoT solution. The first one is that the devices can operate offline following the specified rules. When the core device reconnects, all the information is synchronised. Another benefit is the capability to respond to local events without having to communicate to the cloud, which is faster and less costly.

5.2.2 AWS IoT Analytics

AWS IoT Analytics is a service that allows to run sophisticated analytics on massive volumes of IoT data in a simplified way. The analysis of IoT data is usually complex and costly, as there are a lot of data that can be noisy and have significant gaps. Furthermore, IoT data is often more meaningful in the contest of other external data. This service offers an interface between the capture and the storing of data. This interface can filter, transform and enrich the IoT data before storing it, and all these actions can be configured.

6 System's Architecture

In this section, the implementation of the developed system is explained, detailing its architecture. This architecture can be observed in the diagram of figure 2. This section is divided on two parts: first, the device-to-server communication is explained, which refers to the right half of the diagram. The developed software that is installed on the device to enable communication can be downloaded from this GitHub repository: `https://github.com/arnauochoa/AWS_IoT-sensor`. Later, the website-to-server communication is detailed, which is the left half of the diagram. The source code of this website can also be downloaded from its GitHub repository: `https://github.com/arnauochoa/SensorDashboard`.

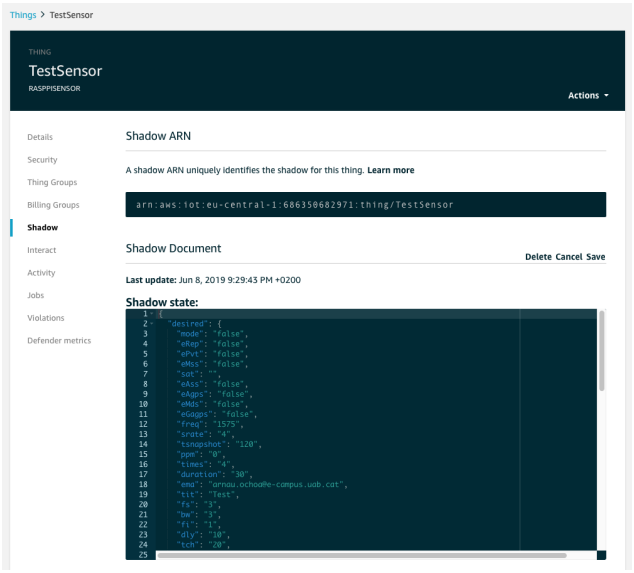

Fig. 3: Capture of the tool provided in the AWS IoT dashboard to modify a shadow.

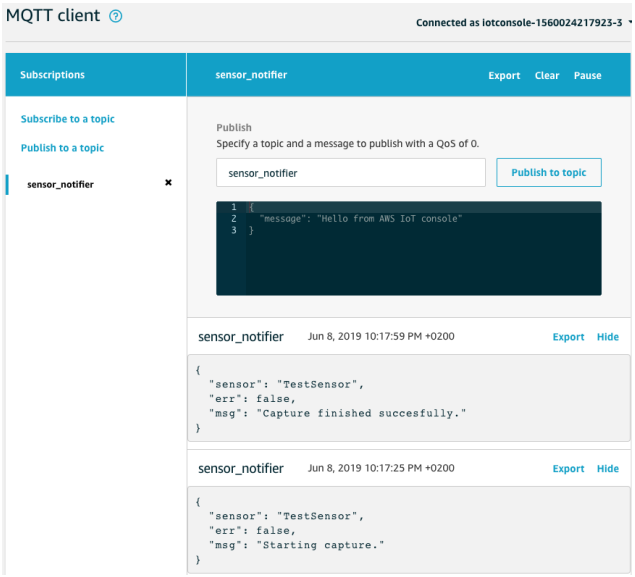

Fig. 4: Capture of the tool provided in the AWS IoT dashboard to visualise the MQTT messaged.

6.1 Device-to-Server communication

The software that runs on the device, which controls the communication between the device and the cloud, has been implemented using the AWS IoT SDK for JavaScript [8]. JavaScript has been the language selected because of its low power consumption compared to the other languages that are available for AWS IoT [9]. This low power consumption is important because typically the devices will have a limited energy source. The controller software is implemented with the Node.js environment, which allows to run JavaScript code on the device. The event-driven architecture of Node.js is also very suitable, since in this project, the devices have a behaviour which is similar to the typical server behaviour: the devices will be constantly waiting to receive orders, and once they receive one they act consequently and they return the desired information.

First of all, the AWS IoT SDK for JavaScript must be installed on the device, this can be done using the *npm* package manager built in Node.js. Also, some certificates and keys must be assigned to thee device in order to identify and validate both parts on the communication with the cloud. As it is explained later on this document, these certificates and keys can be automatically generated and downloaded from the web dashboard developed in this project. The required certificates and keys are described next:

- *Server CA certificate*. This certificate allow the devices to verify that they are communicating with the AWS IoT cloud and not another server impersonating AWS IoT.
- *Client certificate*. This certificate allows the server to verify that it is communicating with the given device.
- *Client public key*. The public key related to the client certificate.
- *Client private key*. The private key related to the client certificate.

The server certificate can be obtained from the AWS developer guide [10], while the device's certificate can be obtained from the AWS IoT console or it can be generated and self-signed by the developer. These certificates and keys are saved on the device. Once the certificates are created and assigned to the device, the corresponding policy must be defined. This policy defines what actions can the device do in relation to the communication with the AWS IoT server. For example, a given device could be forbidden to subscribe to a given MQTT topic. In this case, the policies are equally set for all the devices and they include permission to publish and subscribe to the topics that are used in the project, namely the Device-Shadow topic which is specific to each device and the Devices-Control topic that is used for all devices.

Once the AWS IoT SDK has been installed and the certificates have been saved in the device, a JSON file named `params.json` is created. This file includes all the configuration parameters of the device (e.g. the capture duration, the number of captures or the centre frequency). Once this file is created, it is all ready to run the controller program. The operation of this program is detailed next.

Whenever the device changes from inactive to connected, which means that it is turned on and it has Internet connection, it is registered to its Device-Shadow topic. This is the MQTT topic used between the AWS IoT server and the device for the communication related with the device's shadow. Once the device is subscribed, it updates its *shadow* (i.e. the parameters defining the state of the device). These parameters are saved in the AWS IoT server in the `reported` portion of the device *shadow*. As it is explained before in 5, the AWS IoT server will check if all the parameters of the shadow are equal between the `desired` and `reported` portions of the *shadow* and if so it will do nothing but wait to new changes. In the case of these two parts of the device *shadow* being different, a `delta` is generated. The `delta`, which is a part of the *shadow* in the AWS IoT server, details the values of the parameters in the `desired` part that are different from the `reported`. This message

is then published to the Device-Shadow topic.

Once the device has been subscribed to the Device-Shadow topic, whenever a `delta` is published on this topic, the `delta` handler function will be triggered. This function will first save the new values of the specified parameters on the `params.json` file. Then in the case a parameter named `mode` is set to true, the capturing routine will be triggered. This routine will call a Python script which is in charge of carrying the GNSS signal capture. This Python script will read from the `params.json` file the parameters with which the capture will be carried on. If the capture is realised properly, the capturing script will return a no-error flag to the controller program. Then, the device will send a message to the Devices-Control topic, telling that the capture has been realised properly and specifying the device's name. In the case of having any error during the capture of the GNSS signals, the capturing script will return the error message to the controller program. This error message will be then sent to the Devices-Control topic, together with the name of the device. This routine will be repeated as many times as specified by the corresponding parameter on the device's *shadow*.

Until this point, the device can be controlled from the AWS IoT dashboard. There, the device's *shadow* can be modified by editing the JSON that describes the *shadow*. This can be observed in figure 3, where the *shadow*-edition tool is shown and the device's shadow can be partially observed.

6.2 Website-to-Server communication

As stated previously in this document, there is the necessity to develop a website for the administrators of the devices to use it as the control dashboard of these sensors. Also, this website has to be developed in Ruby on Rails in order to subsequently add it to the Cloud GNSS Rx website, which already has many functionalities implemented such as the control of the users. Therefore, the website implemented on this project does not include some of the functionalities that will be provided after this website is included into the Cloud GNSS Rx one.

The functional requirements of this website are:

- The capacity to visualise all the sensors in the system.
- The capacity to visualise and change the parameters of any given sensor.
- The capacity to create a sensor and, afterwards, to obtain its certificates and keys.
- The capacity to delete any sensor of the system and, consequently, invalidate its certificates and keys.
- The capacity to visualise the messages sent by the sensor devices.

The web dashboard used to control the devices is implemented using the AWS IoT SDK for Ruby on Rails [11] and the MQTT library provided by the same framework [12]. It is also required to obtain the access keys for the SDK and API access to the AWS IoT server. These access keys are needed on the web dashboard since it will obtain and modify information on the AWS IoT server. This keys can be created and obtained by the developer on the AWS IoT console.

For this website, a database has been used over the SQLite3 database management system, since this is the default system in Ruby on Rails. Also, SQLite3 provides a good trade-off between performance, reliability and application complexity. The database contains two tables that are used for saving the information needed to communicate to the AWS IoT server and also for saving the information that can be visualized and modified from the web dashboard. The first table is called `Sensors` and it contains the information of the devices that is needed to retrieve their *shadow* parameters from the AWS IoT server. This table contains the following attributes for each table entry (i.e. each sensor):

- *Sensor name*: Name defined by the system administrator to identify the sensor devices.
- *Type name*: Name of the type of sensor. Used to identify different groups of devices in case it is needed.
- *Certificate ID*: Code used to identify the certificate assigned to the device. It is necessary for the communication between the website and the AWS IoT server.
- *Thing ARN*: The thing Amazon Resource Name uniquely identifies a device inside the AWS IoT system. This is also used for communication purposes between the website and the AWS IoT server.
- *Policy name*: The name of the policy assigned to each sensor. Now it is the same for all the sensors, but it has been left as an attribute for scalability purposes.

This `Sensors` table is updated every time the devices of the system are listed on the main page of the website. Therefore, it is assured that the devices on the AWS IoT server and the devices on the website are the same. Also, this table is modified when a sensor is created or deleted. On the one hand, whenever a device is created, the information of the new device (i.e. its name and the type name) is sent to the AWS IoT server including also the *shadow* of the device, which is created equally for all the devices. Once the *thing* (i.e. the representation of the device on the AWS IoT server) has been created, the Thing ARN is obtained from the server. After that, a new entry is introduced to the table including the sensor name, the type name and the Thing ARN. Later on, when the certificates are created and downloaded by the administrator, the certificate ID and the policy name are added to the sensor's entry on the database table. On the other hand, whenever a device is deleted, a message is sent to the AWS IoT server asking for the deletion of this device and the invalidation of its credentials (i.e. certificate and keys). When a confirmation of the deletion has been received from the AWS IoT server, the table entry corresponding to the deleted sensor is erased from the database.

The second table is called `Messages` and it is used to save the MQTT messages that are published by the devices on the Devices-Control topic, which include information about the state of the devices and any possible error that may occur during the signal capture. This table contains the following attributes:

- *Sensor name*: Name of the sensor that has published the given MQTT message.
- *Error*: Boolean flag that indicates if the message is an error report or it is not.
- *Content*: The main content of the message, for example the description of an error or the notification that the capture has been finished.
- *Time*: The time when the message has been published at.

This table is important because the messages are not saved on the AWS IoT server. Therefore, whenever a message is published by a device on the Devices-Control Topic, it is received on the web dashboard which is subscribed to the topic. The messages are thence saved to the database in order to be able to visualise them from the website.

The modification of the device's *shadow* can be accomplished from the detail visualisation of a device. There, the `desired` parameters of the *shadow* can be visualised and modified using a form. This form has been implemented so that the validity of the parameters is checked. There are, hence, four types of parameters defined. The first type are the boolean parameters, which are modified so that only two possible values can be set (i.e. true or false). The second type are the values that are numbers, which values are checked to be actually numbers and a step between values is defined to be 0.01. The third type is the e-mail address, which validity is also checked. Finally, the fourth type is the text, which has a maximum length of 30. All these values are required and, if they are not set or are invalid, an error message is shown on the given field. This form is also resistant to various code injection attacks. On the one hand, the values introduced on the form are not used or saved on the database, so no SQL injection attacks can be done. On the other hand, the coding recommendations from the Ruby on Rails official guidelines ([13]) for securing Rails applications have been followed on the implementation of the website.

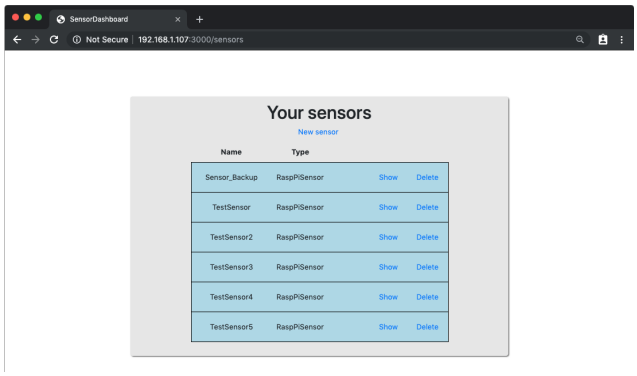

Fig. 5: Visualisation of the list of devices.

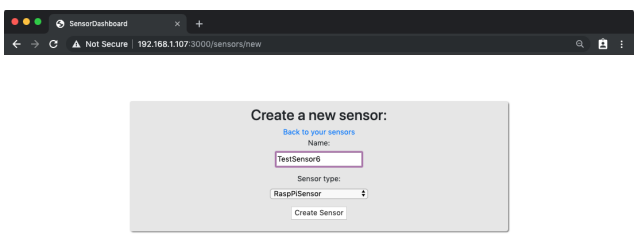

Fig. 6: Page to create a new device.

The `reported` parameters can also be visualised in this page. The shadow of the device is modified on the AWS IoT server using the HTTP protocol. Whenever the desired parameters of the shadow are modified, a JSON is created including the modifications of the *shadow* and this is sent to the AWS IoT server, which changes them after verifying the web client with its certificate. Once, the *shadow* is modified on the AWS IoT server, a `delta` is raised and a message is sent to the Device-Shadow topic corresponding to the given device. Then, the routine that handles this `delta` on the device is started as explained previously in this section. When any device is visualised the *shadow* of the device is requested to the AWS IoT server with an HTTP GET method and then it is shown on the device's detail page.

7 Results

With the implementation detailed on the previous section, an administrator of the system devices is now able to visualise all the sensors on the system and the parameters of each of these devices. The administrator is also able to know how the devices behave and if there is any error on these by means of the MQTT messages that are published on the

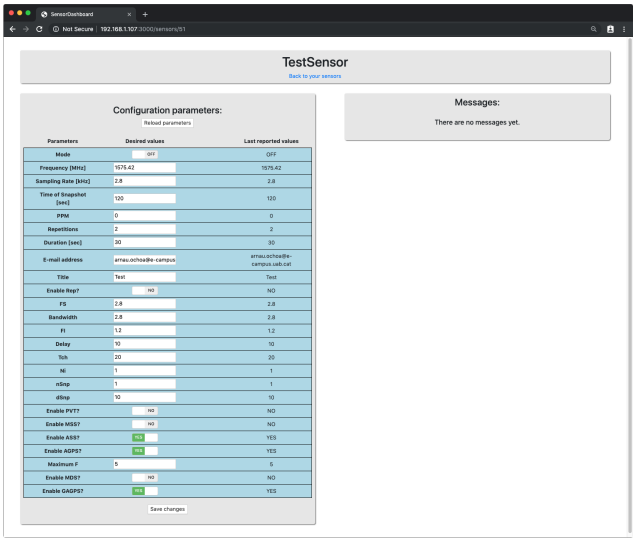

Fig. 7: Detail view of a device.

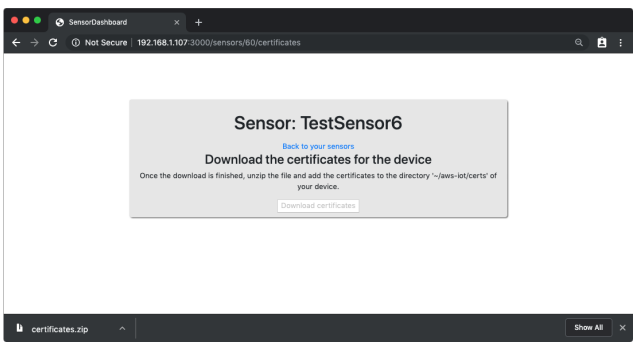

Fig. 8: Page to download the certificates for a new device.

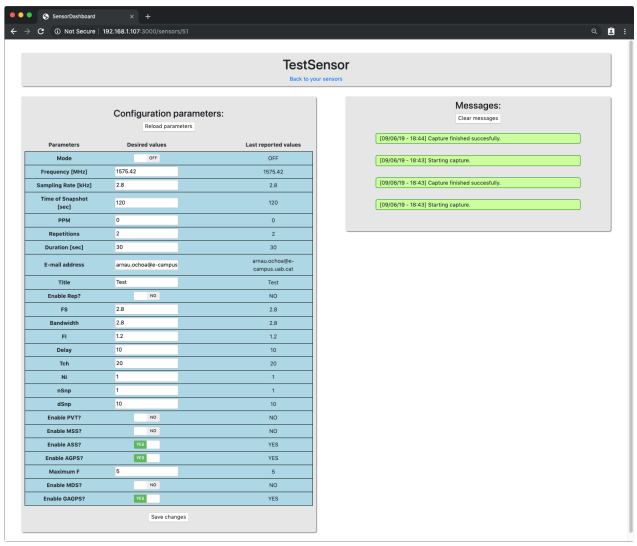

Fig. 9: Detail view of a device with messages showing a normal behaviour.

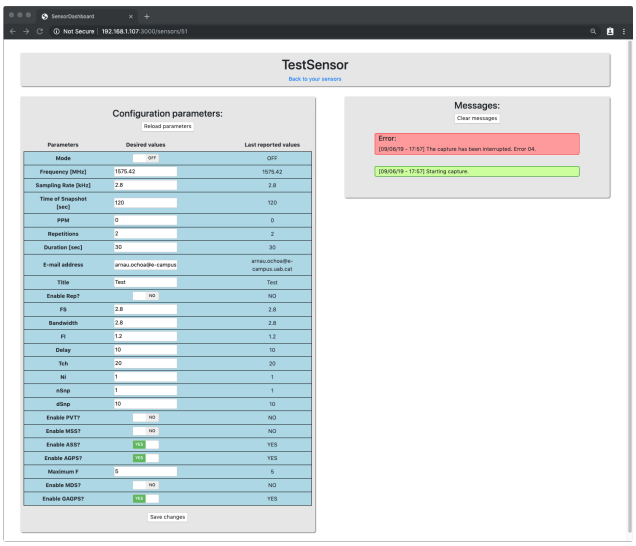

Fig. 10: Detail view of a device with messages showing there has been an error.

Devices-Control topic, which can also be visualised on the web dashboard. Finally the administrator can also create a new device and obtain the certificates and keys needed and he also can delete any sensor from the system. In this section, the functionalities provided by the system implemented are shown by means of some screenshots of the website and the console of the Raspberry device used for the development.

The first page of the web dashboard shows a table with all the sensors that are included in the system (see Fig 5). From this screen, the administrator can also do three main actions: access to the detail of one device, create another device or delete an existing one. The creation of a new device can be seen in figure 6, where a new name is set for the device and the type of device can be selected. After the creation of a new device, the certificates can be generated and downloaded from the page that can be seen in figure 8. Once the certificates are generated and downloaded, they can not be downloaded again for security purposes. This is shown by disabling the button to download but, nonetheless, if the user manages to try to download again the certificates, the controller checks if they have already been downloaded and, if so, it does not allow to download them again.

The detail of a sensor can be accessed from the first page and it has two sections, which can be seen in figure 7. On the left par of the page, the *shadow* of the device is detailed on both the desired and the reported values. The desired values of the device's *shadow* can be modified with a form included in the table. On the right side of the image is where the MQTT messages sent from the device are shown. These messages are shown in green if they are normal messages and they are shown in red if they are an error report. Both situations can be seen in the figures 9 and 10, respectively.

As it has been explained before in this document, whenever the shadow parameters are changed and submitted on the website, these are sent to the AWS IoT server and, then a message is published so the devices know which parameters should be modified and how. This procedure is used for defining the parameters that must be used for the signal captures but also for triggering the capture itself. This can be seen on the figure 11, which shows a screenshot

```
pi@raspberrypi:~/aws-iot/main $ node main.js -F lib/config.json
[2019-06-09T17:16:27.523] [INFO] default - ====================================
[2019-06-09T17:16:27.576] [INFO] default - Subscribe to topic: sensor_notifier
[2019-06-09T17:16:27.584] [INFO] default - ====================================
[2019-06-09T17:16:32.709] [INFO] default - ====== LAUNCHING CAPTURE ======
[2019-06-09T17:16:32.717] [INFO] default - Duration: 30
[2019-06-09T17:16:32.722] [INFO] default - Repetitions: 2
[2019-06-09T17:17:02.757] [INFO] default - ====== CAPTURE FINISHED ======
[2019-06-09T17:17:02.990] [INFO] default - ====== LAUNCHING CAPTURE ======
[2019-06-09T17:17:02.997] [INFO] default - Duration: 30
[2019-06-09T17:17:03.003] [INFO] default - Repetitions: 2
[2019-06-09T17:17:33.011] [INFO] default - ====== CAPTURE FINISHED ======
```

Fig. 11: Capture messages on the device's terminal.

of the messages shown in the device's terminal during the signal capture.

8 Conclusions

This project has shown how to develop a system to remotely control a fleet of GNSS sensors included on the Cloud GNSS Rx project using the IoT platform provided by AWS. Still, it has shown how a system of this kind can be implemented in any kind of application. This document has explained how all the initial requirements have been fulfilled, and how the AWS IoT is a very useful platform to develop an IoT infrastructure.

This document has introduced first the problem to solve and it has contextualised the system by briefly describing the project within which it is included. Then, the technologies that have been selected to develop the project have been introduced, and the reason why these have been selected has been explained. Later on, the methodology and the planning that have been followed on this project has been explained, detailing the different tasks that have been followed during the development. After that, the AWS IoT platform has been described, which is the main technology that has allowed to implement the system. There, the main components of the platform have been described and some illustrative examples have been shown. Also, some related services have been explained, which are not actually used in this project but they are very interesting for future upgrades of the system that has been developed in this project. With the knowledge of the previous section, the architecture of the system has been deeply detailed next. There, the device-to-server communication has been explained first, showing

the AWS IoT Console that has been used during the development for test purposes. After this part of the architecture, the website-to-server communication has been described, including all the main steps that are followed on each action carried out from the website. Finally, the resulting system has been described, showing some descriptive images of the different operations that the system offers.

9 Future work

This project had very specific requirements defined by its inclusion on a bigger project, namely the Cloud GNSS Rx. Although all the requirements have been achieved, there are some doors that are kept open for further extensions. On the one hand, the first and obvious thing that should be done after the realisation of this project is the inclusion of the web dashboard that has been developed into the actual website of the Cloud GNSS Rx, which is already active. This inclusion will be done on the done shortly. On the other hand, some ideas have been arisen during the development of the project, which are presented next.

First, it would be interesting to obtain some hardware-information of the device such as the available battery capacity and send it to the IoT over the Devices-Control topic. This information could be then shown on the web dashboard.

Another interesting option appears from the availability of the AWS Greengrass and the AWS IoT Analytics services. These services could be used to add new features to the Cloud GNSS Rx or also to enhance the current effectiveness of the system. For example, the AWS Greengrass could be used to enable the communication with devices that have lost the connection with the Internet while they can still communicate with a near device.

Acknowledgements

I would like to express my gratitude to Prof. José A. López Salcedo for offering me the opportunity to work on the Cloud GNSS Receiver project at the SPCOMNAV group. I would also like to thank my family and friends for their personal support during the realisation of this project.

References

[1] L. Romero-Holguin, G. Seco-Granados, J. A. Lopez-Salcedo, and J. A. Garcıa-Molina, "Prototype of IoT GNSS Sensor for Cloud GNSS Signal Processing," in *6th International Colloquium on Scientific and Fundamental Aspects of GNSS / Galileo*, Oct. 2017.

[2] V. Lucas-Sabola, G. Seco-Granados, J. A. Lopez-Salcedo, J. A. Garcia-Molina, and M. Crisci, "Cloud GNSS receivers: New advanced applications made possible," in *2016 International Conference on Localization and GNSS (ICL-GNSS)*. Barcelona, Spain: IEEE, Jun. 2016, pp. 1–6.

[3] V. Lucas-Sabola, G. Seco-Granados, J. A. Lopez-Salcedo, and J. A. Garcıa-Molina, "GNSS IoT Positioning From Conventional Sensors to a Cloud-Based Solution," *Inside GNSS*, vol. May/June, pp. 53–62, May 2018.

[4] V. Lucas-Sabola, G. Seco-Granados, J. A. Lopez-Salcedo, J. A. Garcıa-Molina, and M. Crisci, "Computational performance of a cloud GNSS receiver using multi-thread parallelization," in *2016 8th ESA Workshop on Satellite Navigation Technologies and European Workshop on GNSS Signals and Signal Processing (NAVITEC)*. Noordwijk, Netherlands: IEEE, Dec. 2016, pp. 1–8.

[5] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of Things (IoT): A vision, architectural elements, and future directions," *Future Generation Computer Systems*, vol. 29, no. 7, pp. 1645–1660, Sep. 2013.

[6] J. Guth, U. Breitenbucher, M. Falkenthal, F. Leymann, and L. Reinfurt, "Comparison of IoT platform architectures: A field study based on a reference architecture," in *2016 Cloudification of the Internet of Things (CIoT)*. Paris, France: IEEE, Nov. 2016, pp. 1–6.

[7] U. Hunkeler, H. L. Truong, and A. Stanford-Clark, "MQTT-S — A publish/subscribe protocol for Wireless Sensor Networks," in *2008 3rd International Conference on Communication Systems Software and Middleware and Workshops (COMSWARE '08)*, Jan. 2008, pp. 791–798.

[8] "AWS SDK for Ruby Developer Guide - AWS SDK for Ruby." [Online]. Available: https://docs.aws.amazon.com/sdk-for-ruby/v3/developer-guide/welcome.html (Last accessed on 8 June 2019).

[9] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes, and J. Saraiva, "Energy efficiency across programming languages: how do energy, time, and memory relate?" in *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering - SLE 2017*. Vancouver, BC, Canada: ACM Press, 2017, pp. 256–267.

[10] "X.509 Certificates and AWS IoT - AWS IoT." [Online]. Available: https://docs.aws.amazon.com/iot/latest/developerguide/managing-device-certs.html#server-authentication (Last accessed on 8 June 2019).

[11] "File: README — AWS SDK for Ruby V3." [Online]. Available: https://docs.aws.amazon.com/sdk-for-ruby/v3/api/ (Last accessed on 8 June 2019).

[12] "Module: MQTT — Documentation for njh/ruby-mqtt (master)." [Online]. Available: https://www.rubydoc.info/github/njh/ruby-mqtt/MQTT (Last accessed on 8 June 2019).

[13] "Securing Rails Applications — Ruby on Rails Guides." [Online]. Available: https://guides.rubyonrails.org/security.html (Last accessed on 8 June 2019).