UNIVERSIDAD
DE LA REPUBLICA
URUGUAY

Tesis de grado

Aplicación de tecnologías de Internet de las Cosas para recolección de datos

Osimani, Felipe

Stecanella, Bruno

Ingeniería en Computación
Facultad de Ingeniería
Universidad de la República

Montevideo – Uruguay
Julio de 2018

UNIVERSIDAD
DE LA REPUBLICA
URUGUAY

Tesis de grado

Aplicación de tecnologías de Internet de las Cosas para recolección de datos

Osimani, Felipe

Stecanella, Bruno

Tesis de grado presentada en la Facultad de Ingeniería de la Universidad de la República, como parte de los requisitos necesarios para la obtención del título de grado en Ingeniería en Computación.

Directores:

Eduardo Grampín

Lorena Etcheverry

Montevideo – Uruguay

Julio de 2018

, Osimani, Felipe
Stecanella, Bruno
Tesis de grado / Osimani, Felipe
Stecanella, Bruno . - Montevideo: Universidad de la República, Facultad de Ingeniería, 2018.
x, 93 p.: il.; 29, 7cm.
Directores:
Eduardo Grampín
Lorena Etcheverry
Tesis de Grado – Universidad de la República, Ingeniería en Computación, 2018.
Referencias bibliográficas: p. 81 – 93.
1. IoT, 2. Computación en la nube, 3. Plataforma IoT en la nube, 4. Serverless, 5. Mobile Device Management. I. Grampín, Eduardo, Etcheverry, Lorena, . II. Universidad de la República, Ingeniería en Computación. III. Título.

INTEGRANTES DEL TRIBUNAL DE DEFENSA DE TESIS

Gabriel López

Libertad Tansini

Leonardo Vidal

Montevideo – Uruguay
Julio de 2018

Agradecimentos

En este proyecto, muchas personas nos proporcionaron su apoyo para poder llevarlo adelante, tanto técnico como emocional y espiritual. No fue fácil llevar este proyecto a cabo, y sin este apoyo probablemente no hubiera resultado posible. Le extendemos nuestro más profundo agradecimiento a todos.

En particular, nos gustaría agradecer al Plan Ceibal y a Germán Capdehourat por prestarnos tiempo, infraestructura, y atención a lo largo de este proyecto. También a Germán Hoffman y a Fernando Suzacq por el invaluable apoyo técnico que nos proporcionaron mostrándonos como se hace algo *de verdad*, y a Eugenio Rovira por enseñarnos cómo trabaja un data scientist.

Agradecemos también a Jim, por su firme soporte, y a todos nuestros otros compañeros de trabajo por escucharnos hablar de nuestros problemas, y por darnos ánimo para seguir adelante.

Debemos también agradecer especialmente a nuestros tutores Lorena Etcheverry y Eduardo Grampín por su apoyo, paciencia y orientación a lo largo del proyecto.

Finalmente, nuestro más profundo agradecimiento a nuestras familias, por el infalible apoyo y cariño que nos han dado siempre.

RESUMEN

Internet de las Cosas (IoT, por su sigla en inglés) es una idea que refiere a lograr la interconexión de objetos físicos, tanto estáticos como móviles, con el fin de que capturen y transmitan datos y/o respondan a comandos. Para lograr este fin, estos objetos integran sensores, actores, elementos de cómputo y software, además de la infraestructura de comunicación para lograr la integración. La popularidad de este concepto ha llevado a que salgan al mercado plataformas que simplifican el desarrollo de aplicaciones para IoT, como por ejemplo IBM Cloud IoT, Azure IoT Suite, AWS IoT, FIWARE y Kaa, entre otras. Asimismo, este paradigma presenta desafíos a nivel de tecnologías de interconexión de dispositivos, representación y manejo de datos en escala masiva.

En este proyecto, aplicamos las arquitecturas y plataformas de IoT que están surgiendo al problema de recopilación de datos de los dispositivos del Plan Ceibal. Generamos una prueba de concepto de un sistema que, utilizando una plataforma de IoT en la nube, se conecta a ceibalitas — potencialmente todas las del país, el sistema es autoescalable —, recopila en vivo datos de uso, y los almacena de manera estructurada, con la posible extensión de servicios de procesamiento de datos. Presentamos un análisis sobre los datos obtenidos al ejecutar este prototipo en 500 dispositivos reales a lo largo de dos meses.

Palabras clave:
IoT, Computación en la nube, Plataforma IoT en la nube, Serverless, Mobile Device Management.

Tabla de contenidos

Capítulo 1

Introducción

Internet de las Cosas (IoT por sigla en inglés) es uno de los conceptos que están más de moda en la industria de la computación. Para ponerlo en pocas palabras, la idea es llevar el mundo físico al mundo computacional a través de la interconexión de objetos que obtienen datos o interactúan con la realidad y centros de cómputo que procesan esos datos o manejan esas interacciones. Todo esto corre sobre una infraestructura de comunicación que abarca muchas tecnologías y protocolos de red de todas las capas[1].

Varios estudios apuntan a que para 2020 más de veinte mil millones de dispositivos estén conectados a Internet[2]. Esto es una muestra clave del crecimiento que ha tenido y que se estima tendrá el ecosistema IoT en el futuro. Este *boom* hizo que muchas empresas empezaran a ofrecer servicios para simplificar el desarrollo. Dentro de éstas tenemos a Watson IoT[3], Azure IoT[4], FIWARE[5], Kaa[6] y AWS IoT[7], entre muchas otras.

Este rápido crecimiento trajo consigo un problema de falta de estándares. Al no existir muchos en la industria, el desarrollo de nuevos proyectos en el ecosistema puede quedar muy ligado a la tecnología que se use, y puede verse obsoleto en poco tiempo con la introducción de nuevas prácticas o plataformas.

A grandes rasgos, la infraestructura de IoT tiene como función administrar cantidades masivas de dispositivos equipados de conexión a la red, que es una característica que la puede hacer adecuada para la administración de algunas facetas del plan Ceibal.

El Plan Ceibal es una organización creada en el 2007 como un "plan de inclusión e igualdad de oportunidades con el objetivo de apoyar con tecnología las políticas educativas uruguayas"[8]. Desde su comienzo, muchos disposi-

tivos diferentes fueron agregados al colectivo, desde laptops y chromebooks hasta tablets[9]. Además de distribuir laptops y otros dispositivos a escolares y liceales, Ceibal cuenta con su propia infraestructura de red que provee conectividad a los dispositivos[10]. Esta heterogeneidad de dispositivos puede hacer que llevar un registro del rendimiento (tanto de la red como de los dispositivos de los alumnos y docentes) sea una tarea de gran complejidad. Todos estos datos son muy relevantes para la toma de decisiones en cuanto a adquisición y actualización de dispositivos, por lo que tener acceso confiable a ellos es de gran importancia para la organización.

1.1. Motivación y problema

La enorme de cantidad de dispositivos diferentes que posee el Plan Ceibal en la actualidad — alrededor de 797.000 unidades[11] — tiene como consecuencia que llevar un registro del rendimiento de cada uno, así como el rendimiento de la red, sea una tarea de gran complejidad.

Ceibal cuenta con herramientas para dicha monitorización, pero son particulares de cada caso y no hace uso de las tecnologías presentes en la actualidad para atacar esta clase de problemas. Por lo tanto, este escenario resulta un caso de estudio interesante como enfoque para una investigación del ecosistema de plataformas IoT. Nos encontramos frente a un problema real y abierto, en donde podemos relevar las diferentes plataformas, considerar nuestras opciones, desarrollar una arquitectura para este caso particular y desarrollar el prototipo de una posible solución utilizando estas tecnologías.

Luego de esta experiencia, esperamos tener un entendimiento más profundo acerca del panorama actual de IoT y de cuáles son los requisitos para construir un sistema de IoT en el mundo real.

1.2. Objetivos

El objetivo principal de este trabajo es estudiar la situación actual del ecosistema IoT, concentrándonos en el problema de recolección de datos. Lograremos esto mediante un relevamiento de las plataformas ofrecidas por diferentes empresas y organizaciones, el desarrollo de una arquitectura de un sistema de recolección de datos y finalmente la implementación de un prototipo. Espera-

mos proveer al Plan Ceibal de un nuevo enfoque para solucionar el problema de recolección de datos, además de ganar la experiencia del desarrollo de un prototipo funcional que será desplegado en dispositivos reales.

1.3. Resultados esperados

Esparamos que el trabajo produzca los siguientes resultados:

- Estado del arte en lo referente a tecnologías del ecosistema IoT, junto con un relevamiento de las diferentes plataformas ofrecidas.
- El diseño de una arquitectura que facilite el diseño de soluciones a problemas de este tipo en el futuro, así como una perspectiva en cómo llevar a cabo la construcción de dichas soluciones.
- Un prototipo funcional desplegado en dispositivos reales (ceibalitas), que recolecte datos del rendimiento interno de cada una de ellas así como también del rendimiento de la red.
- Un pequeño análisis de los datos obtenidos al finalizar el período de prueba.

1.4. Estructura del documento

A continuación explicamos la estructura de lo que resta de este documento. En el capítulo 2 presentamos el estado del arte en lo referente a tecnologías IoT e investigamos el concepto de *plataforma IoT*. Hacemos especial énfasis en este último punto, ya que existe mucha heterogeneidad en las diferentes plataformas existentes. En el capítulo 3 presentamos el prototipo desarrollado para el Plan Ceibal. Explicamos las tecnologías utilizadas, las decisiones tomadas, el desarrollo y despliegue del prototipo y finalmente un pequeño apéndice que indica los pasos a seguir al considerar migrar el sistema a otra plataforma. En el capítulo 4 presentamos los resultados obtenidos, tanto de datos reales recolectados a lo largo del período de prueba, como del análisis del ecosistema de plataformas IoT y las consideraciones que se deben tener en cuenta al abordar un proyecto de este tipo. Finalmente, en el capítulo 5 presentamos las conclusiones que sacamos de la realización de este trabajo y algunas posibles líneas de trabajo futuro. Además, agregamos un capítulo con la bibliografía utilizada y anexos con manuales de usuario para el despliegue y el uso del prototipo.

Capítulo 2

Estado del Arte

En este capítulo presentamos un marco teórico con los conceptos más importantes para entender el proyecto, describiremos en detalle algunos de los principales estándares existentes de Internet de las Cosas (IoT) y haremos un estudio de las plataformas disponibles para el desarrollo de un sistema de recolección de datos, mencionando sus ventajas y desventajas.

2.1. Conectividad

En este sección presentamos diferentes arquitecturas y protocolos de comunicación utilizados en la industria hoy en día, referentes al ecosistema IoT. Nuestro análisis es específicamente sobre las tecnologías utilizadas en las capas superiores — de capa 5 a capa 7 del modelo OSI. Las tecnologías utilizadas en las demás capas fueron relevadas en detalle en otros estudios [12][13][14].

2.1.1. Arquitecturas

En primer lugar, los dispositivos precisan conectarse a alguna red ya sea para comunicarse entre ellos o con algún servidor. Para esto hay varias soluciones, dependiendo del tipo de dispositivo, y del tipo de aplicación. Los dispositivos pueden estar conectados directamente a internet, o conectarse con un *gateway* o *hub* quien sí se conecta a internet.

A modo de ejemplo, en el mundo del *Smart Home* (hogar inteligente), se pueden encontrar soluciones de ambos tipos. Para instalar lámparas inteligentes Philips Hue en un hogar es necesario adquirir un dispositivo (un *Bridge*) además de las lámparas. Todas las lámparas se comunican con el *Bridge*, quien

se comunica con el servidor. El servidor recibe las órdenes desde una app de celular controlada por el usuario, o de otras fuentes[15].

Por otro lado, Nest, que vende productos inteligentes como termostatos, alarmas, cámaras y timbres, no requiere ningún tipo de *hub*, sino que cada uno de los dispositivos se conectan directamente con el servidor, que recibe órdenes de manera similar[16].

2.1.2. Protocolos de comunicación

La comunicación en sí entre dispositivos — o *gateways* — y servidores sucede a través de internet, y los protocolos de capa de transporte utilizados suelen ser los mismos que para otras aplicaciones (principalmente TCP y UDP). Por otro lado, entre los protocolos de capa de aplicación utilizados suele aparecer HTTP, pero son muy comunes protocolos propios del ámbito de IoT como MQTT y otros menos populares como AMQP, XMPP, STOMP y CoA.

A continuación describimos en detalle las características del protocolo MQTT y algunas de las alternativas.

MQTT

MQTT (*Message Queuing Telemetry Transport*) es un protocolo de comunicación muy utilizado en el ámbito de IoT, por su simplicidad y por su eficiencia tanto en términos de ancho de banda como de energía. Implementa un patrón de mensajería *publish-subscribe*, y funciona sobre el protocolo TCP/IP.

Desde el año 2014 MQTT 3.1.1 es un estándar OASIS (Organization for the Advancement of Structured Information Standards) [17].

A diferencia de HTTP que es *request-response*, MQTT es *publish-subscribe*, lo cual le permite a los clientes conectados enviar mensajes, pero también recibirlos, mediante el uso de *push notifications*. Esto es, un cliente MQTT se suscribe a un canal (en MQTT esto se llama *topic*) y luego recibe los mensajes que se envíen a ese canal. La comunicación entre clientes no es directa, sino que sucede a través de un servidor que actúa como *message broker* (corredor de mensajes). Siempre hay una conexión TCP abierta entre el cliente y el servidor. Si la conexión es interrumpida, el servidor puede guardar los mensajes no enviados en un *buffer* y enviarlos al restaurarse la conexión.

Los mensajes tienen contenido y un *topic* objetivo. Un cliente envía un mensaje al *broker*, quien se encarga de reenviarlo a todos los clientes suscritos al *topic* del mensaje.

Los clientes se comunican entre sí a través de un *message broker* (corredor de mensajes) quien se encarga de reenviar un mensaje a todos los clientes suscritos al *topic* del mensaje Una vez que un cliente se conecta a un *broker*, puede enviar mensajes a *topics*, y suscribirse para que le lleguen los mensajes enviados a *topics*.

Un *topic* es información de ruteo para el *broker*. Los *topics* son jerárquicos, y en el nombre de un *topic* los niveles son separados por el caracter "/". Un cliente se puede suscribir a muchos *topics* usando caracteres comodín: + y #. El primero sirve para reemplazar un nivel, y el segundo cualquier cantidad de niveles — pero solamente puede ser utilizado al final de un nombre.

No hay necesidad que un *topic* sea creado para poder enviar mensajes, porque un *broker* acepta cualquier nombre válido sin inicialización previa.

Como ejemplo, en un caso donde tuviéramos una casa equipada con sensores conectados por MQTT, suscribirse al *topic* `Casa/+/Temperatura` tendría como resultado recibir mensajes de `Casa/Dormitorio/Temperatura`, `Casa/Cocina/Temperatura`, etc. Esta expresividad sirve para que el *topic* forme parte de la semántica del mensaje: en este caso los sensores mandan a *topics* asociados con su posición física, pero obviamente el uso que se le de depende del caso.

El protocolo provee algunas características avanzadas, que tienen que ver con el uso que se le da al protocolo en ambientes de baja fidelidad y fallas de conexión frecuentes. Los mensajes se pueden enviar con *Quality of Service* (QoS) 0, 1, 2, que da distintas garantías por parte del *broker*:

- **0** no da ninguna garantía acerca de que el mensaje llegará a destino.
- **1** garantiza que va a llegar a los destinatarios al menos una vez
- **2** garantiza que va a llegar a los destinatarios exactamente una vez.

QoS da un balance entre overhead de protocolo (más ancho de banda y energía) y garantías de que el mensaje llegará.

No todos los *brokers* soportan todos los niveles. Por ejemplo, los *brokers* de AWS IoT y Azure IoT soportan un QoS 0 y 1, pero no 2[18][19].

También es posible abrir una sesión persistente, que conserve datos de subscripción entre conexiones y almacene mensajes que fueron enviados mientras

el cliente estaba desconectado. Además, un cliente tiene la capacidad de configurar un mensaje de *Last Will and Testament* al conectarse, para que si se desconecta de forma abrupta ese mensaje sea enviado. Es posible configurar un mensaje retenido para un *topic*, y siempre que un cliente se suscriba a ese *topic* recibirá ese mensaje.

Por otro lado, la seguridad de la conexión no está dentro de las características de MQTT, que está diseñado para funcionar sobre TCP. Para encriptar la conexión, las implementaciones de *broker* permiten el uso de TLS, e incluso hay un puerto reservado, 8883 [20]. El uso de TLS obviamente agrega mucho *overhead* sobre el sencillo protocolo MQTT, pero si la red pública va a ser utilizada no hay mucha alternativa si se busca una conexión segura.

AMQP

AMQP (Advanced Message Queuing Protocol por sus siglas en inglés) es un protocolo de la capa de aplicación para *middleware* orientado a mensajería. Al igual que otros protocolos, posee colas de mensajería y enrutado de mensajes.

Este protocolo corre sobre un protocolo de transporte como TCP y puede estar autenticado tanto por SASL y/o TLS[21]. Al igual que MQTT, provee tres formas distintas de garantía del envío de mensajes, *at-most-once* (como máximo una), *at-least-once* (por lo menos una) y *exactly-once* (exactamente una).

Como diferencia principal encontramos que posee un cabezal con datos adicionales.

STOMP

La mayor diferencia de STOMP con el resto es que es un protocolo *text-based*, basado en texto. Esto lo acerca mucho más a HTTP que a MQTT y AMQP.

No lidia con *topics*, en vez de eso tiene algo denominado *destination string*. El *broker* debe ser capaz de interpretar dicho string para lograr la lógica deseada. Los componentes que deseen subscribirse al *broker* deberán también tener dicha lógica.

Algo que puede ser muy útil en algunos casos es que utilizando RabbitMQ[22] (un *broker* que tiene compatibilidad con STOMP), el mensaje mismo puede ser expuesto en un web socket[23]. Con esto se podría lograr que

una aplicación web reciba los mensajes directamente, sin intermediario.

CoAP

CoAP está fuertemente basado en el modelo REST fuertemente utilizado en HTTP. Un servidor expone servicios a través de una URL, y clientes los utilizan a través de operaciones como GET, PUT, POST, y DELETE. Esta similitud le permite transportar mensajes en el formato que se necesite, como JSON o XML.

Es bastante liviano, comprovado para su funcionamiento en microcontroladores con tan solo 10 KiB de RAM and 100 KiB para espacio de código[24]. Esta filosofía también la incorpora en los mensajes, teniendo un cabezal de 4 bytes y corriendo sobre UDP

2.2. Dispositivos

Los dispositivos de una red IoT están definidos por el caso de uso. El hardware de una red de cerraduras inteligentes será completamente distinto al de una red de sensores de movimiento, que a su vez será disinto al de un *smart TV*.

Se distinguen dos tipos de dispositivos: **sensores** y **actores**. Los sensores son dispositivos que miden una variable (física, de comportamiento, etc. periódicamente o disparado por eventos) y lo comunican, mientras que los actores reciben mensajes y realizan acciones[25]. Un dispositivo puede también ser de ambos tipos: por ejemplo, una cámara envía constantemente datos de video, pero también está escuchando por mensajes que la ordenen moverse o hacer zoom.

El hardware mismo de los dispositivos también es tan amplio como dispositivos hay. Algunos son computadoras hechas y derechas, por ejemplo, un Raspberry Pi tiene un procesador ARM y utiliza el kernel de Linux. Desarrollar software para esa plataforma no es muy diferente que desarrollar para cualquier computadora. Por otra parte, si el dispositivo está basado en un microcontrolador como Arduino las rutinas de comunicación serán escritas directamente sobre el hardware y dependerán de su arquitectura.

Por estas razones, la experiencia de desarrollar un sistema de IoT es altamente dependiente del caso de uso y del hardware a utilizar.

2.3. Plataformas

Los dispositivos deben conectarse a algún lado. Es raro que se comuniquen directamente unos con otros; lo normal para la mayoría de los protocolos es que los mensajes pasen por alguna entidad central. En MQTT, esta entidad es el *broker*, que administra las suscripciones a los canales, recibe todos los mensajes y se encarga de reenviarlos a los suscriptores apropiados.

Casos de usos reales suelen requerir que el servidor haga algo con esos mensajes. Esto puede ser almacenarlos en una base de datos, o interpretar el contenido de los mensajes y actuar en consecuencia, ya sea enviando otros mensajes o disparando acciones en otros sistemas. Esto quiere decir que para cada parte del sistema, es necesario desarrollar un módulo que comunique a esa parte con el *broker*.

Por ejemplo, si se quisiera almacenar todos los mensajes en una base de datos, habría que mantener un servicio que esté suscrito al canal apropiado y que reenvíe todos los mensajes a la base de datos. Si se quisiera generar una alerta en algún lado cuando la temperatura de cierto termómetro supere cierta temperatura, habría que mantener otro servicio que esté suscrito al canal apropiado y esté constantemente chequeando la temperatura de los mensajes que llegan.

Esto rápidamente se transforma en una gran cantidad de servicios similares entre sí que habría que desarrollar para cada aplicación de IoT. Es aquí donde entran las plataformas de IoT. El objetivo de una plataforma de IoT será simplificar el trabajo de los desarolladores a la hora de construir una aplicación de IoT, proveyendo implementaciones de los patrones comunes en IoT.

2.3.1. Características principales de las plataformas de IoT

Aunque las plataformas soportan protocolos estandarizados (ya sea MQTT para la comunicación o certificados X.509 para la autenticación), hacen la mayoría de las cosas de formas ligeramente diferentes entre sí. A continuación listamos una serie de características que tienen en común las plataformas IoT, ya sean de propósito general u orientadas a algún caso de uso particular. Pueden variar en nombre o implementación, pero prácticamente todas las plataformas de IoT proveen estos servicios de una forma u otra.

Recepción y envío de mensajes

La parte fundamental de una plataforma IoT es la capacidad de comunicarse con dispositivos. Para esto son normales los protocolos MQTT y HTTPS, habiendo alternativas menos populares reducidas a algunas plataformas particulares, como AMQP y CoAP.

Una plataforma provee una implementación de un *broker* MQTT o equivalente, facilidades para ejecutarlo y un *endpoint* para que los dispositivos se conecten a él.

Además, es normal que una plataforma imponga ciertas limitantes sobre los protocolos estándar. Por ejemplo, permitir sólo *topics* MQTT formateados de determinada manera, o no permitir el uso de ciertas características del protocolo, o incluso agregar características no estandarizadas por encima del protocolo.

Algunos ejemplos de las capacidades de envío y recepción de mensajes de diferentes plataformas se pueden encontrar en los protocolos soportados por AWS IoT[18] y en los *Transports* de Kaa[26].

Registro de dispositivos

Una plataforma IoT tiene la capacidad de llevar alguna forma de registro de los dispositivos que la usan. Algunas plataformas agrupan dentro de su concepto de"dispositivo' a todas las entidades que se pueden conectar en la plataforma (por ejemplo, los *things* de AWS, los *endpoints* de Kaa), ya sean dispositivos físicos, servicios externos, u otro tipo de programas. Otras plataformas las separan según los roles que los diseñadores de la plataforma esperan que sean de utilidad (la de IBM separa entre *application*, *device*, y *gateway*).

Este registro usualmente permite asignarle un identificador y otros atributos a cada dispositivo. Estos atributos pueden ser campos que describen al dispositivo, el tipo del dispositivo, las claves públicas de los certificados que el dispositivo utiliza para conectarse, los permisos de las acciones que puede realizar el dispositivo dentro de la plataforma, el estado actual del dispositivo, y cualquier otra cosa que pueda ser relevante dentro de la plataforma.

De esta forma un administrador puede (tanto manual como programáticamente) ver qué dispositivos están registrados, sus atributos, y realizar acciones sobre ellos, que pueden ir desde enviarles mensajes hasta cambiar sus permisos.

Algunos ejemplos de un registro de dispositivos son el *Thing Registry* de

AWS IoT[27], el *Identity Registry* de Azure IoT[28], y el *Endpoint Registry* de Kaa[29].

Descubrimiento de dispositivos

Otra parte fundamental es cómo los dispositivos se registran en la plataforma en primer lugar. Específicamente, cómo una entidad totalmente externa a la plataforma pasa a ser un dispositivo registrado dentro de la plataforma con certificados y permisos de publicación o suscripción y cómo ella discierne entre qué dispositivos aceptar en el registro y qué dispositivos rechazar. Cada plataforma utiliza su propia forma de autenticar dispositivos con algún tipo de credencial, y su propia forma de generar estas credenciales.

Aún así, nos encontramos con que si bien agregar un dispositivo en el registro de una plataforma suele ser un proceso sencillo, lograr que los dispositivos mismos negocien con la plataforma y sean registrados (sin dejar el registro abierto para cualquiera) es un proceso complejo que las plataformas — al menos las que analizamos — no resuelven directamente. Es decir, la plataforma es capaz de mantener un registro de dispositivos y proveer certificados, claves o algún otro tipo de secreto para estos dispositivos, pero no tiene por qué definir cómo es que llegan estos secretos a los dispositivos mismos. Explicamos en profundidad esta problemática más adelante, en la sección Registro de Ceibalitas.

Motor de reglas y eventos

Otra característica fundamental de una plataforma IoT es tener la capacidad de interpretar mensajes (u otros eventos que actúen como disparador), realizar acciones en consecuencia, y que el administrador tenga el poder de definir qué eventos desatan qué acciones. La capacidad y expresividad de este motor de reglas varía según la plataforma, pero en todas está presente en alguna medida — ya sea una implementación directa o como una integración de un servicio externo. Normalmente las reglas se escriben en forma declarativa ("cuando suceda tal condición tomar tal acción"), pero el tipo de reglas que se pueden escribir y las acciones que se pueden tomar dependen del uso esperado de la plataforma.

El principal disparador para las reglas suele ser la recepción de un nuevo mensaje, ya sea un mensaje cualquiera como uno que cumpla ciertas condiciones — que haya sido enviado por cierto dispositivo, o a cierto *topic*, o que

el contenido cumpla ciertas características. Otros disparadores pueden ser periódicos, o provenir de las integraciones externas de la plataforma.

Por otro lado, una de las acciones más comunes de un motor de reglas es enviar mensajes a los dispositivos. El objetivo es que no sea necesario desarrollar otro cliente que se conecte al *broker* para poder enviar mensajes a los dispositivos, sino que sea cuestión de agregar una regla que envíe un mensaje cuando sucede cierto evento. Aún así, sigue siendo responsabilidad del desarrollador que el dispositivo reciba los mensajes provenientes de la plataforma y actúe en consecuencia.

Las otras acciones posibles suelen involucrar cambiar algo en el registro de dispositivos, o comunicarse con alguna de las integraciones de la plataforma (desde bases de datos hasta servicios de análisis).

Algunos ejemplos de un motor de reglas son el *Rules Engine* de AWS IoT[30] y el *Event Processing* de FIWARE IoT[31].

Integración con otros servicios

Aunque las características anteriores alcanzan para tener todo un servicio de comunicación M2M que funcione de forma autónoma, para la mayoría de los casos de uso de valor, es necesario que los datos generados en base a los mensajes intercambiados puedan ir a algún otro servicio, o que otros servicios puedan disparar eventos en el motor de reglas.

La integración que suele estar presente en toda plataforma, es una facilidad para almacenar datos, ya sea directamente almacenar los mensajes, resúmenes de ellos, o metadatos. Una plataforma puede tener integrada directamente una base de datos, pero lo más normal es que provean facilidades de comunicación con bases de datos externas (tanto SQL como NoSQL).

Las integraciones de una plataforma de IoT son uno de sus diferenciantes más importantes. Servicios de análisis de datos, servicios de respaldo, telefonía, monitorización y otros pueden ser ofrecimientos de integraciones de una plataforma, que pueden resultar de valor a algunos clientes. Plataformas orientadas a resolver ciertos verticales pondrán énfasis en integraciones con servicios que tengan sentido para esos verticales.

El mayor *selling point* de AWS IoT no es las características en sí de la plataforma, sino la sencilla integración con el resto de los servicios en la nube de AWS. Esto es similiar en otras plataformas de la nube como Azure IoT;

hacen gran énfasis en la interoperabilidad con los otros ofrecimientos de la plataforma.

2.3.2. Otras características de las plataformas de IoT

Además de las principales, las plataformas de IoT proveen otras características que pueden resultar deseables según el uso que los diseñadores de la plataforma esperan que se le de. Listamos a continuación algunas de las más comunes:

Almacenamiento del estado de cada dispositivo

En algunos casos resulta deseable tener la capacidad de almacenar el estado actual de un dispositivo en la plataforma. Esto es particularmente útil para dispositivos actores, que reciben mensajes y realizan acciones en base a esos mensajes. Un dispositivo de este tipo cuenta con alguna especie de máquina de estados interna que dicta qué acción puede realizar, y lo que las plataformas suelen ofrecer es una forma de guardar ese estado dentro de la plataforma, y sincronización automática con el estado del dispositivo.

AWS llama a esto *Thing Shadows* [32] y Azure lo llama *Device Twins* [33].

Una interfaz gráfica de administración

Las plataformas de IoT suelen contar con una interfaz gráfica (usualmente web) para que un administrador pueda gerenciar la plataforma sin recurrir a archivos de configuración o una interfaz de línea de comandos. Normalmente es posible acceder a resumenes de uso y a configuración de las características de la plataforma. Las plataformas suelen contar también con una API, para poder realizar todas estas acciones programáticamente[34][35][36].

Facilidades de disponibilidad (implica infraestructura)

Una plataforma de IoT puede incluir su propia infraestructura, y garantizar cierto nivel de disponibilidad. Las plataformas en la nube (por ejemplo, AWS IoT, Azure IoT Hub e IBM Watson IoT) ejecutan siempre remotamente, en las granjas de servidores propia del proveedor. Por otra parte, otras plataformas (por ejemplo, Kaa) son simplemente un software para instalar en un

servidor propio. No es parte de su negocio proveer los servidores ni garantizar un servicio.

Facilidades para realizar actualizaciones de *firmware* en los dispositivos

Una característica importante de las redes IoT es que suele ser necesario actualizar el *firmware* de los dispositivos. Ya sea para arreglar defectos, agregar nuevas características, cambiar los certificados de seguridad, o actualizar otros programas dentro del dispositivo (como el sistema operativo en caso de contar con uno), una actualización implica un complicado y riesgoso proceso. Algunas plataformas proveen facilidades para llevar registro de qué versiones están instaladas en qué dispositivos y desplegar actualizaciones.

Una plataforma que provee esta funcionalidad es la de IBM [37].

Integración con hardware específico

Una característica que algunas plataformas de IoT usan para diferenciarse es integraciones con hardware específico. Normalmente una plataforma provee SDKs en varios lenguajes para facilitar la integración un poco más que desarrollar directamente sobre el protocolo base, pero algunas plataformas llevan esto un paso más. Kaa por ejemplo provee binarios que funcionan en chips específicos como Intel Edison y UDOO [38]. Donde en otras plataformas sería necesario un importante trabajo para poder comunicarse con este hardware, aquí la compatibilidad la proveen ellos.

Ligado al punto anterior, si la plataforma conoce el hardware con el que trabajan los dispositivos, puede proveer también facilidades para actualizar el firmware específico del chip a través de los SDKs personalizados.

2.4. Amazon Web Services

En esta sección introducimos la arquitectura general y los conceptos clave de los servicios de Amazon Web Services (AWS) que utilizamos en el desarrollo del proyecto.

AWS es una plataforma en la nube que ofrece una gran gama de servicios, en los que se incluyen poder de cómputo, servicio de bases de datos y almacenamiento de archivos.

2.4.1. Arquitectura general

La plataforma se puede dividir en varias partes

- *AWS Global Infrastructure*: Engloba todos los servicios de AWS. Es un conjunto de *datacenters* distribuidos alrededor del planeta. Actualmente se encuentran en 16 regiones diferentes, con 44 *availability zones* — *datacenters* separados dentro de cada región. Nuevas regiones son agregadas todos los años.
- *Compute*: Servicios de poder de cómputo. Incluye tanto máquinas virtuales de diferentes prestaciones como procesamiento en la nube. El servicio más utilizado es *EC2* (*Elastic Compute Cloud*), proporcionando máquinas virtuales y físicas de diferentes características para uso general. Otro servicio popular dentro de *AWS Compute* es *AWS Lambda*, en el cual nuestro proyecto se apoyó fuertemente. Provee una forma de ejecutar código sin administrar servidores, al instante, escalando sin intervención de personas y cobrando solamente por el tiempo de cómputo y cantidad de invocaciones. Esto es explicado en profundidad en la sección AWS Lambda
- *Almacenamiento*: Servicios de almacenamiento de archivos. AWS tiene varios ofrecimientos, que van desde almacenamiento simple de archivos en *S3* (*Simple Storage Service*), hasta servicios de sistemas de archivos en la red en *EFS* (*Elastic File System*), que es un sistema de archivos que puede ser adjuntado a varias máquinas virtuales.
- *Bases de datos*: Servicios de bases de datos. AWS ofrece bases de datos relacionales en el servicio llamado *RDS* (*Relational Database Service*), DynamoDB, una base de datos no SQL auto escalable propietaria, *ElastiCache*, servicio de cache que puede correr bases de datos en memoria como *redis* o *memcache* y *RedShift*, almacén de datos del orden de peta bytes, entre otros. Detalles del funcionamiento de *RDS* y DynamoDB pueden encontrarse en la sección AWS RDS y DynamoDB.
- *Redes de computadoras*: AWS ofrece servicios como *VPC* (*Virtual Private Cloud*), que provee formas de crear un *datacenter* privado virtual en la nube, *route53* (servicio de DNS), *CloudFront* (distribución de contenido en el globo), *API Gateway* (servicio de creación y publicación de APIs RESTFul), entre otras. Detalles de API Gateway se encuentran en la sección API Gateway.

- *Seguridad*: El servicio más importante dentro del área de seguridad en AWS es *IAM* (*Identity Access Management*). Es un servicio centralizado para el control de acceso tanto de usuarios de AWS como de otros servicios. Se utilizan políticas basadas en identidad (usuarios, grupos, funciones) o recursos (expresadas en un documento *JSON* estandarizado, donde se controla qué recurso y qué función dentro de ese recurso puede ser utilizada por quién).

  El servicio de autenticación de dispositivos para este proyecto lo desarollamos dentro del *broker* MQTT, utilizando servicios propios de este. Detalles de esta implementación, así como del *broker* MQTT se encuentran en la sección AWS IoT.
- *AWS IoT*: Incluye *Plataforma AWS IoT* (utilizado en este proyecto), servicios de administración de dispositivos *AWS IoT Device Management*, sistemas operativos para microcontroladores (*Amazon FreeRTOS*) y por último *AWS Greengrass* (capacidades locales de cómputo, mensajería, almacenamiento de datos en caché y sincronización e inferencia de aprendizaje automático para dispositivos conectados). En este proyecto utilizamos muy fuertemente los servicios utilizados por la *Plataforma AWS IoT*, y está explicado en detalle en la sección AWS IoT.

Existen muchos otros servicios más que van más allá del alcance de este proyecto. Estos incluyen servicios de aprendizaje profundo, integración de aplicaciones (cola de mensajería, servicio de mensajería, entre otros), servicios de análisis de datos, procesamiento multimedia, etc.

A continuación detallamos las principales características de los componentes que utilizamos en el proyecto, incluyendo el modelo de facturación y costos de cada uno.

2.4.2. AWS Lambda

Según AWS, la premisa de Lambda es la siguiente: *"AWS Lambda permite ejecutar código sin aprovisionar ni administrar servidores"* [39]. Es la implementación de AWS de las arquitecturas *serverless* (sin servidor), y es el mayor exponente de esta arquitectura.

"Sin servidor" no quiere decir que literalmente no haya servidores, sino que están ocultos para el desarrollador. En una arquitectura de este tipo, el desarrollador escribe los programas y el proveedor se encarga que se ejecuten

de manera correcta en sus servidores, pero el desarrollador nunca se entera de lo que está sucediendo en la sala de máquinas. Es el paso siguiente a los servidores virtualizados.

Lambda provee una plataforma para ejecutar código de forma autónoma en un ambiente de alta performance, alta disponibilidad, consumiendo los recursos y el tiempo de cómputo necesario y nada más, dejando al desarrollador la posibilidad de programar en un lenguaje de programación a elección. Basados en esta premisa, problemas de escalabilidad y requerimientos de rendimiento quedan completamente por fuera del análisis de cada proyecto.

Para utilizar Lambda uno tiene que tener dos consideraciones, cuándo y cómo se va a disparar (utilización de *triggers*) y qué se va a hacer con la salida de la función. Como *triggers* uno puede utilizar una llamada HTTPS a través de *API Gateway*, cambios en bases de datos dentro de AWS (tanto dentro de *RDS* como de DynamoDB, ambas explicadas en profundidad en la sección AWS RDS y DynamoDB), colas de mensajería, envío de mails, otras funciones Lambda, etc. Al mismo tiempo, la función puede llamar a cualquier otro servicio dentro o fuera de AWS una vez realizado el procesamiento deseado.

Una vez invocada la función esta queda en estado *warm* (caliente), guardando varias variables en un caché (invisible para el usuario), que permite que la siguiente invocación sea más rápida. Un ejemplo de esto es la creación de conexiones con bases de datos relacionales — una conexión se mantiene abierta entre invocaciones a la función, aumentando considerablemente el rendimiento.

Costos

Lambda posee un sistema de cobro por invocación, tiempo de uso y cantidad de RAM asignada (que es un parámetro establecido por el desarrollador). El primer millón de invocaciones por mes es gratis, junto con 400,000 GB-segundos (un GB-segundo es un segundo de uso de un GB de RAM). Luego de pasados los niveles gratuitos, se cobra USD 0.2 por cada subsecuente millón de invocaciones y USD 0.00001667 por cada GB-segundo.

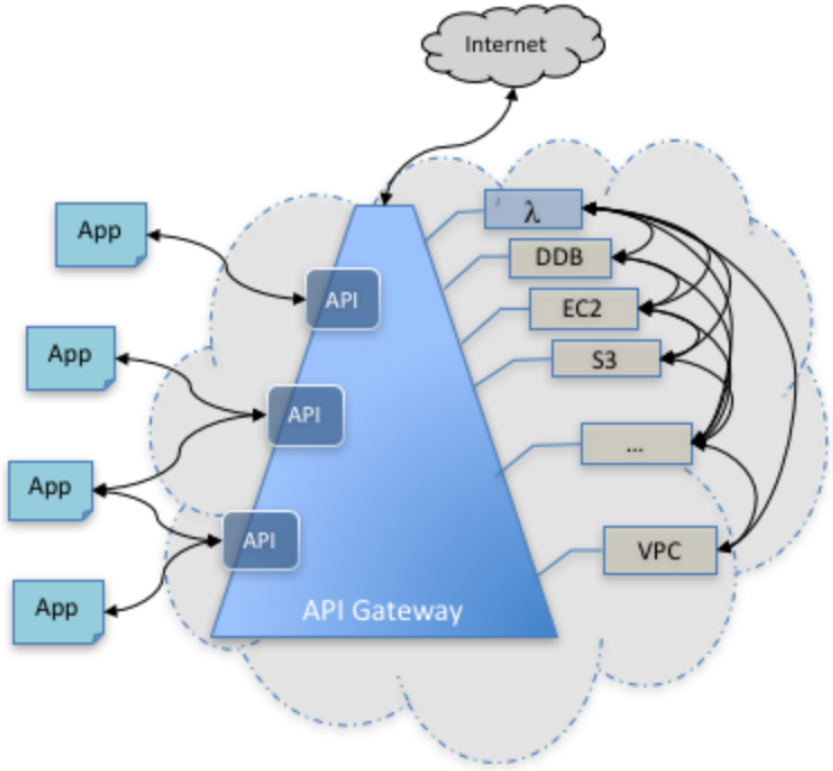
Para este proyecto calculamos no superar el nivel gratuito, logrando hacerlo sin inconvenientes.

2.4.3. AWS API Gateway

API Gateway es un servicio de creación de API RESTFul para permitir a agentes externos acceso programático a los servicios de la nube de AWS.

El diagrama representado en la figura 2.1 muestra la arquitectura básica de *API Gateway*, mostrando la interacción que tiene tanto con aplicaciones externas como con servicios de AWS.

Figura 2.1: Diagrama de interacciones de API Gateway (Amazon APIGateway 2018[40]



La integración con los servicios de AWS, incluyendo su configuración, no requiere mucho trabajo por parte del desarrollador. Por ejemplo, para utilizar *API Gateway* como proxy de una función Lambda el desarrollador simplemente tiene que indicarlo en una interfaz gráfica y AWS creará la API y el *endpoint* para futuras comunicaciones por parte de consumidores externos de esa función.

Costos

AWS provee un millón de *requests* gratuitas por mes, a partir de las cuales normalmente cobra USD 3,50 por cada millón de *requests* extra (dependiendo de la *availability zone* que se encuentre la API). En cuanto a transferencia de datos, los costos son los siguientes: [41]

- USD 0,09/GB para los primeros 10 TB
- USD 0,085/GB para los siguientes 40 TB
- USD 0,07/GB para los siguientes 100 TB
- USD 0,05/GB para los siguientes 350 TB

Existen también opciones de caché para las API, con costos dependiendo del tamaño del caché requerido.

2.4.4. AWS RDS y DynamoDB

AWS RDS

El servicio provee administración y manejo de base de datos relacionales (Amazon Aurora, MySQL, MariaDB, Oracle, Microsoft SQL Server y PostgreSQL), con especial énfasis en la automatización de tareas administrativas como actualización, detección de fallas y suministro de recursos.

Posee varias características que la diferencian de un sistema de bases de datos tradicional (un DBMS corriendo en un servidor tradicional), pero que en general son similares a las de otros servicios de bases de datos en la nube. Entre ellas encontramos las siguientes:

- Varios tipos de instancias. Ya sean con gran capacidad de memoria o normales, existen una enorme cantidad de especificaciones.
- Escalabilidad. Opciones de aumento de tamaño de la base de datos instantáneo, y de aumento de especificaciones en cuestión de minutos.
- Opción de replicación de datos en varias *availability zones* automática.
- Modalidad de pago por uso, pudiendo cancelar, o cambiar especificaciones que auenten o bajen el precio total.

En cuanto a costos, varía dependiendo de varias variables, entre las que se encuentran qué manejador de base de datos es utilizado, tamaño, prestaciones, replicación de datos y total de transferencia de datos. [42]

AWS DynamoDB

DynamoDB es una base de datos NoSQL propietaria de Amazon [43]. Su principal característica es la profunda integración que tiene con otros servicios de AWS. Por ejemplo, DynamoDB se integra automáticamente con AWS IoT, permitiendo desde la interfaz gráfica crear una regla que guarde todo un

mensaje recibido en una nueva línea de la tabla — hacer la misma acción pero guardando en otras bases de datos requieren el uso de una función Lambda.

Promete latencias con el servidor de menos de 10 milisegundos, con opciones de caché que reducen aún más el tiempo al orden de los microsegundos. Esto es prometido en cualquier escala, teniendo algoritmos automatizados de partición de datos cuando se alcanzan ciertos volúmenes. Combinado con sus sistemas de auto escalabilidad (aumento automático de la capacidad de cómputo dependiendo de la demanda) hace que sea una destacada opción para sistemas IoT.

En lo referente a costos [44], DynamoDB cobra dependiendo de las capacidades de cómputo (llamadas *Write Capacity Unit (WCU)* y *Read Capacity Unit (RCU)*) y el tamaño total de la base de datos. Como referencia, una WCU es capaz de realizar una escritura por segundo, mientras que una RCU puede realizar una lectura por segundo. Si la base de datos tiene la opción de auto escalabilidad, el precio varía por mes dependiendo de cuánto se hayan aumentado o reducido las WCU y RCU en el correr del mes.

2.4.5. AWS IoT

Como discutimos en la sección Relevamiento de plataformas IoT, AWS IoT es una plataforma de propósito general, sin ningún escenario vertical en particular.

Usabilidad de la plataforma

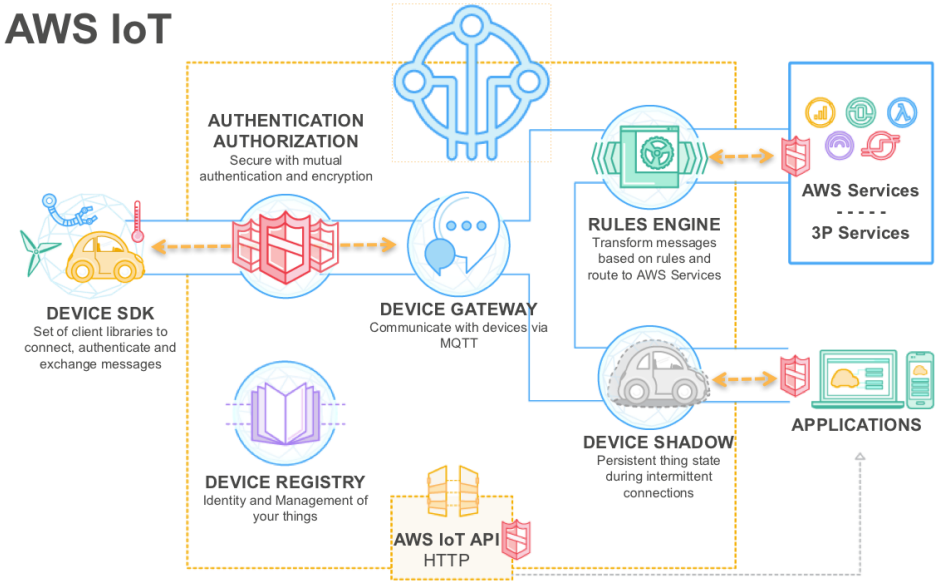
A diferencia de muchas otras plataformas relevadas, AWS es la que mejor usabilidad general tiene. La página y *dashboard* provistos son intuitivos, y si bien en el curso del desarrollo del proyecto cambió su diseño más de una vez, nunca fue dificultoso utilizarlo.

Al igual que todos los otros servicios provistos por AWS, la documentación de la plataforma es organizada, extensa y bien detallada. Si a esto le sumamos el hecho que existe una comunidad muy grande de desarrolladores que utilizan la plataforma, nunca nos fue difícil resolver los problemas que fueron surgiendo.

Terminología y conceptos claves

Dentro de la plataforma, los dispositivos son llamados *things*. La plataforma cuenta con un registro de dispositivos, el *thing registry*, donde la información acerca de los dispositivos está almacenada. Los *things* pueden poseer atributos para describirlos y certificados que pueden utilizar para conectarse a la

Figura 2.2: Diagrama de componentes generales de la plataforma (Amazon IoT 2018[45]



plataforma. A los certificados se les puede asignar *policies* (políticas), que especifican en un documento JSON, a través del lenguage de permisos de AWS, qué permisos tiene un agente que se conecte utilizando ese certificado dentro de la plataforma.

Como visto en otras plataformas, AWS IoT posee un sistema opcional de manejo de estado para los dispositivos llamado *device shadow*. Para interactuar con él, los diferentes dispositivos o programas publican en un topic particular, y automáticamente la plataforma se encarga de actualizar el estado en todos los lugares pertinentes. Es particularmente útil cuando los dispositivos del sistema reciben y mandan datos, permitiendo de una forma organizada el control de información y ejecución de acciones.

Para el manejo de eventos, la plataforma posee un sistema de reglas llamado *rules engine*. Está basado en un lenguage pseudo SQL para la obtención de datos de los mensajes, para su posterior envío y procesamiento a varios servicios de AWS, incluyendo *Lambda*.

Autenticación

Como mencionamos anteriormente, cada dispositivo se autentica a través del o los certificados que tiene asociados. Se pueden utilizar certificados *X.509* propios, o AWS puede crearlos, firmarlos, y adjuntarlos a los *things*.

La plataforma posee también una opción de autenticación personalizada.

A través de una función *lambda*, se puede desarrollar un sistema de autenticación personalizado utilizando verificación a través de JWT [46], OAuth [47], o cualquier otro medio que se desee. El proceso debe retornar un JSON con las *policies* que el *thing* tiene que tener adjuntadas.

Costos

Detallamos a continuación el modelo de costos, separado por las diferentes variables que lo afectan:

- **Tiempo de conexión.** AWS cobra por tiempo de conexión, USD 0,08 por cada minuto de conexión de cada dispositivo.
- **Cantidad de mensajes.** El modelo de precios funciona de la siguiente manera: los primeros 500.000 mensajes son gratis, y luego son USD 5 por cada millón de mensajes extra. Si un mensaje pesa más de 1 KB, cuenta como un mensaje extra por cada 1 KB por encima del primero. Una cosa a tener en cuenta es que la conexión MQTT cuenta como un mensaje extra.
- **Registro de dispositivos.** Los precios del uso del registro son por cantidad de modificaciones del registro y de las *Thing Shadows*, costando USD 1,25 por cada millón de modificaciones. Estas se cobran también por peso, cada KB que pese la modificación se cobra como una independiente.
- **Motor de reglas.** Las llamadas al motor de reglas también son cobradas. El precio es USD 0,15 cada millón de reglas disparadas y cada millón de acciones ejecutadas, por lo cual en nuestro caso — 2 millones de mensajes, cada uno dispara una regla y ejecuta una acción — el total no llega a superar USD 1.

Capítulo 3

El prototipo de Ceibal

En este capítulo presentamos la motivación y requisitos del prototipo planteado por Plan Ceibal. Realizamos un análisis y relevamiento de las plataformas IoT disponibles en el mercado, y planteamos las razones detrás de elegir una sobre las otras. Pasamos por un desarrollo del prototipo, arquitectura diseñada y despliegue para terminar con problemas encontrados y una pequeña guía para la eventual migración del prototipo a otra plataforma.

3.1. Motivación

En el marco de realizar un relevamiento de las plataformas de IoT e investigar qué son, buscamos un caso de uso para ponerlas a prueba, con el propósito de experimentar el proceso de construcción de una aplicación real. Dicho caso de uso será un marco común de referencia para tanto la experimentación como la comparación de dichas pruebas. Esto nos daría una visión más clara de por qué las plataformas están diseñadas de la forma en la que están, y qué características son deseables que una plataforma tenga.

Plan Ceibal (en adelante *los clientes*) nos plantearon un problema de recolección de datos. En un principio el proyecto consistía en relevar la calidad de servicio de las redes del Plan Ceibal (que son su propio proveedor de internet) usando los mismos dispositivos del Plan Ceibal — las ceibalitas. Es decir, los dispositivos miden cómo ven la red y lo reportan en vivo a un servidor central, que luego analiza y saca conclusiones acerca del estado de la red y lo almacena con motivos de generar un historial. Esta aplicación es útil en caso de tener muchos dispositivos en una red ajena a la propia red de uno. De esta forma se

puede saber si el que vende la conectividad está realmente proveyendo lo que dice proveer, y en general tener una idea de cómo funciona la red sin tener directamente acceso a los datos con los que cuenta el proveedor.

Aunque desarrollar este caso de uso nos sería de gran utilidad para poder ver cómo se comportan las plataformas de IoT en la realidad, a los clientes realmente no les es de mucho interés porque al ser ellos sus propios proveedores de internet, ya cuentan con la mayoría de los datos que podamos recabar. En otras palabras, ya tienen una idea de cuál es la calidad del servicio de red en los centros que administran y no precisan otra forma de medirlo.

En su lugar, lo que a ellos les interesa conocer es el uso en sí de los dispositivos. Cuando los clientes adquieren dispositivos nuevos, realizan una licitación durante la cual ejecutan tests para decidir qué dispositivos comprar, pero una vez desplegados en el campo no saben la realidad empírica de su desempeño. ¿Cuánto dura la batería? ¿Es suficiente el tamaño del disco? ¿Cuál es la carga promedio del CPU? Este es el tipo de preguntas que a los clientes les gustaría poder responder.

Viendo las ceibalitas como una red de sensores que recolectan datos, tiene sentido elaborar una solución en base a una plataforma de IoT. Ellos ya habían dado algunos pasos tentativos en la dirección de recolectar estadísticas de uso de las máquinas (utilizando el sistema de actualizaciones ya implementado), pero les interesaba ver un sistema de monitorización con otra óptica. De esta forma, aprovecharíamos las ventajas que otorga una plataforma de IoT en la nube: escalabilidad, disponibilidad y usabilidad. Los clientes luego desplegarían esta solución en algunos cientos de computadoras para poder probar la solución en un entorno real. Esto logró hacerse sin mayores inconvenientes y resultados del despliegue pueden encontrarse en la sección Resultados.

3.2. Requisitos

Los clientes nos plantearon varios requisitos alrededor de los cuales desarrollaríamos nuestro prototipo. El principal interés para ellos son los datos a recolectar, para lo cual nos proveyeron de una lista. Ella contiene tanto métricas de rendimiento y uso del dispositivo como de rendimiento y datos de la red. La tecnología y sistema de implementación en general los dejaron a nuestro criterio, pues esa parte en la que radica la innovación de nuestro proyecto.

3.2.1. Datos a recolectar

Los datos a recolectar son en vivo y con cierta resolución, que les gustaría poder especificar, pero en un principio la fijamos en 5 minutos. Es decir, que cada 5 minutos el dispositivo tome medidas y las envíe al servidor central. De las medidas mismas, los clientes distinguen tres tipos: del funcionamiento del dispositivo y su desempeño, de la red y su desempeño, y del uso del dispositivo y sus aplicaciones. Además, nos proveyeron de sugerencias acerca de cómo realizar esas medidas en su hardware.

Medidas del funcionamiento del dispositivo

Estas medidas son acerca del desempeño del hardware mismo. En esta categoría se destacan medidas como el número serial (identificador de cada ceibalita), tiempo en funcionamiento, uso de CPU, uso de memoria, espacio libre en disco, tiempo de arranque, nivel de batería y su temperatura, y monitorización de uso de recursos por parte de aplicaciones (es decir, qué aplicaciones son las que usan más recursos). Teniendo una monitorización continuo de estas medidas es posible saber aproximadamente, para cada dispositivo, si la experiencia que está teniendo el usuario es buena o está siendo obstaculizada de algún modo.

Medidas de la red

Estas medidas son acerca de la red a la que la computadora está conectada. En esta categoría se destacan la dirección MAC (identificador de la tarjeta de red de la ceibalita), la dirección IP pública de salida a internet, los datos del *Access Point* y conexión específicos a *WiFi*, el tráfico en cada interfaz de red del dispositivo, las redes vistas por el dispositivo, y algunas medidas activas de latencia y *throughput* a distintos sitios de internet.

Un escenario en el cual este sistema sería particularmente útil es el de tener información del rendimiento de una red en una escuela, por ejemplo. Las ceibalitas recolectan y envían los datos, obteniendo una imagen en tiempo real del estado de la red, sin precisar acceso a los dispositivos de borde. Además, podríamos conectar el sistema a servicios de análisis de eventos complejos que levanten alertas y realicen otras acciones en base al estado de la red.

Medidas del uso de dispositivo y sus aplicaciones

Estas medidas son específicas del uso de aplicaciones. En otras palabras, a qué aplicaciones le da uso el usuario y con qué frecuencia. Esto les sería útil para la evaluación de las diferentes aplicaciones adquiridas. A los clientes les interesa saber esto porque adquieren y publican aplicaciones, y quieren saber si los usuarios efectivamente les dan uso. Estas medidas quedaron descartadas para este proyecto, pero hipotéticamente no sería complejo incluirlas.

3.2.2. Tecnología

Como nuestro proyecto se basa en la tecnología a utilizar, la decisión de qué y cómo utilizar fue dejado a nuestro criterio. Ahora, como probablemente utilizáramos tecnología propietaria en una nube de otro país, y a los clientes les interesa seguir las regulaciones de privacidad de datos, determinamos que una migración a otra plataforma debería ser lo más sencilla posible. Es decir, intentaríamos no aprovecharnos de características exclusivas de la plataforma elegida de no ser necesario.

A pesar de esto, hay algunas cosas a tener en cuenta para la elección de la tecnología. Los clientes cuentan con 797.000 dispositivos[11] (entre laptops y tablets para escolares, liceales y jubilados) que eventualmente les interesaría monitorizar. Aunque en nuestro prototipo no sería necesario soportar nada cercano a ese volumen de dispositivos, la tecnología elegida debería ser capaz de escalar para llegar a ese nivel de demanda, y de ser posible, sin mucha intervención de administradores.

3.2.3. Distribución

Los dispositivos en los que se ejecuta la recopilación de datos son laptops del Plan Ceibal, que cuentan con Ubuntu 12.04, 14.04 y 16.04. Los clientes se encargan de la distribución del código en estos dispositivos, y para esto nos pidieron que les entregáramos un paquete de instalación de Debian. Este paquete debe ser autosuficiente, y luego de la instalación debe quedar ya configurado para recopilar datos y enviar mensajes periódicamente.

Para probar la compatibilidad del código nos prestaron dos ceibalitas.

3.3. Relevamiento de plataformas de IoT

En esta sección detallaremos las pruebas que hicimos con varias plataformas de IoT para decidir cuál utilizar para desarrollar el prototipo.

3.3.1. Metodología de prueba

Tratamos de ejecutar un *hello world* en cada plataforma, viendo las diferencias y la dificultad de pasar el software del dispositivo de una a la otra. En primer lugar es necesario definir qué es un *hello world* en una plataforma IoT. Para esto nos basamos en las características de la plataforma usadas en los tutoriales introductorios de cada una, y creamos una prueba que correríamos en cada una de ellas a modo de test.

Identificamos en un principio que todas las plataformas de IoT tienen las mismas características básicas — conectividad, registro de dispositivos, autenticación, etc —, pero en todas se acceden de diferentes formas. Para esta prueba definimos dos objetivos: identificar cómo se realizan las acciones que las plataformas tienen en común e identificar las diferencias entre las plataformas. La prueba consiste en, para una plataforma dada, realizar las siguientes tareas:

- Agregar un dispositivo a la plataforma.
- Conseguir credenciales de autenticación para ese dispositivo.
- Desde ese dispositivo, conectarse a un *broker* MQTT de la plataforma (usando *SSL*), y enviar un mensaje a un topic. Para las pruebas, nuestros dispositivos fueron computadoras. Luego, suscribirse a ese topic desde la interfaz web (en caso de haber una consola MQTT web en la plataforma) y recibir el mensaje correctamente.
- Almacenar automáticamente los mensajes enviados a cierto topic en una base de datos.
- Desde el dispositivo, suscribirse a un topic. Desde la interfaz web, enviar un mensaje a ese topic y recibirlo correctamente en el dispositivo.
- Agregar y conectar otro dispositivo distinto. Desde ese dispositivo, enviar un mensaje a un topic para que el otro lo reciba, y viceversa.

Al realizar todas estas tareas, tenemos un panorama general bastante amplio de la plataforma. Sabemos cómo funciona el registro de dispositivos, la obtención y utilización de credenciales, la configuración y testeo del *broker*

MQTT, la gestión de los permisos de suscripción y publicación a diferentes topics, el motor de reglas y la conexión a servicios externos — una base de datos.

3.3.2. Plataformas relevadas

Con estas tareas en mente, relevamos con esta metodología algunas plataformas de IoT en la nube que mostraban potencial en el desarrollo del prototipo.

Amazon Web Services IoT

AWS es la principal plataforma de computación en la nube del mercado[48], y cuenta con un ofrecimiento de IoT que se integra con el resto de su plataforma. Logramos ejecutar todas las tareas propuestas con pocos problemas, y en general nos dejó una buena impresión. Al ser la plataforma elegida para el prototipo, detalles de la plataforma se pueden encontrar en la sección AWS IoT.

IBM Cloud Watson IoT

Watson IoT es el nombre que IBM decidió ponerle a su plataforma de IoT dentro de su nube. Conocida anteriormente como IBM Bluemix IoT, es el ofrecimiento de IBM como manejador de una infraestructura de IoT.

Actualmente, IBM posee 6 % del mercado de las plataformas de internet[48], muy por debajo de otros competidores como Amazon o Microsoft.

Usabilidad de la plataforma web

La primer cosa que salta a la vista es la muy mala usabilidad de la interfaz. Los ofrecimientos en la nube de IBM están muy segmentados, con diferentes partes teniendo nombres distintos e incluso estilos de diseño completamente diferentes. En particular, para acceder a Watson IoT es necesario pasar por dos pantallas de *login*: primero para entrar a IBM Cloud, y una vez ahí encontrar el link a Watson IoT e introducir otra contraseña. El hecho que el flujo básico no esté estandarizado nos da a entender que la plataforma tiene una dependencia muy grande a su estructura y forma de trabajo.

Terminología y conceptos claves

Tiene una terminología mucho más específica que AWS, si bien en el fondo corre un *broker* MQTT y diferentes programas se suscriben y publican en él, quieren forzar una forma muy particular de pensar cada dispositivo y cada flujo, haciendo más difícil llevar a cabo la implementación del test básico que preparamos. Sobre este problema nos encontramos además que tampoco tienen documentación adecuada que explique la arquitectura esperada por ellos, su forma de ver cómo deberían ser las cosas.

Estructura de publicación

Diferencian entre *events* y *commands*. La idea es que un *event* es un dato que genera un dispositivo sensor, y un *command* es un comando para un dispositivo actuador. Algo que se podría hacer fácilmente a mano lo diferencian y hacen que la escritura de publicación de uno u otro sea más compleja. Para dar un ejemplo, si un dispositivo quiere mandar un *event* lo debe publicar al topic

```
iot-2/evt/{event type}/fmt/{format type}
```

pero si quiere hacer un *command* entonces será

```
iot-2/cmd/{command type}/fmt/{format type}
```

Lo mismo sucede con las aplicaciones conectadas al sistema (no dispositivos *per se*).

Además de la estructura mencionada anteriormente, el *payload* debe tener una estructura determinada. Se denominan *Device Schemas* [49]. Hay que declararlos de antemano para hacer que el sistema de reglas funcione, haciendo que no se puedan tener eventos personalizados.

Acotación de estructura de dispositivos

Application, dispositivo y *gateway* son los tres tipos dispositivos, que en el mundo físico pueden ser exactamente iguales. Los dispositivos sólo pueden publicar eventos o comandos, haciendo que su funcionalidad se vea reducida. Esto genera que se limiten mucho la posibilidades de uso que tiene la plataforma, que dentro de todo es bastante completa.

Conclusiones

Creemos que proveen una plataforma bastante completa, pero el hecho que fuerzan al usuario a utilizar su estructura no estandarizada y no particularmente bien documentada la convierte en una mala opción para nuestro

proyecto, donde la facilidad de migrar de plataformas es algo deseable. Por otro lado, tienen buenos tutoriales para utilizarlo con dispositivos específicos como un Arduino con determinados sensores. Si el problema que se quiere resolver está dentro de lo que ellos tienen pensado entonces concluimos que es una buena alternativa.

Dentro de su estructura propia hacen un fuerte incapié en el lenguaje *node-RED* [50], donde se programa de forma visual, para conectar diferentes dispositivos con APIs en internet. Esto está bien integrado con el resto de los ofrecimientos en la nube de IBM. El hecho que hacen tanto hincapié en un lenguaje de programación visual nos da a entender que quizás buscan un público menos técnico con su producto.

Todo lo mencionado anteriormente se puede hacer fácilmente con AWS, ya que para dar permisos a diferentes dispositivo se utiliza la nomenclatura básica del resto de la plataforma, y subscribirse y publicar a topics siguen al pie de la letra las posibilidades que nos da MQTT, por ejemplo, varios niveles de topics, entre otros.

Microsoft Azure IoT

Microsoft Azure provee una plataforma IoT basada en los estándares de MQTT, haciendo que su manejo sea relativamente simple. Las caracterísitcas que la diferencian de otras no interfieren con flujos normales, y en muchos aspectos es bastante similar a la oferta de AWS. Es por este motivo que la consideramos para el desarrollo del prototipo, ya que eventualmente migrarlo a otra plataforma no sería costoso.

Características particulares

Como muchas otras plataformas, Azure IoT soporta MQTT y HTTP, pero además soporta AMQP, que tiene algunas características interesantes que no están presentes en MQTT pero su adopción no es tan global.

Sobre el soporte nativo a estos protocolos, tienen lo que ellos llaman *Microsoft Azure IoT Protocol Gateway* [51], un módulo que se instala en la plataforma del usuario para actuar como intermediario de protocolos específicos fuera de los nativos y el hub de la plataforma. Si bien esto no es de nuestro interés, nos pareció una característica muy interesante para otro tipos de proyectos, en los que se quiera tener centralizado la plataforma IoT pero se tenga una gran variedad de dispositivos diferentes.

Dentro del *broker* MQTT, expanden las características de los *topics* agregándole la posibilidad de pasar atributos en el *topic* mismo, similar a lo que se puede hacer con HTTP. Esto quiere decir, por ejemplo, publicando a

```
devices/{device_id}/messages/events/{property_bag}
```

con *property_bag* con el siguiente formato:

```
RFC 2396-encoded(<PropertyName1>)=
RFC 2396-encoded(<PropertyValue1>)&RFC 2396-encoded(<PropertyName2>)=
RFC 2396-encoded(<PropertyValue2>)...
```

uno podría comunicar atributos fuera del *payload* del mensaje mismo.

Encontramos también una similitud con una característica de AWS, *thing shadows*, llamados *device twins*, que básicamente son un documento JSON que guarda metadatos de cada dispositivo registrado en la nube de forma automática, al cual pueden acceder varios agentes y generar acciones. Más detalles sobre esta característica se encuentran en la sección AWS IoT .

Algo a destacar es que si bien la idea de los *thing shadows* y los *device twins* es la misma, y ambos funcionan sobre MQTT, la implementación es completamente distinta. Requieren reportar eventos de forma diferente en *topics* distintos, y en general si se tuviera que migrar de una plataforma a otra, habría que implementar la parte que usa los *shadows* (o *twins*) desde cero.

Como desventaja principal, vimos que no se permite más de una conexión MQTT por dispositivo.

Conclusiones

Es una plataforma sólida que bien podría haber sido usada en el desarrollo del prototipo. La interfaz gráfica es la más unificada que encontramos, todas las diferentes secciones de Azure son muy similares en cuanto a diseño y usabilidad y se comportan de forma similar. La única diferenciación que nos llevó a elegir de otra manera es que estamos más familiarizados con los otros servicios y ofrecimientos de AWS, por lo que las partes del desarrollo referentes a bases de datos, APIs, y funciones sin servidor, entre otras características, nos resultarían más sencillas.

Kaa

Kaa [6] es la única plataforma relevada que es open source y gratuita, y es la única que no tiene infraestructura en la nube como parte de sus ofrecimiento.

Kaa es simplemente el software en sí que puede ser descargado e instalado en servidores propios (ya sean servidores propios físicos como en la nube).

Características particulares

La plataforma tiene soporte nativo para ciertos dispositivos y lenguajes, para los cuales genera un SDK específico. Este SDK luego se encargará de toda la comunicación entre el dispositivo y la plataforma. Consideramos que esta es la caracterísitca principal que diferencia a Kaa del resto; la posibilidad de poder pensar la solución buscada sin tener que preocuparse por autenticaciones, protocolos y comunicación en general. Hasta ahora la comunicación entre dispositivos tiene que ser hecha siempre a mano.

Por otro lado, el hecho que no esté ligado a ninguna infraestructura particular permite que sea la plataforma con más fácil mobilidad de todas. Si uno quisiera mover el sistema entero de un lugar a otro, incluso de una infraestructura *on premises* a la nube, lo podría hacer sin mucho trabajo.

Al ser una plataforma de código abierto un usuario no está completamente atado a las decisiones que la empresa desarrolladora tome — siempre tiene la libertad de hacer un *fork* y cambiar las características deseadas. Además, un usuario tiene libertad de elegir su infraestructura, ya sea un proveedor de computación en la nube o él mismo administrar directamente sus servidores.

Desventajas

Si bien el concepto de Kaa es original y trae muchas ventajas al poder olvidarse de la comunicación (al ya estar resuelta), trae problemas de compatibilidad. Si uno tiene que trabajar con un dispositivo o un lenguaje fuera de los soportados, no tendrá acceso a un SDK y no podrá utilizar su mejor característica, el sistema automático de gestión de comunicación. Si quitamos esto, podríamos realizar un sistema con los módulos de comunicación hechos a mano, pero no sería diferente a desarrollar para cualquier otra plataforma.

Otro problema, que viene como contraparte a tener la libertad de elegir la infraestructura, es que no es posible funcionar de manera *serverless*: Kaa corre en un servidor que es necesario administrar y actualizar. Cuando sea necesario escalar, habrá que tomar consideraciones y tomarse mucho más trabajo al ampliar la solución para más dispositivos.

Conclusiones del relevamiento

En general, la principal conclusión que sacamos de este relevamiento es el mundo de plataformas IoT es muy nuevo y necesita madurar. Ejemplos claros de esto se pueden ver en el hecho que durante el desarrollo de este proyecto, varias plataformas vieron cambios de nombre o re-estructuraciones (IBM Bluemix IoT se convirtió en IBM Cloud Watson IoT [52], y los creadores de Kaa Project se reformaron en una nueva compañía, KaaIoT [53]). No es raro encontrar un tutorial de hace menos de un año que sea obsoleto, o tutoriales de funcionalidades fundamentales de las plataformas escritos hace pocas semanas, indicando el rápido ritmo de cambio en estas plataformas

Es común que al consultar artículos de relevamiento de plataformas con algunos años de antigüedad [54] [55] [56] [57] [58] [59] [60] [25], que poseen listados de decenas de plataformas, uno se encuentre que muchas de ellas ya no existen.

Saliendo de el análisis de los cambios repentinos, todas las plataformas relevadas tienen librerías específicas a su plataforma. Muchas veces esto hace que el desarrollo inicial sea más simple, pero puede complicar el desarrollo de funcionalidades complejas, al estar atado a la librería. También trae el problema que moverse de una plataforma a otra es imposible si todo el desarrollo fue basado en dichas librerías.

Saber qué hace una plataforma es difícil. Algunas ofrecen un *broker* MQTT con funcionalidades de persistencia de datos, otras analítica sobre gran cantidad de información, otras software particular para controlar ciertos dispositivos, etc. Pero todas están dentro de la misma categoría, plataforma IoT.

Todos estos puntos nos hacen concluir que el ambiente del software y plataformas de IoT está muy verde. El ofrecimiento es muy grande dentro del área de soluciones verticales (no se aplica a nuestro caso), y pocas soluciones de propósito general. Dentro de estas, no existen estándares que permitan la buena compatibilidad y comparación entre plataformas.

3.4. Elección de Herramientas

En esta sección exponemos las herramientas elegidas para el desarrollo del prototipo, motivos por los cuales fueron elegidas, y alternativas consideradas.

3.4.1. MQTT

Para el protocolo de comunicación con el servidor elegimos MQTT (*Message Queue Telemetry Transport*). El protocolo es estándar en el ámbito de IoT, siendo soportado por todas la plataformas analizadas en nuestro sondeo. Usar una tecnología estandarizada nos permite cambiar de plataforma sin tener que reescribir el código que maneja los dispositivos. Otros protocolos de mensajería como *AMQP* (*Advanced Message Queuing Protocol*) tienen sus puntos fuertes, pero no cuentan con un soporte tan amplio.

La principal alternativa que encontramos a MQTT presente en la mayoría de las plataformas es *HTTPS*. A pesar de no estar diseñada con IoT en mente, cuenta con amplio soporte y está bien adaptada a este caso de uso — dispositivos relativamente poderosos que actúan únicamente como sensores que envían datos. Decidimos en contra de *HTTPS* porque los paquetes son de mayor tamaño y presenta mayores dificultades a la hora de enviar información desde el sistema a los dispositivos, una característica que podría ser requerida a futuro.

3.4.2. Python

Tomamos la decisión de programar el código utilizando el lenguaje Python por ser de alto nivel apto para este tipo de proyectos y con el cual contamos con mucha experiencia. Python además tiene amplio soporte por parte de gran cantidad de plataformas IoT y posee muchas bibliotecas bien documentadas que resuelven todo tipo de problemas puntuales.

Bibliotecas

Las siguientes bibliotecas fueron elegidas para simplificar un poco el desarrollo del código de recolección de datos dentro de las ceibalitas.

La comunicación con las utilidades de Linux — para obtener los datos a recolectar —, la realizamos a través de la biblioteca `sh` [61]. Provee una interfaz más limpia y conveniente que `subprocess` [62], la biblioteca estándar de Python para este propósito.

En cuanto a la comunicación con el servidor, utilizamos Eclipse Paho (`paho-mqtt`) [63], una implementación abierta del cliente MQTT. Amazon provee su propia biblioteca de Python para conectarse al *broker* MQTT (`aws-iot-device-sdk-python`) [64], pero ésta simplemente utiliza Paho de

fondo. Decidimos en contra de usar la biblioteca de Amazon — a pesar de que podría resultar más conveniente — para que sea más sencillo conectar el programa con una plataforma de IoT distinta, uno de los requisitos del cliente. Eclipse Paho es estándar, y sirve para conectarse a cualquier plataforma de IoT que soporte MQTT.

Virtualenv

Para aislar nuestro entorno de Python del resto del sistema utilizamos Virtualenv [65], una herramienta para crear entornos aislados de Python. Crea un entorno con sus propios directorios de instalación, que no comparte bibliotecas con otros entornos de virtualenv ni tiene acceso a las bibliotecas instaladas globalmente. Esta herramienta nos da la libertad de instalar las bibliotecas que precisemos sin afectar las del sistema, así como actualizarlas en el futuro.

3.4.3. FPM

Uno de los requisitos era que la instalación en el dispositivo sea mediante un paquete de Debian (de extensión `.deb`). Un paquete de este tipo es en realidad un archivo comprimido con una estructura particular [66], pero generarlo a mano es una tarea complicada para la cual no contábamos con experiencia previa. Como la instalación en sí es sencilla — copiar los archivos a un directorio en particular y configurar Crontab — optamos por utilizar FPM [67].

FPM es una herramienta que permite generar paquetes de instalación de múltiples tipos en base a directorios u otros paquetes de instalación. Su principal ventaja es su simplicidad de uso, no moleste al usuario con detalles propios de cada plataforma.

3.4.4. Cron

Para repetir la recolección y el envío de datos con el intervalo de tiempo requerido utilizamos Cron [68]. Esta probada herramienta de Unix permite repetir tareas a intervalos regulares y está presente en todas las versiones de Ubuntu. La alternativa a esto era usar servicios, que en Ubuntu tienen la característica de funcionar distinto en diferentes versiones. En particular, el sistema de inicio de Ubuntu 14.04 es `init.d` mientras que el de Ubuntu 16.04 es `systemd`. Por estas razones y evitando problemas de configuración decidimos

utilizar Cron. Si precisáramos que los dispositivos estén siempre escuchando por mensajes desde el servidor, sería inevitable usar un servicio, teniendo que tener un código particular para cada versión del sistema operativo de las ceibalitas.

3.4.5. PostgreSQL y bases de datos relacionales

Una decisión importante a tomar siempre fue el cómo persistir los datos. Esta decisión impactaría en el rendimiento del sistema, en la facilidad de acceso a los datos, en la escalabilidad (tanto en cantidad de dispositivos como en tamaño de la base de datos), entre otros.

El conjunto de datos a guardar siempre fue estructurado y acotado, por lo tanto desde el primer día una base de datos relacional parecía la mejor opción para la persistencia. Ya habiendo decidido que ibamos a desarrollar el sistema en AWS, teníamos dos opciones principales, utilizar AWS RDS (explicado en detalle en la sección AWS RDS y DynamoDB) o una base de datos externa provista por Ceibal o propia. Para mantenernos dentro del ecosistema de AWS decidimos utilizar RDS, que posee integraciones mucho mejores a los otros servicios que cualquer otra opción.

En cuanto a qué base de datos relacionales, elegimos PostgreSQL principalemnte por dos razones. Primero, posee buenas opciones para almacenar datos no estructurados en formato *JSON*, dándonos la posibilidad de poder almacenar datos poco estructurados como las redes vistas por las ceibalitas, y nos da la posibilidad de agregar cualquier otro atributo de este estilo en el futuro si es que fuera necesario. Segundo, era la base de datos con la que teníamos más experiencia trabajándola junto con Python.

Esta decisión tiene una complicación, y es que si bien una base de datos relacional dentro de RDS puede escalar tanto en tamaño como en rendimiento, no existen formas automáticas de hacerlo y pueden no ser instantáneas. Otras opciones evaluadas (como DynamoDB) no tienen este problema, siendo posible establecer condiciones específicas para que la base de datos aumente o reduzca su capacidad de cómputo.

3.4.6. AWS Lambda y API Gateway

Desde el comienzo pensamos en hacer una solución *serverless* para todos los componentes involucrados, siendo Lambda la mejor solución dentro de las que

ofrece AWS para los requerimientos de procesamiento de los datos y eventual persistencia. Una vez escritas las funciones, están disponibles al instante en un ambiente de producción y listas para ser ejecutadas millones de veces por minuto, ya que cada invocación genera instantáneamente una instancia nueva de la función. No requieren un trabajo de monitorización de recursos y mantenimiento del servidor, y funcionan como una caja negra que ejecuta nuestro código, sin intervención nuestra en cuanto a administración y control de rendimiento. Detalles de su funcionamiento y arquitectura pueden encontrarse en la sección AWS Lambda.

La única alternativa que consideramos fue utilizar una cola de mensajes con los datos provenientes de las ceibalitas, y tener un servidor dedicado a leer de dicha cola, procesar los datos y guardarlos en la base de datos. Esto hubiera traído la complicación de tener que hacernos cargo del servidor y monitorizar su rendimiento, además de poder generar un cuello de botella en el sistema al no escalar automáticamente.

Por estas razones es que Lambda estaría satisfaciendo todas nuestras necesidades, falta entonces hacerle llegar los datos. Para esto elegimos API Gateway, la opción más simple para lo que necesitábamos y la que AWS siempre recomienda por defecto. API Gateway entonces pasó a funcionar como un proxy que expone una API RESTFul, y ejecuta funciones Lambda con el contenido del *payload* HTTPS.

Alternativas a esta combinación de tecnologías podría haber sido un servidor interno en AWS que exponga una API y se conecte con Lambda, pero tiene los mismos problemas que lo mencionado anteriormente, además de ser más complejo de administrar y configurar.

3.4.7. AWS IoT

Luego del amplio relevamiento de las actuales plataformas de IoT disponibles (explicado en detalle en la sección Relevamient de plataformas IoT) llegamos a la conclusión que AWS IoT era la que mejor se acercaba a nuestros requerimientos, por los siguientes motivos:

- Buenas opciones específicas de AWS pero muy buen sistema genérico. Como nuestro sistema no requería recursos fuera de lo que un *broker* MQTT provee, necesitábamos una plataforma que nos permitiera crear el sistema a nuestro modo, sin caer en una estructura o arquitectura

particular.

- Buena integración con los servicios de AWS. Al ser parte del ecosistema, AWS IoT se integra sin ningún conflicto a todos los servicios provistos por la nube de Amazon. En particular, Lambda y RDS, así como también cualquier otro servicio que pueda ser útil en el futuro (como servicios de análisis de datos).
- Buena documentación y comunidad. Al ser una de las principales plataformas en la nube, AWS (y por lo tanto AWS IoT) tiene una excelente documentación y comunidad que la respalda.
- Sistema de autenticación. Uno de los requerimientos más importantes en el sistema es la seguridad. Como AWS IoT posee un sistema de autenticación basado en certificados individuales y políticas de acceso, encontramos que podíamos adaptarlo perfectamente a nuestras necesidades.

3.4.8. Arquitectura *serverless*

Como el objetivo principal del proyecto fue siempre el análisis y relevamiento del ecosistema IoT en la actualidad, decidimos guiarnos por una arquitectura *serverless* para concentrarnos en la investigación y el rápido prototipado y no tanto en la elección de servidores y su mantenimiento y administración.

Trabajar con este tipo de arquitecturas trae sus ventajas y desventajas. Entre las ventajas encontramos su escalabilidad, facilidad de despliegue y rápido prototipado. Todas estas ventajas van de la mano con el planteo de lo que buscamos del proyecto. En cuanto a costos, cada proyecto es diferente y hay que analizar los requerimientos puntuales para ver qué resulta más conveniente. Aún así, algo seguro es que utilizando una arquitectura *serverless* uno paga por lo que usa y nada más.

En cuanto a desventajas, las arquitecturas *serverless* suelen estar muy ligadas a la plataforma utilizada, dificultando el proceso de migrar de una plataforma a otra. A su vez, hacen que desplegar un ambiente de desarrollo para probar las soluciones pueda ser más complicado y caro que en un servidor dedicado, en donde todo el sistema puede ser levantado en una computadora personal.

Una vez tomada la decisión de trabajar con AWS IoT, el sistema mismo nos lleva a una arquitectura de este tipo. Desarrollar una solución con servidores dedicados resulta más complejo, a tal punto que AWS IoT fomenta la

integración con funciones Lambda y otros servicios sin servidores, haciendo que sea más sencillo usar esta arquitectura.

3.5. Diseño e implementación

El sistema desarrollado consta de dos componentes principales, un cliente escrito en Python que corre en cada una de las ceibalitas del sistema y un servidor desplegado en AWS [69], que cuenta con el *broker* IoT y dos componente escritos en Python desplegados en Lambda para el registro de dispositivos y persistencia de datos.

3.5.1. En el servidor

Como dijimos anteriormente, pensamos en implementar una solución *serverless* y escalable, es decir, nosotros no administramos un servidor físico en ningún momento y automáticamente funciona sin problemas para pocos o muchos dispositivos, sin necesidad de intervención.

Registro de ceibalitas

Registrar las ceibalitas en el sistema para que empiecen a enviar datos no es un problema trivial. Como detallamos en la sección AWS IoT, AWS provee formas de generar certificados y adjuntarlos a un *thing* en la plataforma. De esta manera, un dispositivo con un certificado se autentica y puede enviar mensajes, mientras que uno sin certificado no puede hacerlo. Por otro lado, AWS no nos provee de una facilidad directa para hacer llegar esos certificados al hardware mismo (en este caso, las ceibalitas). Tampoco provee en su documentación recomendaciones acerca de cómo hacerlo.

Soluciones posibles

Nos planteamos varias soluciones posibles a este problema.

Lo más sencillo es tener un certificado único en la plataforma, generado de antemano, e incluirlo en el instalador. De esta forma, todas las ceibalitas envían mensajes usando este certificado, e implementamos una regla para que cuando llegue un mensaje desde una ceibalita nueva, sea agregada automáticamente al sistema. Este enfoque tiene varios problemas, el más importante de ellos siendo un único certificado maestro, es un punto único de falla: si se viera

comprometido (lo cual no es difícil, considerando que hay una copia en todos los dispositivos), un agente malicioso podría enviar datos erróneos y llegar a causar una negación de servicio. O, si simplemente perdiera su validez por algún motivo, todos los dispositivos del sistema dejarían de poder enviar mensajes.

Desde un punto de vista práctico, resulta deseable poder tener un certificado por dispositivo, poder darles permisos diferentes a cada uno, y poder desactivar algunos certificados cuando se desee sin tirar todo el sistema abajo (de hecho, poder hacer esto resultó útil cuando efectivamente desplegamos el prototipo). Finalmente, mirando a largo plazo es importante que los certificados puedan ser generados automáticamente, ya que estos tienen una validez limitada y deben ser renovados — idealmente sin intervención de administradores.

Una solución un poco más elaborada es generar un instalador diferente para cada dispositivo, cada uno con un certificado distinto. En nuestro caso esto no es posible, ya que no contamos control directo sobre el proceso de instalación y debemos proveer un único instalador para todas las ceibalitas. Por otro lado, aunque esta solución resuelve el problema del único punto de falla, los certificados siguen siendo estáticos. Bajo una solución de este estilo, si un certificado pierde su validez, ese dispositivo quedará inutilizable por el sistema hasta que un técnico manualmente le agregue un certificado nuevo.

Por lo tanto, es necesario implementar un sistema de generación automática de certificados. En esta solución, el instalador no provee al dispositivo de un certificado, sino que en la primer ejecución el dispositivo pide al servidor un certificado nuevo. El servidor genera este certificado, registra al dispositivo nuevo en el sistema (en caso de no estar ya registrado), y le envía al dispositivo el certificado generado. El dispositivo luego guarda este certificado en disco y lo utiliza para enviar mensajes. Si por algún motivo el certificado pierde su validez, el dispositivo repite el proceso y el servidor le genera un nuevo certificado (sin registrar un dispositivo nuevo, obviamente).

El problema con esta solución es que si el *endpoint* de generación de certificados es cerrado, los dispositivos precisan tener algún tipo de secreto para poder acceder a él, lo cual nos lleva de nuevo al mismo problema. Por otro lado, si el *endpoint* es abierto, el sistema tendría una vulnerabilidad similar al problema planteado anteriormente.

Para evitar este problema del huevo y la gallina, consideramos razonable usar un dato único que esté embebido en el hardware, el número serial. Si

el servidor cuenta con una lista de los números seriales de los dispositivos en los que se instala el programa, el dispositivo puede enviar su serial con la solicitud de certificado para que el servidor sepa si es realmente un dispositivo autorizado o no.

Esto no es inmune a problemas, ya que si se compromete una serial, cualquiera que la posea puede solicitar certificados. Aun así, estos certificados sólo le darán permiso a publicar mensajes en nombre de una ceibalita sola, lo cual no es tan malo.

Implementación

Logramos el registro de las ceibalitas a través de dos servicios, una API RESTFul provista por AWS ApiGateway [70], y código Python desplegado en AWS Lambda [39].

ApiGateway proporciona una API RESTFul que permite solamente los requests de tipo *POST* (todos los mensajes se envían sobre HTTPS). Como *body* del llamado requiere el número serial de la computadora. Cada ceibalita, en la primer ejecución del programa, realiza esta llamada para obtener un certificado y poder empezar a enviar datos. Al realizarse esta llamada es disparada una función Lambda, que recibe este dato y tiene permisos de acceso al *AWS IoT registry*.

La función Lambda toma el número serial y registra un *thing* en el *thing registry*. También crea un certificado *X.509* firmado por AWS. Este certificado será utilizado para las conexiones MQTT que se generen entre la ceibalita y el *broker*. Es único para la ceibalita, y es sencillo desactivarlo en caso de ser necesario. A este certificado le adjunta una *policy* [71], que registra los permisos de una ceibalita en el *broker*. En este caso, autorizamos a una ceibalita que porte ese certificado a publicar a dos *topics*, específicos para esa ceibalita:

`ceibalitas/<ID-DEL-CERTIFICADO>/logs`

y

`ceibalitas/<ID-DEL-CERTIFICADO>/errors`

Si intenta publicar en otro lado, la publicación queda denegada. Finalmente, el servidor le responde a la ceibalita con el certificado creado, y ésta lo guarda en disco.

Estas *policies* del certificado son chequeadas en el momento que un dispositivo intenta utilizar el *broker*. Esto es, conectarse, suscribirse a un *topic* o

realizar una publicación. Los documentos fueron escritos de tal modo que el sistema chequee el ID del certificado con el que el dispositivo abre un pedido para publicar, y el sistema valide contra el *topic* al que el dispositivo quiere publicar. Así, logramos un sistema de validación en el que cada ceibalita del sistema publica en un *topic* exclusivo para ella, y no tiene permiso para publicar en los de las demás.

Tener a todos los dispositivos publicando en diferentes *topics* resulta útil para poder segmentar la suscripción a mensajes por parte de otros servicios. En nuestro caso, dividimos los mensajes de error y los mensajes de datos, pero podrían hacerse otras jerarquías de *topics* más elaboradas, como mencionamos en la sección MQTT.

Destacamos que en este prototipo la API de registro no chequea los números de serie que recibe contra ninguna lista de números conocidos. Aunque esto es sencillo de implementar dentro de la función Lambda de registro, no lo hicimos para que sea más sencillo el despliegue, y porque juzgamos que el riesgo de seguridad no era relevante para un prototipo de este estilo.

Registro de datos

Realizamos el registro de datos de las ceibalitas a través del *broker* MQTT mencionado anteriormente, una función Lambda, y la base de datos.

Una vez recolectados los datos locales a enviar, la ceibalita establece una conexión MQTT con el *broker* utilizando el certificado que tiene almacenado. Si esta conexión es exitosa, publica en

```
ceibalitas/<ID-DEL-CERTIFICADO>/logs
```

El *broker* automáticamente valida si la policy que está adjuntada al certificado de conexión tiene el permiso de publicar en dicho *topic*.

Hay una regla en el motor de reglas que está suscrita a $ceibalitas/+/logs$, y se activa cada vez que llege un mensaje a esta colección de *topics*. Esta regla ejecuta una función Lambda en cada mensaje nuevo.

Una vez ejecutada, recibe como parámetros todos los datos del mensaje MQTT — el serial y el JSON con información recabada. Luego de un proceso de los datos y normalización de los mismos, abre una conexión con la base de datos desplegada en RDS e inserta una nueva línea en la tabla *measurements*.

Cada invocación de esta función es logueada en CloudWatch [72], donde son registrados hora, identificador de ceibalita, atributos recibidos y cualquier

error de ejecución en la función. Estos datos son útiles para hacer *debugging* de la función.

Registro de errores

El registro de errores tiene la funcionalidad de llevar un rastreo de cualquier clase de falla encontrada en el módulo de recolección de datos instalado en las ceibalitas. El objetivo es tener la mayor cantidad de información posible acerca del comportamiento del programa en las ceibalitas, sin precisar acceso físico a ellas. Obviamente no podemos tener información de errores si hay un error de conexión, pero cualquier otro error de ejecución debería poder ser enviado y almacenado en un registro en la nube.

Similar al registro de datos, para registrar errores una ceibalita publica en

`ceibalitas/<ID-DEL-CERTIFICADO>/errors`

un *topic* único para cada ceibalita.

Hay una regla suscrita a $ceibalitas/+/errors$, y en cada mensaje recibido genera una llamada a Dynamo. Dynamo automáticamente mapea las claves de los datos (enviados en formato JSON) como columnas en una tabla, en donde el serial number de la ceibalita actúa como *hash key* y un timestamp como *range key*.

Cabe aclarar que el envío de erroes va por separado del de datos; se traduce en un mensaje MQTT por fuera del mensaje principal.

Arquitectura general

En las figuras 3.1 y 3.2 mostramos dos diagramas de flujo para los tres casos de uso que fueron desarrollados (registro de ceibalitas, registro de datos y registro de errores).

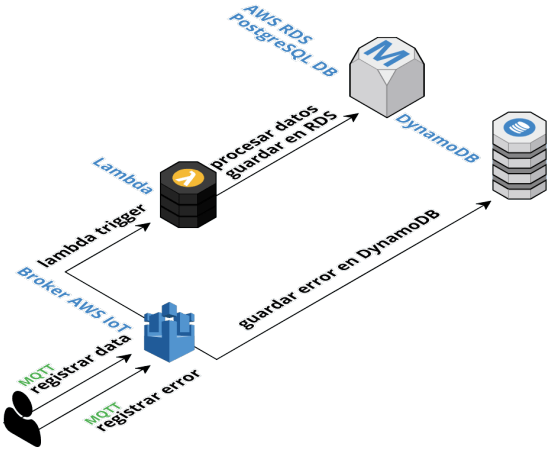
Estos diagramas muestran todos los pasos que se ejecutan cuando el usuario (en nuestro caso, el *daemon* corriendo en las ceibalitas) requiere hacer algo con el sistema.

En la figura 3.1 mostramos tanto el registro de datos como el registro de errores. Estos se desglosan en los siguientes pasos:

Registro de datos (flujo de la izquierda)

1. Ceibalita envía un mensaje MQTT al topic $ceibalitas/<ID-DEL-CERTIFICADO>/logs$, utilizando su certificado único, con los datos

Figura 3.1: Diagrama de envío de datos y errores



a registrar.

2. El *broker* MQTT recibe el mensaje y manda ejecutar una función Lambda, pasándole todo el contenido del *payload* como parámetros.
3. AWS levanta una nueva instancia de nuestra función Lambda y la manda ejecutar con todos los datos a guardar. Esta función los procesa, abre una conexión con la base de datos[1] y lo guarda en una nueva entrada.

Todo esto se da de forma asincrónica, ya que luego de recibir el mensaje, todos los procesos se ejecutan en el fondo.

Registro de errores (flujo de la derecha)

1. Ceibalita envía un mensaje MQTT al topic $ceibalitas/ < ID - DEL - CERTIFICADO > /errors$, utilizando su certificado único, con los datos a registrar.
2. El *broker* MQTT recibe el mensaje y ejecuta una de sus reglas predefinidas para guardar los datos en una tabla de DynamoDB.
3. Esta regla chequea que el formato del *payload* sea un JSON válido, con la forma de un diccionario *key: value*. Además, chequea que una de las

[1]Esta conexión puede ser completamente nueva o puede ya estar creada, dependiendo de la cantidad de ejecuciones recientes de la función. Explicado en detalle en la sercción AWS Lambda.

keys tenga el mismo nombre que la *HASH Key* y de la *RANGE Key* de la tabla (en nuestro caso, el número serial de la ceibalita y el timestamp del registro).
4. Si la validación es correcta, se agrega una nueva línea en la tabla con los datos.

La figura 3.2 por otro lado, muestra el flujo completo de registro de ceibalitas. Se puede desglosar en los siguientes pasos:

Registro de ceibalitas

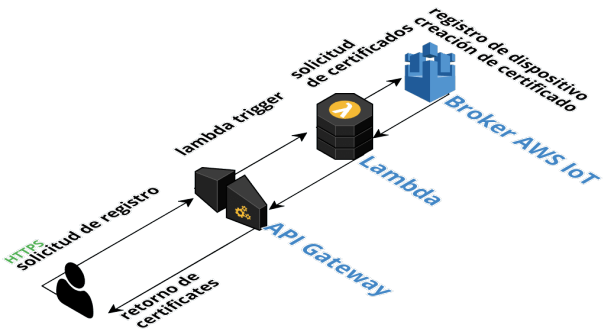
1. Ceibalita genera una *request* HTTPS POST a nuestra API corriendo sobre AWS API Gateway.
2. API Gateway está programada como un disparador de la función lambda de registro de ceibalitas, por lo tanto se ejecuta una nueva instancia de la función con los datos posteados en la API (número de serie de la ceibalita) como parámetros.
3. Lambda se ejecuta y utilizando la librería `boto3` de AWS para python, se comunica con el *Broker* MQTT para pedir un nuevo certificado.
4. El *broker* crea un certificado único firmado por AWS, le asigna la policy especificada por nosotros[2] y lo retorna.
5. Lambda recibe el certificado y se conecta con el *Broker* nuevamente para agregar la ceibalita en el registro.
6. Al finalizar, Lambda retorna el certificado que será suministrado a la ceibalita a través de API Gateway como respuesta a la request HTTPS.
7. La ceibalita guarda el certificado en su disco para poder usarlo en el futuro.

3.5.2. En los dispositivos

Ya explicado el proceso de desarrollo e implementación en el servidor, pasaremos a explicar los programas desarrollados para ser ejecutados en cada ceibalita.

[2]Explicado en detalle anteriormente en esta sección

Figura 3.2: Diagrama de registro de ceibalitas



Entorno virtual

Las dependencias de Python las manejamos a través de un entorno virtual utilizando `virtualenv`. Esta herramienta permite generar una instalación de Python con sus propias bibliotecas completamente aislada de la del sistema. Al permitir independencia casi total del Python instalado en el sistema, nos da la ventaja de tener pleno control del versionado de las bibliotecas y no tener que modificar nada en los dispositivos.

Decimos *casi total* porque hay bibliotecas que no se pueden poner en el entorno virtual, y Python tomará las del sistema. En particular la biblioteca SSL (que implementa el protocolo SSL), utilizada para la comunicación encriptada, depende de la instalación de Python del sistema y se comporta distinto en diferentes versiones de Ubuntu — que vienen con diferentes versiones de Python.

Instalación

Un requisito es que el programa se instale en los dispositivos con un paquete de Debian (de extensión .deb). Para esta tarea utilizamos la herramienta FPM, que permite crear paquetes de instalación de Debian (entre otros).

Para automatizar el proceso de generación del paquete de instalación crea-

mos un script — `build.sh` — que ejecuta los siguientes pasos:

- Generar un nuevo entorno virtual.
- Instalar en este entorno virtual todas las dependencias que requiere nuestro programa para funcionar en el dispositivo.
- Crear un paquete de Debian utilizando FPM, que incluya el directorio con el entorno virtual y el directorio con nuestros scripts.

El paquete generado también incluye los scripts `after-install.sh` y `after-remove.sh` que se ejecutan después de la instalación y después de la desinstalación del programa respectivamente. El script `after-install.sh` configura el crontab para que `mqtt_client.py` sea ejecutado cada 5 minutos, y genera las credenciales con `credentials.py`. Es conveniente realizar este paso durante la instalación ya que es necesario contar con conexión a la red para crear las credenciales — es necesario conectarse al servidor central —, y es seguro contar con conexión durante la instalación. El script `after-remove.sh` se encarga de borrar del crontab el comando que se ejecuta periódicamente.

La instalación en sí sencillamente copia las carpetas del entorno virtual y del código a un directorio en el dispositivo. Luego, Cron ejecutará cada 5 minutos `mqtt_client.py` usando el intérprete de Python del entorno virtual, que incluye a todas las dependencias.

3.5.3. Funcionamiento

El programa principal funciona de la siguiente manera:

1. Chequea si hay conexión a internet (en caso negativo, termina). [3]
2. En caso de que sí haya conexión, chequea si tiene las credenciales, y en caso de no tenerlas, repite el proceso de generar credenciales (igual al que se realiza luego de la instalación).
3. Recolecta los datos y genera el JSON que va a ser enviado. Los datos se recolectan de diferentes formas, la mayoría con comandos de Linux seguidos de un procesamiento con expresiones regulares.

La recolección de cualquiera de los datos puede fallar. Cada una de estas fallas genera una excepción, que es atrapada, genera una entrada el *log* local,

[3] La funcionalidad de almacenar paquetes hasta que se restaure la conexión quedó descartada para el prototipo, pero en una versión final sería importante tenerla.

y un mensaje de error que se envía a la base de datos remota. Luego de un error, ese campo se deja en blanco en el JSON, y el programa continúa con los demás datos. Aquí es donde el envío remoto de errores es útil: cuando de otra forma recibiríamos un campo en blanco y deberíamos ir hasta la máquina física para saber cuál es el problema, con el envío remoto de errores podemos tener información acerca del error, arreglarlo y desarrollar un parche sin en ningún momento tener acceso local a la máquina problemática.

Finalmente, el JSON generado es enviado al *topic* correspondiente y el programa termina.

3.6. Despliegue del prototipo

En la siguiente sección explicamos el proceso de despliegue del prototipo, empezando por una estimación de costos del período de prueba, problemas e inconvenientes encontrados a lo largo de este período y finalmente una pequeña guía para eventualmente mover el sistema a otra plataforma.

3.6.1. Estimación de costos

Para poner en producción el prototipo los clientes querían saber en cuántos dispositivos desplegarlo. Ellos esperaban poder desplegarlo en 500 dispositivos, enviando una muestra cada 5 minutos, y que el sistema permaneza en línea durante 1 mes. Aunque en teoría el *backend* de AWS debería escalar sin problemas, nosotros contábamos con un presupuesto reducido que limitaría la cantidad de dispositivos. En particular, AWS nos había provisto — gracias a *AWS Educate* [73] — de USD 40 de crédito para utilizar en su plataforma por encima del nivel gratuito. Intentamos estimar cuál sería el gasto máximo en el que podríamos incurrir, para saber si nuestro presupuesto alcanza para pagar el despliegue del prototipo.

A continuación analizamos los costos desglosados por cada uno de los servicios de AWS que usamos.

Base de datos

Los detalles del modelo de cobro se pueden encontrar en la sección AWS RDS y DynamoDB.

Gratuitamente, AWS provee una instancia de RDS de 4 GB. En base a las pruebas que hicimos, sabemos que cada línea de la base de datos pesa en promedio 1600 B. Como los clientes dispusieron que se envíe una muestra cada 5 minutos, estimamos que en el peor caso la ceibalita promedio probablemente no envíe datos más de 12 horas al día. Entonces, una ceibalita generaría 225 KB por día, y 500 máquinas generarían 110 MB por día. En 30 días, 500 máquinas generarían alrededor de 3300 MB en el peor caso, por debajo del límite de 4 GB gratuitos y con espacio adicional.

Aún así, identificamos que una gran parte de cada línea es ocupada por los datos de las redes vistas, porque es común que sean más de 10 y se almacenan directamente como un JSON dentro de la base de datos. Antes del despliegue, los clientes dijeron que esos datos revelaban mucha información personal y que deberían ser excluidos del experimento. Sin ellos, una línea de la base de datos baja a 800 B en promedio, y en consecuencia el tamaño total en el peor caso cae a 1650 MB, muy por debajo del máximo.

AWS IoT

Los detalles del modelo de cobro se pueden encontrar en la sección AWS IoT. Pasamos a desglosarlos para el uso en nuestro proyecto.

- El tiempo de conexión es despreciable, ya que no mantenemos ninguna conexión activa con los dispositovos.
- El contenido de cada uno de nuestros mensajes (sin los datos de redes vistas) pesa alrededor de 1600 B, por lo cual cada mensaje cuenta como dos. Además, la conexión inicial se realiza es un mensaje separado, así que en total cada 5 minutos una ceibalita consume 3 mensajes. Al cabo de 30 días, siguiendo la estimación pesimista de 12 horas de envío de datos en la ceibalita promedio, llegamos una cantidad de mensajes cerca de 6,5 millones; entre USD 30 y USD 35 de gasto.
- Los precios del uso del registro de dispositivos son por cantidad de modificaciones del registro y de las *Thing Shadows*, que nuestro sistema no realiza (salvo para crear los dispositivos en primer lugar).
- Como pensamos ejecutar al rededor de 2 millones de reglas (sin contar las ejecutadas por envío de errores), su costo es despreciable.

AWS Lambda

Los detalles del modelo de cobro se pueden encontrar en la sección AWS Lambda. El modelo de precios de Lambda considera tres cosas: cantidad de ejecuciones, la duración de esas ejecuciones y la cantidad de memoria RAM utilizada. La duración de cada una de nuestras ejecuciones — que consisten en simplemente procesar e insertar una línea en la base de datos — es despreciable dentro del nivel gratuito. En cuanto a cantidad de ejecuciones, el primer millón son gratis y luego el precio es USD 0,20 por millón. Como es una ejecución de Lambda por cada acción del motor de reglas, este total también está por debajo de USD 1, siempre y cuando se haya elegido la menor cantidad de memoria RAM disponible, que no trae problemas ya que lo que se necesita para el procesamiento de los datos es mínimo.

API Gateway

Los detalles del modelo de cobro se pueden encontrar en la sección AWS API Gateway. En nuestro sistema las únicas llamadas a API Gateway son para el registro de dispositivos, por lo que deberían ser 500. Esto está por debajo del millón de llamadas gratuitas que ofrece AWS.

Conclusiones de la estimación

Cualquier otro gasto no considerado en este análisis es porque es muy menos y está por debajo del nivel gratuito. En total, el único gasto importante en el cual incurriríamos según esta estimación es la cantidad de mensajes enviados, que en el peor caso está por debajo del gasto máximo. De esta forma, determinamos que desplegar el prototipo a 500 computadoras, enviando datos cada 5 minutos, durante un mes, costaría menos de nuestro crédito de USD 40.

3.6.2. Problemas e imprevistos encontrados

Como con cualquier otro sistema, al ponerlo a trabajar en el mundo real empiezan a aparecer problemas no previstos en el testeo. Pasamos a enumerar los más significativos, y la forma que tuvimos para resolverlos.

Puesta en producción del sistema

Al iniciar el experimento la primer noticia que obtuvimos es que los clientes accidentalmente instalaron el prototipo en alrededor de 2000 ceibalitas y no las 500 planificadas. Esto era preocupante, porque si bien había un sobrante en el cálculo del presupuesto, no sería suficiente para continuar el experimento durante un mes. Más allá de eso, no observamos ningún otro problema en el sistema, habiéndose cumplido el requisito de escalabilidad.

El sistema comenzó a funcionar con los siguientes registros:

- 7400 certificados registrados en IoT.
- 4222 certificados tienen asignado un *thing*.
- 2272 direcciones MAC diferentes en la base de datos.
- un flujo de mensajes de alrededor de 2000 cada 5 minutos.

Estos números reflejaron un problema menor en el registro de las ceibalitas, habiendo certificados sin asignar a ninguna ceibalita, y ceibalitas que tenían dos o más certificados. Encontramos que era un problema del algoritmo implementado para realizar el registro. No decidimos hacer nada al respecto ya que no afectaba el funcionamiento futuro del sistema habiendo podido proporcionar al menos un certificado válido para cada ceibalita.

Reducción de cantidad de ceibalitas

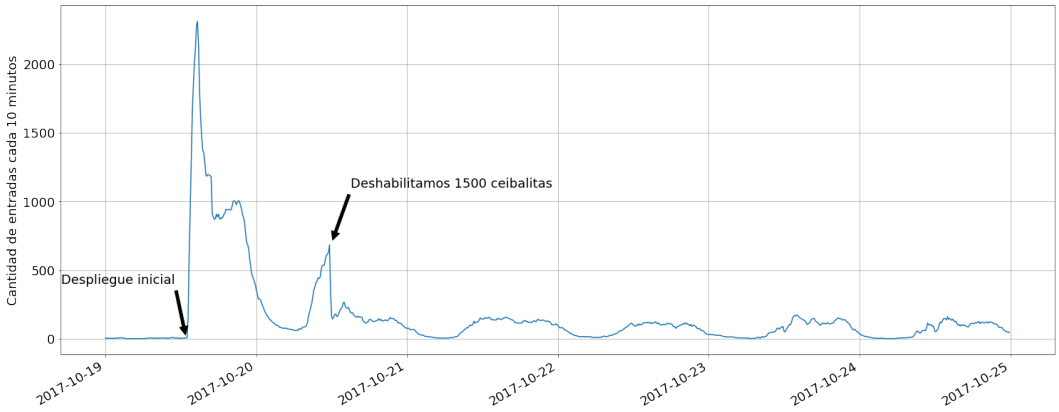
Nuestro sistema estaba preparado para esta eventual ocurrencia. La primer acción que tomamos al respecto fue de cerrar la API de registro, para que nuevas ceibalitas no se pudieran registrar. Así, cualquier otra ceibalita programada para recibir la actualización con nuestro código y aún no fue conectada a internet no pueda registrarse en nuestro sistema, impidiendo que el número se incrementara aún más.

Habiendo creado certificados únicos para cada ceibalita, y siendo esta la única forma de autenticación que tiene cada una, decidimos desactivar una cantidad de certificados hasta tener solamente 500 activos. Esto es una gran funcionalidad de la plataforma de AWS aprovechada por nuestro sistema, certificados pueden ser activados, desactivados o eliminados en cualquier momento de una forma relativamente sencilla.

Una cosa a remarcar es que para responder preguntas como ¿cuántos certificados hay registrados?, ¿cuántos certificados hay registrados que están asignados a un *thing*? y ¿cuántos *things* tienen asignados más de un certificado? la interfaz web no es suficiente y es necesario escribir un programa que haga estas consultas directamente sobre la API de AWS. Con un certificado desactivado

o eliminado, los dispositivos tendrán un error de autenticación SSL al intentar abrir una conexión MQTT con el *broker*.

Figura 3.3: Cantidad de entradas en la base de datos en los primeros días



Utilizando `boto3` [74] y *Python*, desarrollamos simples programas que se conectan al *broker*, obtiene todos los certificados con ceibalitas asignados, y uno por uno los desactiva hasta terminar con sólamente 500 activos. Así, pocas horas luego de la puesta en producción, el sistema tenía sólo 500 certificados activos, pudiendo continuar con su funcionamiento normal.

Reactivación de ceibalitas

Luego de tres semanas de la puesta en producción analizamos el avance de la recolección de datos. Este análisis arrojó un hecho particular, que la cantidad de datos era muy menor a la esperada en un primer lugar. Encontramos entonces las siguientes causas:

- Cantidad de ceibalitas activas. De los 500 certificados que quedaron activados en el sistema, sólo fueron utilizados 270. Esto quiere decir que hubo 230 ceibalitas que no volvieron a conectarse.
- Tiempo de conexión. Nuestra estimación (conservadora) asumía 12 horas de conexión diarias de cada ceibalita, cuando en la realidad fueron mucho menos. Esto se puede visualizar en la figuras 4.1 y 4.2 de la sección Análisis de datos.

Por esta razón decidimos aumentar la cantidad de certificados activos, y así aumentar la frecuencia de la recolección de datos.

La reactivación de 500 nuevas ceibalitas no fue sin errores, encontrando ningún aumento del tráfico hacia el sistema. Tras un análisis, encontramos un

error en el código desplegado en las ceibalitas, en el que al encontrar un fallo de autenticación el certificado sería borrado. Esto, sumado al hecho que la API de registro no pudo ser reactivada en la misma url (que se encuentra como constante en el código), hizo que se perdieran para siempre los certificados desactivados y que no fuera posible proveer de certificados nuevos a aquellas computadoras que lo perdieron.

Como trabajo a futuro planteamos una versión de la API de registro que se muestre sobre un único dominio actuando de proxy, pudiendo desplegar diferentes versiones de la API sin que se tenga que redesplegar el código en las ceibalitas. Además, planteamos el despliegue de una nueva versión del código con el arreglo (poco complejo) para el error mencionado anteriormente.

Esto podría haberse evitado habiendo implementado un sistema de actualizaciones desde el sistema mismo, sin ser necesario redesplegar el código en las ceibalitas. Más detalles de esto se encuentran en la sección Trabajo a futuro.

3.6.3. Migración a otra plataforma

Si bien el prototipo fue planteado como evaluación de la arquitectura propuesta, uno de los requisitos por parte de los clientes es que migrarlo a otra plataforma sea lo más sencillo posible. Es decir, salir de la plataforma de AWS y desplegarlo en otro lado, ya sea otra nube o una solución local. En esta sección presentamos los pasos a seguir para lograr este tipo de migración.

Recolección de datos en las ceibalitas

Este componente sería prácticamente igual. Los únicos cambios necesarios serían en la autenticación, que dependen de la nueva plataforma. Más allá de eso, siempre y cuando la nueva plataforma soporte MQTT, la recolección de datos no vería otros cambios.

Broker MQTT

El componente principal del prototipo es el *broker* MQTT provisto por AWS IoT. Cualquier otra forma de desplegar el sistema va a necesitar un *broker* para manejar las conexiones y recibir mensajes desde las ceibalitas.

Como analizamos en la sección Relevamiento de plataformas de IoT, existen muchas posibilidades para este componente de la arquitectura. Sea cual

sea la plataforma elegida, es fundamental que el *broker* acepte publicación a *topics* sin estructura, dejando cualquier clase de decisión a los desarolladores. Encontramos casos de plataformas que requieren una estructura específica durante el relevamiento, y si se eligiera un ejemplo que lleva esta forma de publicación sería necesario modificar en gran medida tanto el módulo de las ceibalitas como los programas que procesan y guardan los datos.

Otra aspecto a tener en cuenta es la configuración necesaria para conectar el *broker* con el resto de los módulos, ya que AWS lo hace de forma automática a través de su sistema de permisos internos.

Registro de ceibalitas

Como detallamos en la sección Registro de ceibalitas, la autenticación en el prototipo es realizada a través de la creación de certificados *X.509* firmados por AWS. Estos son creados por una función Lambda que es activada a través de una API RESTFul corriendo sobre AWS API Gateway. Están conectados al servicio de IoT a través de los permisos que les otorgamos dentro del ecosistema de AWS.

Si se quisiera usar un sistema similar se necesitaría una autoridad que firmara los certificados, aceptada por el *broker*, y una forma de crearlos y enviarlos a través de un pedido HTTPS. Hacer este sistema manualmente es complejo, y no encontramos ninguna plataforma que lo implemente de una forma que lo hace AWS. Se podría cambiar por el método de autenticación preferido por la plataforma elegida, por ejemplo, usuario y contraseña. De todas formas, probablemente resulte necesario volver a implementar todo este sistema nuevamente en la nueva plataforma.

La forma de autenticación también afectaría los permisos de cada ceibalita, ya que actualmente cada certificado tiene asociado los servicios de publicación y subscripción a los que puede acceder. Esto no es obligatorio ya que se podría habilitar un registro de datos y errores general para todos los dispositivos, pero sería más difícil de aislar problemas en un dispositivo en particular, así como dar permisos diferentes a diferentes dispositivos.

Motor de reglas

La mayoría de las plataformas relevadas poseen un motor de reglas capaz de ser activado en diferentes eventos, como lo son la llegada de mensajes, apertura

de una conexión, y el pasaje de ciertos períodos de tiempo, etc. Actualmente son utilizadas dos reglas, una para procesar datos enviados y otra para procesar errores enviados, ambas disparadas cuando llegan mensajes a los *topics* de registro de datos y registro de errores respectivamente. No utilizamos ninguna funcionalidad específica de AWS para lograr esto, excepto su integración con Lambda en el caso de los datos, y con DynamoDB en el caso de los errores.

En caso de una migración sería necesario reemplazar este sistema con la versión de la nueva plataforma, pero la lógica de guardado de datos y errores sería la misma.

Base de datos

La base de datos es lo más simple de migrar, ya que sólo se requieren las variables para abrir una conexión con ella en el código de procesamiento de datos. Se puede optar por una base de datos diferente dentro de AWS, alguna ofrecida por otra plataforma en la nube, o una base de datos servida localmente. El *schema* de la base de datos y la lógica de procesamiento de datos serían mantenidos sin cambios.

El problema se complejiza cuando consideramos la base de datos de errores, ya que se encuentra desplegada en DynamoDB, propietaria de Amazon. Se podría cambiar en cualquier momento por una base de datos SQL y tratarla como tratamos la base de datos principal, u optar por otras opciones NoSQL. Cualquiera sea la opción elegida, se tendría que crear un procedimiento para procesar y guardar los datos manualmente, ya que ahora esto se hace de forma automática utilizando la conexión directa entre el *broker* MQTT de AWS IoT y DynamoDB.

Procesos de persistencia, procesamiento y normalización de datos

La utilización de Lambda hace que el despliegue de los procesos de persistencia, procesamiento y normalización de datos sean muy fáciles de aislar, teniendo sólamente código para lograr lo que se necesita sin ninguna clase de configuración — como mencionamos en la sección AWS Lambda, solamente es necesario escribir el código que se desea ejecutar. Aún así, si bien el código es reutilizable casi en su totalidad, para replicar estos procesos sería necesario generar la configuración y conectar el motor de reglas de la nueva plataforma.

Este código puede ser desplegado tanto en otros servicios de ejecución

de código en la nube (como puede ser Google Cloud Functions[75] o Azure functions[76]), como en un servidor dedicado con Apache OpenWhisk[77] o alguna solución propia.

Conclusiones generales

Como las plataformas IoT siempre tienen uno o más componentes propietarios, una migración siempre terminará requiriendo volver a realizar partes del trabajo. Durante el desarrollo intentamos usar la menor cantidad de servicios exclusivos de AWS para no estar ligados a esta plataforma, pero lograrlo en su totalidad (utilizando a la vez las virtudes propias de la plataforma) es imposible.

Otro aspecto a tener en cuenta es la escalabilidad luego de la migración. Este prototipo está fuertemente basado en los principios de auto escalabilidad de AWS, y para mantener esta cualidad sería necesario verificar que la nueva plataforma provee garantías similares, o hacer un análisis de requsitos computacionales y poner esos recursos a disposición de la nueva solución.

Capítulo 4

Resultados

En este capítulo presentamos los resultados obtenidos al cabo de dos meses de funcionamiento del prototipo. Los resultados incluyen tanto las lecciones aprendidas acerca de infraestructura y sistemas de IoT en general así como los datos generados por el prototipo en sí.

Por el lado del sistema, analizamos la infraestructura generada, concentrándonos en las decisiones tomadas y su resultado, llegando a un resumen de consideraciones a tomar al momento de iniciar un proyecto de este tipo. Por el lado de los datos, describimos qué datos fueron recolectados exactamente, discutimos potenciales usos y riesgos, y presentamos un pequeño análisis sobre estos datos a modo de ejemplo.

4.1. Proyecto

En esta sección exponemos los resultados del proyecto, centrándonos en el diseño de la infraestructura, uso de la plataforma IoT elegida y las consideraciones tomadas a lo largo del desarrollo del prototipo.

4.1.1. Consideraciones para el desarrollo de un sistema de IoT

Luego del análisis, relevamiento, desarrollo y puesta en producción de un proyecto de IoT sobre una plataforma en la nube, planteamos varias consideraciones a tener en cuenta si uno desea embarcarse en un proyecto de este tipo. Considerando que el ecosistema de IoT está todavía muy inmaduro, creemos que esta sección es el uno de los resultados más importantes del proyecto.

Definitivamente no estamos planteando una guía para el desarrollo ni un manual de pasos probados para seguir a la hora de empezar un proyecto de IoT. Lo que sí planteamos es una serie de factores que consideramos importante tener en cuenta antes y durante la evolución de un proyecto de este tipo.

Caso de uso y requisitos

En primer lugar, dependiendo del caso de uso que se quiera trabajar, es posible encontrar plataformas que ya incluyan una solución específica ya desarrollada (como las hay para problemas de "ciudad inteligente" y sensores en el agro, por ejemplo), lo cual puede ahorrar mucho trabajo en el corto plazo (desarrollo) y en el largo plazo (mantenimiento).

Algunas empresas que venden tanto el sistema de software (e incluso hasta el hardware) para una serie limitada de soluciones (sensores de intensidad de luz para ciudades, sensores que indican qué tan lleno está un contenedor de basura, sensores de humedad para el campo). Un ejemplo de una plataforma orientada a soluciones más específicas es Carriots[78].

Por otro lado, otras empresas venden plataformas de propósito más general, como lo son AWS IoT, Microsoft Azure, y Google Cloud IoT. Desarrollar una solución en una plataforma de propósito general puede resultar más costoso comparado con utilizar una que ataque ese vertical en particular, pero a la vez permite mayor flexibilidad y personalización.

Creemos que si el caso de uso requerido es ofrecido por alguna empresa, probalemente sea más rápido y sencillo (pero no necesariamente más barato) contratar su servicios, ya que el desarrollo de un sistema de cero es una tarea compleja y requiere un equipo de desarrollo dedicado para su implementación y mantenimiento. Esto es, por supuesto, si otros requisitos no impiden el uso de un sistema externo, presuntamente propietario.

Si no fuera el caso, el desarrollo de un sistema de cero puede ser realizado en cualquier plataforma que ofrezca un servicio genérico, para no estar ligado a estructuras o procesos particulares.

Presupuesto, recursos humanos e infraestructura preexistente

Obviamente, una vez definido el caso de uso, es importante tener en consideración el presupuesto con el que se cuente para la implementación del sistema, y los recursos humanos y de infraestructura con los que ya cuenta la

organización.

En cuanto a los costos, la mayor diferencia estará en la infraestructura que se use. Aquí el análisis de costos se debe realizar como cualquier otro emprendimiento de software. Poseer los servidores *on premises* hace que el costo de tenerlos corriendo sea menor, pero hay que tener en cuenta que el mantenimiento va a ser realizado por parte del equipo, y no se cuenta con el soporte y respaldo una empresa que ofrece servicios en la nube. Si se trata de una organización existente que ya administra directamente sus propios servidores para otras cosas (como es el caso de Ceibal), es razonable pensar en una solución de IoT que corra en sus propios servidores, junto a sus otros servicios. Por otro lado, si es una organización nueva o que no cuenta con su propia división de servidores físicos, una solución *on premises* viene con un costo agregado mucho mayor.

Entonces, si se trata de una organización que ya tiene una infraestructura existente corriendo en servidores locales, el equipo de administración de sistemas y la cultura de mantener servidores, es razonable la instalación de un sistema local de una plataforma IoT como Kaa o FIWARE IoT. Por otro lado, si no se cuenta con infraestructura y se piensa adquirir una, hay que tener en cuenta los costos de armar un equipo de mantenimiento contra utilizar un servicio IaaS (*Infrastructure as a Service*, infraestructura como servicio) de alguna de las plataformas en la nube.

Es importante tener en cuenta que plataformas las IoT en la nube normalmente facturan solamente por los recursos utilizados. Esto puede generar que en algunos casos con mucha variabilidad usar servicios en la nube resulte una solución más económica que tener servidores potentes corriendo en todo momento, pero con alto nivel de ociosidad.

Respecto a los recursos humanos, el aspecto relevante es si la organización ya cuenta con un equipo de desarrollo armado para el caso, familiarizado con ciertas tecnologías. En este caso, es importante que la elección de plataforma tome en consideración las tecnologías y la forma de trabajo a la que los desarrolladores de la organización están acostumbrados. Un equipo de desarrollo no familiarizado los procesos y tecnologías de la plataforma va requerir más tiempo y recursos al comienzo del desarrollo para adquirir los conocimientos necesarios. Cualquier ventaja que puedan ser gandas utilizando ciertas tecnologías pueden ser truncadas por la experiencia anterior del equipo de desarrollo utilizando otras.

Escalabilidad de la solución

La escalabilidad siempre será un problema para la infraestructura; sin importar qué se está haciendo, no es lo mismo manejar cien dispositivos que manejar un billón. Este problema puede resultar ser complejo o sencillo de resolver, según la situación.

Si se tiene una solución con un conjunto limitado de dispositivos que actúan de una forma predecible (tienen una cantidad de interacciones con el servidor predeterminada, por ejemplo una red de sensores), es posible calcular con gran precisión la capacidad de cómputo necesaria para el sistema. En este tipo de casos la escalabilidad del sistema no será un problema difícil de resolver — puede resultar costoso si es un sistema muy grande, pero es predecible. Probablemente sea posible conseguir costos y complejidad bajos utilizando servidores dedicados.

Por otra parte, en un caso más impredecible es muy difícil calcular de forma exacta cuántos recursos se necesitan, teniendo siempre que hacer estimaciones pesimistas para que el sistema no falle. Esto resultará en desperdicio de recursos o fallas en el sistema. Por ejemplo, si la cantidad de dispositivos no es conocida de antemano, o el uso de ellos no es fácilmente predecible (por ejemplo, dispositivos actores controlados directamente por usuarios u otros eventos), será difícil o incluso imposible estimar con exactitud el uso esperado durante un mes. En un escenario de este tipo, utilizar una solución *serverless* similar a la de nuestro prototipo, en donde la escalabilidad es solucionada por la plataforma y no por el equipo de desarrollo.

Es gratificante para los desarrolladores saber que sin importar la cantidad de dispositivos, la plataforma garantiza que va a tener recursos suficientes para procesar los datos entrantes. Ejemplos de este servicio los ofrece AWS Lambda (auto escalable[79]), AWS EC2 Auto Scaling Group (despliega nuevos servidores una vez pasado determinado umbral de consumisión de recursos[80]) y Microsoft Azure Functions[76] (similares a Lambda).

Interacción con otros sistemas

Los sistemas suelen estar conectados con varios componentes, muchas veces desarrollados por equipos diferentes. Un sistema de IoT no existe aislado, sino que existe conectado con bases de datos, sistemas de monitorización de datos, visualización de datos y procesamiento de datos. Para todo esto será necesario

construir integraciones. Por tanto, al encarar el desarrollo de un sistema de IoT, como en cualquier otro sistema de software con este nivel de complejidad, es necesario tomar las integraciones en cuenta desde un principio.

La plataforma elegida será la mayor decisión a tomar en este aspecto, ya que las integraciones tendrán que ser hechas desde ahí. Recomendamos en este aspecto elegir la plataforma que mejor compatibilidad tenga con el resto de los sistemas de la organización, si es que ya existen. En el caso de las plataformas en la nube, la respuesta es sencilla: suelen tener una excelente interacción con el resto de los servicios que se ofrecen dentro de la misma nube.

Por ejemplo, desarrollar el prototipo en AWS permitió que muchísima configuración y funcionalidades sean dadas *out of the box* (conexión con Lambda, guardar directamente mensajes MQTT en DynamoDB, etc.). Por otro lado, si se tratara de una organización que usa servicios de Azure para otras tareas, no sería una buena decisión usar AWS para la parte de IoT, ya que la plataforma no está pensada para esto. Las integraciones en ese caso serán muy costosas y problemáticas.

Por otro lado, de no utilizar plataformas en la nube y seguir el camino de servicios locales, la investigación de los medios disponibles ofrecido es un poco más compleja. Por ejemplo, Kaa tiene buenas integraciones con Hadoop, Cassandra y MongoDB[81], entre otros. Otras plataformas tienen integraciones con otros servicios. Una organización debe tener muy en cuenta las integraciones que deberá soportar a la hora de elegir la plataforma.

Hardware de los dispositivos

Como hemos mencionado en secciones anteriores, la mayoría de las plataformas poseen SDKs para diferentes lenguajes y arquitecturas. La elección de la plataforma a utilizar limita el tipo de dispositivos compatibles, por lo tanto se debe tener bien claro con qué hardware se va a trabajar para asegurar que la compatibilidad no termine siendo un problema grave. En general, algunas plataformas ponen más énfasis en soportar algunos chips determinados que otras, en las que se espera más trabajo de parte de los desarrolladores para lograr la comunicación con el sistema.

Las plataformas de uso genérico generalmente ofrecen herramientas en la forma de librerías para varios lenguajes de programación, que abstraen muchas configuraciones y procedimientos para hacer el desarrollo más fácil. Si estas

librerías no proveen lo necesario, o es necesario usar un chip particular, no proveen más ayuda que las interfaces genéricas de MQTT o equivalente. AWS es un ejemplo de este tipo de plataformas.

Otras plataformas proveen SDK desarrollados para determinados dispositivos o lenguajes. Si justo proveen un SDK que funciona en el hardware que se vaya a utilizar (por ejemplo, el proyecto utiliza Arduino y la plataforma provee un SDK para Arduino), el proceso para los desarrolladores será mucho más fácil. Si el hardware no está soportado, se está de nuevo en el caso anterior. Kaa es un ejemplo de este tipo de plataformas.

Cualquiera de los dos tipos es válido dependiendo del tipo de proyecto que se quiera desarrollar, y recomendamos considerar si el soporte de hardware es valorado como una ventaja.

Soporte técnico de la plataforma

Con soporte técnico nos referimos al soporte disponible por parte de la organización detrás de la plataforma y la comunidad de sus usuarios.

En las plataformas en la nube el soporte es más importante que en las plataformas servidas localmente, porque uno depende completamente de ellos para solucionar cualquier problema que ocurra en los servidores de la plataforma misma. Dependiendo del plan que se contrate, el soporte directo de hablar directamente con un humano capacitado técnicamente puede ser desde inexistente, hasta garantizado en cortos períodos de tiempo.

Por otro lado, las plataformas que no son en la nube también cuentan con soporte, pero es de una naturaleza un poco diferente, ya que no le dan soporte a toda la infraestructura; simplemente a la plataforma. En particular, en las plataformas abiertas (como Kaa) el soporte es el principal generador de ganancias en el modelo de negocios de la compañía desarrolladora.

Otro tipo de soporte muy importante para el desarrollo es el de la comunidad, el grupo de desarrolladores que utilizan la plataforma, colaboran para hacerla mejor y muchas veces pueden forman parte del equipo de desarrollo de las mismas. Esto depende mucho de la popularidad de las plataformas, generalmente encontrando comunidades más grandes en las plataformas más populares. Por otro lado, aquí las plataformas abiertas pueden tener una ventaja sobre las propietarias, ya que si bien pueden poseer comunidades más pequeñas, suelen mucho más activas.

Privacidad de datos

Dependiendo del ámbito en el que se desarrolle el sistema, va a ser necesario tener en cuenta el ámbito legal, particularmente si se está desarrollando un producto afectado por leyes que impiden que datos salgan del territorio del país, ya sea para almacenamiento o para procesamiento.

Si es así, partes o la totalidad de los servidores del sistema deben estar localizados físicamente en Uruguay. En el caso que fueran solamente datos, podría tenerse la base de datos en servidores propios o dentro de data centers en Uruguay (como puede ser el caso de mi nube[82] de Antel), y el resto del sistema en plataformas en a nube o ubicadas en el exterior. Si la totalidad del sistema debe estar dentro del país, muchas de las opciones van a quedar descartadas, y entre ellas, todas las plataformas propietarias en la nube que no tengan datacenters en Uruguay. La única opción posible para este escenario evidentemente será una plataforma que pueda ser desplegada en servidores locales.

4.1.2. Conclusiones generales del proyecto

La decisión de qué plataformas y tecnologías usar resulta crucial para el futuro de un proyecto. Esto sucede porque desarrollar un sistema complejo en una plataforma específica implica que la migración a otra sea muy difícil. Por lo tanto, una decisión incorrecta puede tener consecuencias graves a la larga Esto sumado al hecho de que todo el ecosistema es muy inmaduro, hace que los riesgos de elegir mal sean muy altos, y las consecuencias caras de pagar. Luego del relevamiento del estado actual de la las opciones disponibles para desarrollar un sistema de IoT y luego de haber desarrollado uno nosotros mismos, la conclusión que podemos sacar es que probablemente lo más seguro que se puede hacer sea utilizar lo más popular.

La inestabilidad que se encuentra en el ecosistema hace que ir por una solución muy específica de una empresa no muy grande pueda llevar al fracaso total, ya que como vimos en el relevamiento, el panorama está sembrado de plataformas aparentemente muertas que hace menos de tres años parecían prometedoras.

Respecto a nuestro proyecto en particular, si bien el análisis, relevamiento y estudio de las actuales plataformas IoT en la nube disponibles en el mercado fue extenso y tardó mucho tiempo, el desarrollo en sí fue algo más directo y

corto. Una vez seleccionada la plataforma, generar familiaridad con ella fue relativamente sencillo, al igual que integrarla con el resto de los sistemas. No nos encontramos con ningún obstáculo particularmente grave, y la puesta en producción ocurrió sin mayores inconvenientes.

4.2. Prototipo

En esta sección presentamos los resultados obtenidos del prototipo y de la plataforma, así como un pequeño análisis de los datos recolectados a modo de ejemplo.

4.2.1. Uso de la plataforma

Obviamente nuestra estimación pesimista de peor caso no se llegó a cumplir, y tampoco estuvimos cerca de alcanzarla. El experimento duró más de dos meses, y nunca nos llegamos a acercarnos a los límites que calculamos (luego de solucionar los inconvenientes de los primeros días).

4.2.2. Datos recolectados

Expondremos aquí estadísticas e información de los datos obtenidos, así como un pequeño análisis realizado sobre estos.

El objetivo principal de este proyecto fue siempre un relevamiento, análisis e implementación de un sistema de IoT basado en las plataformas existentes en el momento. El caso y planteo de los clientes fue propuesto como una forma de desarollar un prototipo y desplegarlo en un escenario real, para convertir nuestro análisis en algo que resulte útil para el organismo. En este sentido, lo que los clientes necesitaban era un análisis al problema de recolección de datos para su infraestructura, una investigación del ecosistema actual que pudiera darles una mejor idea de herramientas y plataformas modernas. Esto les sería muy útil luego para compararlo con las herramientas que ya poseen, y poder diseñar proyectos a futuro de una mejor manera.

Como detallamos en la sección Requisitos, decidimos en conjunto una serie de métricas que sería útil recolectar en cada dispositivo (ceibalita). A continuación mostramos los datos que recolectamos en la implementación final del prototipo. Es decir, cada entrada de la base de datos tiene todos estos datos.

Debemos aclarar que esta lista fue elaborada teniendo en cuenta tanto el interés del cliente por recolectar el dato como por la dificultad de obtener dicho dato desde el dispositivo.

Tabla 4.1: Datos retornados en cada entrada

Campo	Descripción
`mac_addr`	La dirección MAC de la computadora.
`serial_number`	El número serial de la computadora.
`ip_addr`	La dirección IP pública de la computadora.
`timestamp`	Fecha y hora de la entrada.
`ap_mac_addr`	La dirección MAC del *access point* al que está conectada la computadora.
`frequency`	Frecuencia de la red a la que la computadora está conectada.
`rssi`	Indicador de fuerza de la señal recibida.

`tx_packets_quantity`
`tx_packets_overruns`
`tx_packets_carrier`
`tx_packets_errors`
`tx_packets_dropped`
`tx_excessive_retries`
	Información acerca de paquetes transmitidos por la interfaz.

`rx_packets_quantity`
`rx_packets_overruns`
`rx_packets_frame`
`rx_packets_errors`
`rx_packets_dropped`
`rx_bytes`
	Información acerca de paquetes recibidos por la interfaz.

`charging`	Indica si la computadora está enchufada a la corriente.
`battery_temp`	La temperatura de la batería
`battery_power`	El nivel de carga de la batería.

uptime	La cantidad de tiempo que la computadora lleva encendida.
boot_time	Tiempo que demoró la computadora en arrancar.
load_avg_5_min	La carga promedio del CPU de la computadora en los últimos 5 minutos.
total_memory_kb	El total de RAM de la computadora.
free_memory_kb	Cantidad de RAM libre.
total_swap_memory_kb	Tamaño total de la memoria *swap*.
free_swap_memory_kb	Cantidad de memoria *swap* libre.
cached_memory_kb	El tamaño del cache de paginado.
buffers_memory_kb	Tamaño de los *buffers* de I/O.
root_dir_total_disk_space_kb	Espacio total de disco en el directorio raíz.
root_dir_free_disk_space_kb	Espacio libre de disco en el directorio raíz.
home_dir_total_disk_space_kb	Espacio total de disco en el directorio /home.
home_dir_free_disk_space_kb	Espacio libre de disco en el directorio /home.

La infraestructura desarrollada puede, sin modificación alguna, recolectar todo tipo de datos desde cualquier dispositivo que posea una conexión a internet. En este sentido, agregar más datos en el futuro, tanto para este escenario como para un nuevo problema con datos completamente diferentes, implicaría no más que modificar el código que corre en cada dispositivo para que mande la serie de datos deseada, y las columnas de la base de datos.

Además de la facilidad mencionada anteriormente, la infraestructura también es capaz de trabajar con problemas con una cantidad reducida de dispositivos (cientos, como nuestro escenario), o una cantidad muy grande (millones). Sobre esto, la infraestructura tiene también la ventaja de utilizar sólo los recursos necesarios para la cantidad de dispositivos demandada.

Privacidad de datos

Como se puede ver en la lista de datos recolectados, la dirección MAC y el número de serie de las ceibalitas son recolectados en cada entrada (y son usados como identificador en la base de datos). Si bien estos datos de por sí no implican un riesgo al anonimato de los dueños de dichas ceibalitas, es posible mapear estos datos a personas cruzándolos con una tabla que diga el usuario al que pertenece cada ceibalita según su serial o dirección MAC. Ya que sin esta hipotética tabla no es posible mapear datos directamente a personas, consideramos que, por lo menos para una fase de prototipo, la privacidad de los usuarios está asegurada.

Es importante considerar que con este sistema es posible recolectar cualquer dato de la computadora que se pueda obtener teniendo acceso *root*. Esto incluye, pero no está limitado a, todos los datos del usuario, las aplicaciones que usa, los sitios web que visita, y todas las teclas que oprimió. Nosotros obviamente no recolectamos estos datos, pero esta capacidad está siempre presente en este tipo de sistemas de monitorización, y es responsabilidad de los desarolladores no recolectar datos que comprometan la privacidad de los usuarios. Consideramos que en este sentido respetamos la privacidad de los usuarios, incluso descartando datos como los nombres de las redes inalámbricas vistas, que potencialmente podrían comprometer el anonimato de los usuarios.

Análisis de datos

Si bien el análisis de los datos no era parte principal de este proyecto, mostramos a continuación algunos resultados interesantes que obtuvimos. Para un buen análisis de los datos se require conocimientos en el área, y quizás datos de períodos más largos de tiempo. Esta sección es un ejemplo de las posibilidades que nos brinda el resultado el prototipo.

Aunque estos análisis fueron realizados después del hecho, perfectamente se podrían realizar en vivo, a medida de que los datos van siendo recolectados, ya que apenas son enviados los datos están disponibles para el análisis.

Estos análisis se hicieron sobre las 651.655 entradas en la base de datos recolectadas a lo largo del prototipo, y a modo de ejemplo los realizamos sobre los datos de batería.

Temperatura

En primer lugar identificamos que 26,8ºC es el valor más común de temperatura de la batería, pero esto sucede porque muchas ceibalitas sólo reportan este valor. Los clientes confirmaron que esto se trata de un problema del hardware, así que toda computadora que reportara este valor fue descartada del análisis. Un 63.8 % de las computadoras reportan este problema.

Algo que nos pareció interesante analizar es la relación entre los cambios de temperatura promedio de las computadoras y los cambios de la temperatura en Montevideo. Es posible que la temperatura aumente con la temperatura del país a medida que avanza el año.

Analizamos este fenómeno comparando el promedio de temperatura diaro en Montevideo para los días en los que el prototipo funcionó, contra el promedio de temperatura de las baterías de las ceibalitas. El coeficiente Pearson de correlación entre los dos sets de datos es de 0.37, con $p < 0.01$. Es decir, si bien puede parecer que los resultados están relacionados, no podemos decirlo con certeza. Necesitaríamos más datos a lo largo de más tiempo para poder concluir si hay o no correlación.

Uso de los dispositivos

Otro punto de interés respecto a las baterías es el uso que se les da. Nos preguntamos si las computadoras se usan mayormente enchufadas o desenchufadas, y qué tanto se depende de la batería en el día a día. A continuación presentamos una visualización de la carga de la batería y el uso general de las laptops en lo referente a tenerlas enchufadas o no.

Las figuras 4.1 y 4.2 muestran cuatro gráficas con la variación de carga de la batería a lo largo de cuatro días para cuatro ceibalitas diferentes.

(a) Dispositivo A **(b)** Dispositivo B

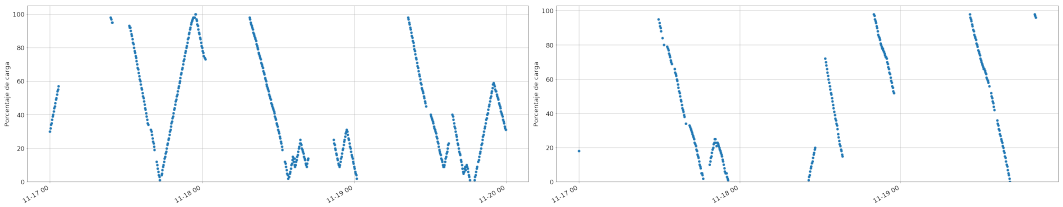


Figura 4.1: Carga de la batería de dos ceibalitas diferentes A y B

Estos datos reflejan que la mayor parte del tiempo, las ceibalitas son usadas desenchufadas. Los datos son similares para otras ceibalitas que no incluímos en la visualización, indicando que probablemente esta forma de uso sea generalizada.

(a) Dispositivo C **(b)** Dispositivo D

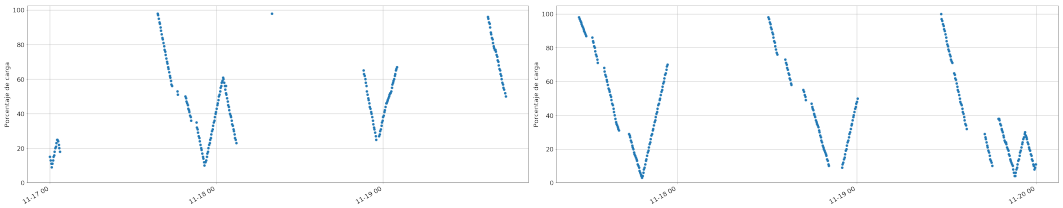
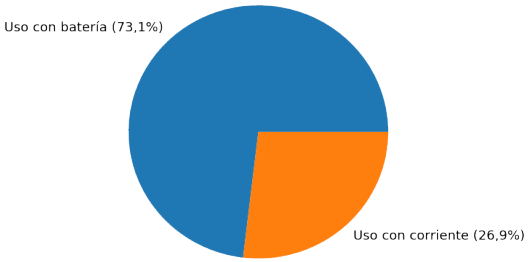


Figura 4.2: Carga de la batería de dos ceibalitas diferentes C y D

Además, podemos inferir que la mayoría de la carga sucede cuando las computadoras están apagadas — un 73.1 % del total de las entradas sucedieron con la computadora desenchufada (tener en cuenta que hay datos solamente de cuando la computadora está prendida y con acceso a internet). Pensamos que la mayoría de las ceibalitas siguen un patrón similar al de las gráficas de más arriba, donde es cargada durante la noche y usada sin enchufar durante el día. Asumimos que en la mayoría de los casos esos períodos sin datos donde la computadora está cargando es porque está apagada, pero es posible también que en esos períodos esté prendida, cargando, pero sin conexión a internet. En la figura 4.3 mostramos el porcentaje de las entradas en la base de datos en las que la computadora estaba conectada a la corriente frente a las que no.

Figura 4.3: Cantidad de entradas con batería contra cantidad de entradas con corriente



Carga y descarga de batería

En esta sección presentamos más análisis sobre los datos de la batería, particularmente sobre su rendimiento a partir de datos de descarga. Este parámetro es de mucha importancia para los clientes, porque permite analizar varios aspectos relevantes tales como el rendimiento de la batería, los hábitos del usuario típico para cargar los dispositivos, y el cambio en la autonomía de los equipos con el paso del tiempo. Además, este tipo de análisis puede ser utilizado para

disparar mantenimiento preventivo o incluso reemplazo de dispositivos, para acortar el tiempo de respuesta en casos de fallas de hardware.

Figura 4.4: Histograma de porcentajes de carga de batería

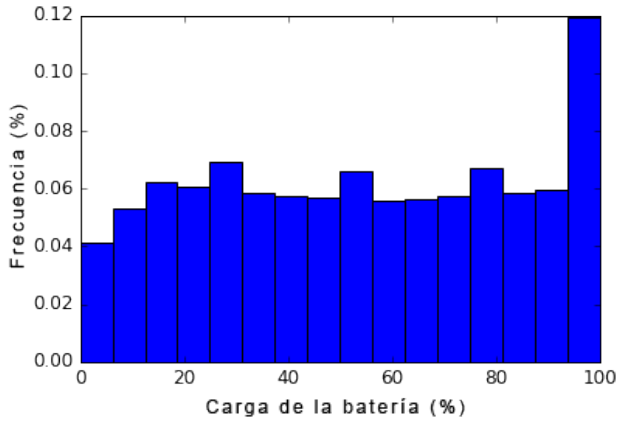
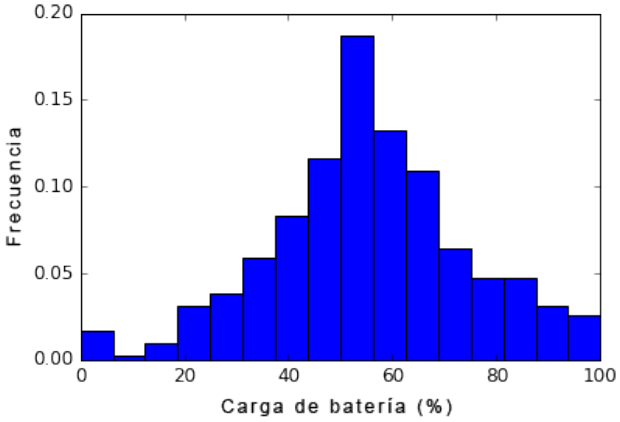


Figura 4.5: Histograma de medianas de porcentaje de batería para cada dispositivo



La figura 4.4 muestra la distribución empírica para los datos recolectados de carga de batería, considerando cada medida de manera individual, sin realizar agregación por dispositivo. Es importante notar que el pico en el último conjunto es explicado por los dispositivos que están enchufados a la corriente y están completamente cargados, ya que en ese caso la carga reportada es 100 %. Dejando de lado este último conjunto, en apariencia los datos se aproximan a una distribución uniforme.

Luego, los datos son agregados por dispositivo, tomando la mediana de los valores para cada uno. La figura 4.5 corresponde a la distribución resultante para ese caso. Como esperamos, la distribución resultante es bien aproximada

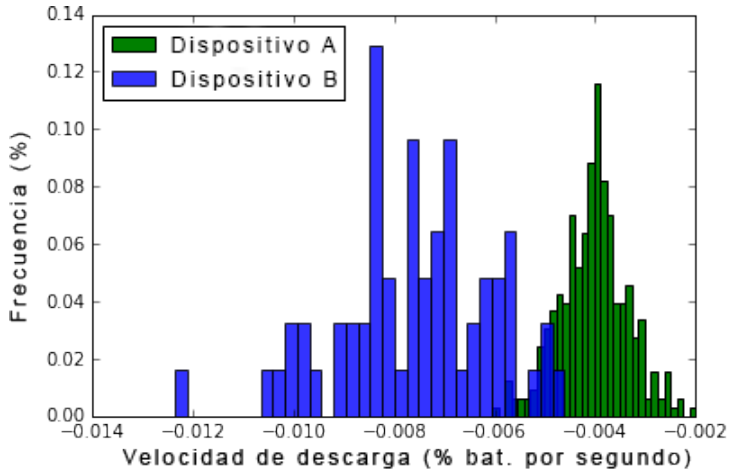
por una distribución gaussiana, ya que la mediana y el promedio son muy similares en los datos recolectados.

Para analizar más profundamente en los datos recolectados, consideramos únicamente los dos modelos de laptop que tuvieron una mayor presencia en el piloto — en adelante A y B. Primero, procesamos la serie temporal de las medidas de la carga de la batería y calculamos el coeficiente de descarga para cada dispositivo (es decir, la pendiente). Para esto, realizamos una regresión lineal de todas las curvas de descarga, utilizando la librería de python *scikit-learn*[83]. Luego, para tener un único valor de descarga por dispositivo, tomamos la mediana de los estimados de todas las curvas de cada dispositivo. La figura 4.6 muestra la distribución empírica de la velocidad de descarga estimada para cada laptop, considerando los valores para dispositivos de tipo A y de tipo B. Utilizando estos valores, es posible calcular la autonomía estimada de cada dispositivo — el tiempo que un equipo con la batería completa demora en descargarse por completo.

Para los dos modelos de laptop analizados, comparmos la autonomía estimada de los dispositivos con medidas tomadas en condiciones de laboratorio por los clientes, utilizando sus procesos de evaluación de equipos estándar. Para cada dispositivo, el Plan Ceibal lleva a cabo dos pruebas de autonomía: una en condiciones de consumo bajo (brillo de pantalla bajo sin que se realice ninguna actividad), y otra en condiciones de consumo elevado (brillo de pantalla alto y reproducción de video en alta calidad).

En la figura 4.7 presentamos la distribución de la autonomía empírica estimada para cada dispositivo. Las líneas rojas verticales corresponden a las medidas de laboratorio en condiciones de bajo y alto consumo respectivamente. Como podemos observar, para ambos modelos de laptop, la mayoría de los dispositivos tienen valores estimados (calculados con las medidas de campo) dentro del rango dado por los valores mínimos y máximos de autonomía medidos en el entorno de laboratorio. Para dispositivos de tipo A, el 51 % caen dentro del rango de medidas de laboratorio, mientras que para el tipo B el porcentaje sube a 92 %. Dispositivos con una autonomía estimada por encima del máximo de laboratorio pueden corresponder a condiciones de uso de baja energía que consumen aún menos batería que las condiciones extremas de laboratorio. Además, con esta comparación es posible identificar las baterías con rendimiento peor al esperado, es decir, las que están por debajo de la autonomía mínima de laboratorio.

Figura 4.6: Histograma compuesto de velocidad de descarga



(a) Dispositivos A **(b)** Dispositivos B

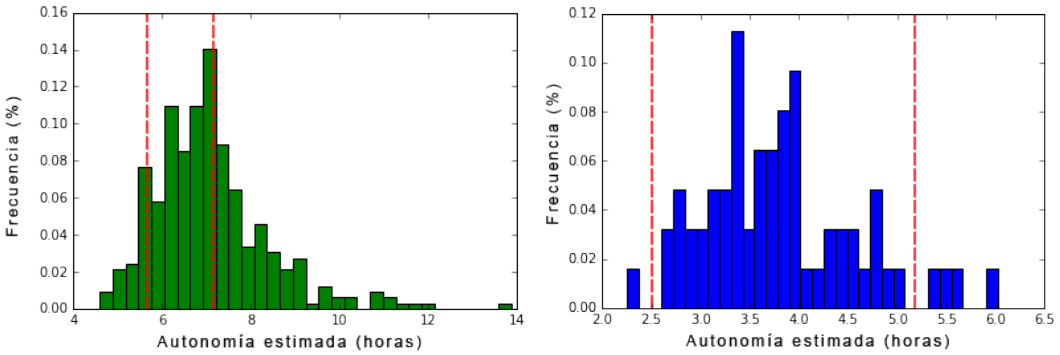


Figura 4.7: Histograma de autonomía aproximada para los dos modelos

Este ejemplo ilustra la utilidad del sistema propuesto y las analíticas asociadas, en este caso, aplicado administración de batería. Mostramos que es posible identificar baterías en malas condiciones, permitiendo por ejemplo alertas de usuario automáticas para reemplazo de batería, contribuyendo a la mejora de la experiencia de usuario. Esto es sólo un ejemplo sencillo del potencial ofrecido por la plataforma desarrollada.

Si contáramos con datos a lo largo de años, se podría medir el rendimiento de las baterías a medida que envejecen, y con datos de uso de aplicaciones, sería posible medir cómo cambia el consumo de batería según las aplicaciones en uso. Los administradores del sistema podrían desarrollar análisis de datos particular basado en los datos recolectados, y/o, dependiendo de los casos de uso proyectados, los parámetros monitoreados pueden ser personalizados,

simplemente mediante el desarrollo de scripts para recolectar nuevos datos luego de una actualización del sistema.

4.2.3. Rendimiento del sistema

Con rendimiento del sistema nos referimos a problemas de escalabilidad y rendimiento de los recursos del sistema. Este aspecto está muy ligado a la plataforma utilizada (AWS). En nuestro caso, el sistema pudo escalar sin problemas al momento de desplegar el código en las 2000 ceibalitas iniciales, y no encontramos ninguna demora o problemas de procesamiento desde entonces.

Concluimos entonces que el sistema tuvo un excelente rendimiento a lo largo del proyecto.

4.2.4. Conclusiones generales del prototipo

Si bien la cantidad de datos recolectados fue menor a la planificada en el inicio del proyecto, creemos que conseguimos una buena cantidad de datos, suficiente para hacer una buena evaluación de ellos. El análisis realizado fue simple, pero es una simple muestra de lo que se podría llegar a hacer con ellos.

El propósito de este proyecto era más que nada el desarrollo de un prototipo de sistema utilizando las plataformas IoT actuales, concentrándonos en la arquitectura desarrollada para su eventual evaluación y desarrollo por parte del Plan Ceibal. Aún así, estamos muy conformes con los resultados en lo referente a la recopilación de datos.

Capítulo 5

Conclusiones y trabajo a futuro

En este capitulo presentamos un análisis de los problemas, interrogantes y proceso de desarrollo del proyecto de grado. Hablamos de las enseñanzas aprendidas tanto en el estudio del ecosistema actual de IoT y de plataformas en la nube, y de las nuevas arquitecturas en auge dentro de la comunidad del software. Finalizamos el capítulo presentando ideas interesantes para seguir el proyecto que fueron apareciendo en el desarrollo del prototipo. Muchas de ellas fueron descartadas por cuestiones de tiempo o porque no eran del todo compatibles con el propósito principal de este proyecto, pero que serían de gran interés investigarlas en este relevamiento del ecosistema de IoT.

5.1. Conclusiones

Elaboramos un análisis del estado del arte en lo referente a tecnologías en el ecosistema de IoT, concentrándonos en el escenario propuesto de desarrollo de un sistema de recolección de datos a través de muchos sensores. Realizamos luego un relevamiento de las plataformas IoT existentes en la actualidad, encontramos sus características comunes, sus diferencias, llegando a una definición de qué es exactamente una plataforma IoT y las cosas que debe tener.

Tomando en cuenta el relevamiento realizado, construimos un prototipo de sistema IoT de recolección de datos para el Plan Ceibal. Utilizamos las ceibalitas como sensores, y desarrollamos un módulo que recolecta datos del dispositivo y los envía a nuestro broker MQTT desplegado en una de las plataformas IoT relevadas, AWS.

Desarrollamos una arquitectura capaz de agregar dispositivos sin interven-

ción de personas, creando certificados únicos para cada uno, permitiendo que no exista un punto único de falla. Además, cada computadora tiene permisos muy específicos en lo referente a lo que pueden o no hacer en nuestro sistema. Esta arquitectura fue desplegada con los servicios provistos por AWS y es *serverless*, requiriendo ningún tipo de mantenimiento de servidores por parte de personas y logrando escalar a cualquier cantidad de dispositivos que se desee tener en el sistema.

Gracias al prototipo realizado, presentamos un informe de consideraciones y decisiones importantes a tomar al momento de embarcarse en el desarrollo de un sistema dentro del ecosistema IoT.

Luego de dos meses de despliegue del prototipo, recabamos una gran cantidad de datos, logrando hacer un pequeño análisis sobre ellos y dejando abierta la posibilidad que cualquier persona interesada dentro del Plan Ceibal tenga datos reales y recientes sobre el rendimiento de alrededor de 300 ceibalitas.

Por último, este trabajo contribuyó a la realización de una publicación científica llamada "Managing a one-to-one computing educational program over an IoT infrastructure", actualmente en revisión para ser publicado en el journal *Software: Practice and Experience*[84].

5.2. Trabajo a futuro

Las diferentes ideas y propuestas que describimos en esta sección no fueron incluidas en el prototipo por cuestiones de alcance y tiempo. Las proponemos tanto como un trabajo de investigación dentro del ecosistema de IoT como un trabajo de ampliación del prototipo, que puede ser útil tanto para Plan Ceibal como para proyectos similares.

5.2.1. Envío de datos bidireccional

Una de las grandes ventajas de utilizar MQTT mencionadas en la sección MQTT es la conexión bidireccional que se establece entre el dispositivo (*ceibalita*) y el servidor (*broker MQTT*). Nuestro prototipo no utiliza esta conexión bidireccional, pues simplemente envía datos de los sensores al servidor, y nunca en sentido contrario. Proponemos como trabajo a futuro hacer uso de esta característica a través de las siguientes dos funcionalidades.

Medición de datos bajo demanda

Manteniendo una conexión activa con cada ceibalita es posible enviar un mensaje MQTT en simultáneo a cada una de ellas, con la instrucción de ejecutar el programa de recolección de datos. La ejecución del programa seguiría normalmente, eventualmente volviendo a utilizar la conexión abierta para mandar los datos.

Esto es particularmente útil para evaluar el estado actual de la red, pudiendo generar capturas de datos instantáneas en un momento específico, posiblemente basados en eventos particulares.

Lograr esto es posible gracias a que el sistema puede soportar cualquier número de mensajes por minuto, sin estar condicionado a la cantidad de dispositivos. Picos de carga son soportados sin cambiar configuraciones y sin necesidad de intervención de una persona, debido a su arquitectura *serverless*, explicado en la sección AWS y Diseño e implementación en el servidor.

Actualización de código a distancia

El despliegue del código es una tarea que requiere de bastante coordinación entre diferentes grupos de trabajo. En nuestro caso, implicó desarrollar el paquete a medida, entregarlo al equipo de Ceibal, y desplegarlo junto con la siguiente actualización del sistema operativo en una cantidad específica de ceibalitas. El largo y dificultoso proceso de despliegue hace que un simple parche pueda demorar días en ser puesto en producción.

Una solución a este problema es realizar un despliegue único (como se hizo en esta instancia), y enviar un mensaje de aviso que se necesita una actualización de una de las dos siguientes formas:

- Utilizando la conexión bidireccional MQTT, enviar un mensaje en simultáneo a todas las ceibalitas (o parte de ellas), comunicando que es necesario actualizar el código fuente.
- Poseer algún servicio en la nube (ya sea un archivo de texto plano, un servicio de caché o un *endpoint* en alguna de nuestras APIs) de consulta de versionado. Así, cada determinado tiempo, las ceibalitas chequean su versión de código contra la más nueva, y si es necesario, se realiza una actualización.

Una vez establecido que es necesario realizar una actualización, el código

de la ceibalita (que corre en su propio entorno virtual, explicado en detalle en la sección Entorno virtual) es eliminado, y la última versión es descargada de un servicio de almacenamiento de datos como AWS S3 [85]. Al finalizar, cron ejecutará la nueva versión del código sin inconveniente ya que no hay que instalar ninguna clase de librerías o paquetes, todo viene listo para ejecutarse *out of the box*.

5.2.2. Análisis de datos

Proponemos un trabajo mayor en lo referido al análisis de datos. Con esto nos referimos a utilizar algoritmos de procesamiento para gran cantidad de datos, pudiendo sacar conclusiones mejores conclusiones, dependientes de las necesidades de los clientes.

Otro aspecto en lo referido al análisis de datos es el análisis en tiempo real. Proponemos construir un componente de nuestra infraestructura que analice en tiempo real el cambio en los datos proporcionados. En caso de datos de red, por ejemplo, se podría tener un *heat map* con el estado actual de diferentes centros. Para lograr algo de este estilo sería necesario cruzar los datos provenientes de las ceibalitas con las ubicaciones geográficas de los routers del Plan Ceibal. Antes de hacer esto se tendría que realizar un estudio buscando posibles problemas de privacidad de datos.

5.2.3. Recolección de datos sin conexión a internet

Nuestro sistema funciona exclusivamente si la ceibalita está conectada a internet. Si no es así la conexión MQTT falla y se termina la ejecución, intentándose nuevamente cinco minutos después en la siguiente interacción de cron.

Sería bueno que los datos recolectados mientras no esté conectada a internet fueran almacenados en el sistema, y se enviaran en la próxima conexión exitosa. Para esto proponemos un sistema de almacenamiento temporal de datos en las computadoras. Apenas la ceibalita encuentre conexión a internet, los datos guardados localmente serían enviados y acto seguido serían borrados del almacenamiento local.

Este flujo se puede separar en dos módulos, uno encargado de la recolección de datos y persistencia local temporal, y otro que tome dichos datos y los intente mandar al broker. Esto presentaría el beneficio de que sería sencillo

cambiar la frecuencia de recolección de datos sin cambiar la frecuencia de envío.

5.2.4. Visualización de datos

La gran cantidad de datos recolectados hace que un usuario sin acceso a la base de datos o a programas que analicen dichos datos no tenga una buena forma de visualizarlos. Proponemos entonces un sistema de visualización de datos a través de una página web.

Para construir dicho sistema es necesario crear una nueva API para servir los datos a través de HTTPS, que ejecute una nueva función Lambda con conexiones a las bases de datos. El *front end* sería servido a través de S3 como una página web estática (conjunto de archivos HTML y Javascript, renderizada en el navegador). Utilizando estas tres tecnologías (S3 para servir de forma estática la página web, y Lambda y API Gateway para servir el *back end*) seguiríamos respetando al arquitectura *serverless* de todo el sistema, permitiendo que la visualización de datos sea auto escalable.

5.2.5. Servicio de recolección de datos más completo

Desde el comienzo del proyecto la idea de recopilar más información de las redes y del uso de las ceibalitas estuvo presente. Al avanzar en el prototipo nos dimos cuenta que algunas variables, especialmente las de uso de aplicaciones, eran difíciles de conseguir, y ya que el enfoque principal del proyecto siempre estuvo en la infraestructura y la plataforma IoT generada, fueron descartadas. Presentamos entonces algunas cuestiones de recolección y análisis de datos que pueden ser incorporadas al proyecto.

Nuevas variables del sistema

Sería bueno en el futuro ampliar la cantidad de variables recolectadas, tanto para obtener otro tipo de métricas como para aprovechar el prototipo generado, ya que agregar dichos atributos no requeriría cambiar nada en la infraestructura. Especialmente métricas de uso de aplicaciones, para poder evaluar si el software adquirido para las ceibalitas cumple lo prometido, es usado de la forma que se quiere, etc.

Preprocesamiento de datos en los dispositivos

Dependiendo de los recursos disponibles en las ceibalitas, qué tan grande puede llegar a ser el paquete con datos enviado (por limitaciones de la red) o qué tanto poder de procesamiento la infraestructura tenga, existe la posibilidad de hacer pre procesamiento de datos en los dispositivos.

En nuestro caso no fue necesario ya que el costo de procesar los datos en AWS era mínimo, pero si se quisiera hacer un análisis mayor o abaratar costos de servicio, se podría implementar algoritmos de procesamiento de datos en las ceibalitas. Esto trae la consecuencia que el costo del programa corriendo en las ceibalitas sea mayor, y podría llegar a traer problemas de rendimiento.

5.2.6. Control de acceso al sistema

Como mencionamos en la sección Registro de ceibalitas, el sistema posee un certificado único para cada ceibalita, pudiéndose activar o desactivar cuando sea necesario (como fue mencionado en la sección Problemas e imprevistos encontrados). Tener que crear un conjunto de programas que hagan esto es tedioso, y la solución que desarrollamos es puntual al problema encontrado. Sería bueno en el futuro tener un sistema de activación y desactivación de dispositivos más amigable para administradores del sistema, que no implique el uso de programas hechos a medida.

Para esto proponemos el desarrollo de una página web con autenticación para administradores del sistema, en donde se puedan listar todas las ceibalitas, con opciones de activación/desactivación o eliminación para cada una de ellas. Esta página podría ser una extensión de la visualización de datos mencionada en la sección anterior Visualización de datos, y también podría incorporar opciones de recopilación de datos de manera instantánea y a demanda mencionadas en la sección Medición de datos bajo demanda.

Lista de figuras

APÉNDICES

Apéndice 1

Descripción en detalle de la implementación del software de los dispositivos

En este apéndice describimos en detalle el funcionamiento de las distintas partes del sistema que se ejecuta en las computadoras del plan Ceibal.

El código descripto en esta sección, al igual que cualquier otro código utilizado en este proyecto se puede encontrar en nuestro repositorio público[86].

1.1. Estructura general

Como describimos en la sección Diseño e Implementación, la instalación copia unos archivos a un directorio en particular, y configura crontab para que ejecute el envío de datos cada 5 minutos.

1.2. Implementación

`datagetter.py`

Aquí está la lógica de obtención de datos. Contiene las clases `NetworkDataGetter` y `SystemDataGetter`. Estas clases son el núcleo de la recolección de datos del dispositivo, ya que contienen la lógica de la obtención de datos. Utilizan la biblioteca `sh`[61] para ejecutar comandos Unix e interpretan las respuestas de los comandos (normalmente utilizando expresiones regulares) Estos datos normalizados serán los que se envíen al servidor.

`SystemDataGetter` se encarga de obtener todos los datos del sistema, tales como número de serie, uso del disco, memoria y carga del procesador. `NetworkDataGetter` se encarga de obtener todos los datos de la red, tales como la dirección MAC, la dirección IP y el uso de la red.

Si la obtención de alguno de los datos falla, retorna el *string* vacío y genera una entrada en el *log* de errores — tanto el local como el remoto.

`credentials.py`

Este *script* se encarga de generar las credenciales para que el dispositivo pueda enviar mensajes MQTT al servidor de Amazon. Al ejecutarlo, envía un HTTP POST a una URL de Amazon con el número de serie del dispositivo como contenido. El número de serie es obtenido usando `SystemDataGetter`. La respuesta del servidor contiene un JSON con las credenciales requeridas para poder autenticarse. Estas se guardan en el directorio `./.credentials`:

- `publish.cert.pem` contiene el certificado PEM.
- `publish.private.key` contiene la clave privada.
- `publish.public.key` contiene la clave pública.
- `certificate.id` contiene el identificador único de Amazon de esta clave. Se utiliza más adelante para identificar de qué dispositivo proviene cada mensaje.

Además, en ese directorio se incluye — estáticamente, en la instalación misma — el certificado raíz, `root-CA.crt`.

`data_processor.py`

Contiene una función, `create_json()`, que genera el paquete JSON a enviar, con todas las claves y los datos que el servidor espera.

`mqtt_sender.py`

Contiene una función, `send_payload(topic, payload)`, que se conecta al servidor utilizando las credenciales y envía el *string* `payload` al *topic* `topic`. Esta funcion es utilizada tanto por el envío principal de datos como por el *logging* remoto de errores.

`mqtt_client.py`

El *script* principal de este pequeño sistema, es el que ejecuta Cron cada 5 minutos. Ejecuta `create_json` para obtener el paquete a enviar, y `send_payload` para enviarlo al servidor.

Apéndice 2

Descripción del proceso de despliegue de la plataforma

En este apéndice explicaremos los pasos a seguir para desplegar la plataforma dentro de AWS como se realizó en el prototipo. Esto se puede realizar de forma gratuita dentro del *AWS free tier*.

El código descripto en esta sección, al igual que cualquier otro código utilizado en este proyecto se puede encontrar en nuestro repositorio público[86].

2.1. AWS Lambda

Para desplegar lambda es necesario general un paquete de despliegue (*deployment package*). Esto se puede realizar de forma automática utilizando librerías como serverless[87], o a mano. Si es este el caso, es necesario que todo el código necesario para que corra se encuentre dentro, esto incluye librerías. En nuestro caso, es necesario tener la librería psycopg2[88] de python para poder interactuar con la base de datos.

En el dashboard de lambda se sube dicho paquete y la función quedará disponible para su uso. Hace falta elegir qué disparador va a hacer que se ejecute. Para esto se indica que queremos utilizar API Gateway, y AWS se encarga del resto. Una vez finalizado, se nos indica cuál será la API a la que tendremos que realizar las *requests*.

Se tendrá que configurar bien los roles de la función, ya que necesita acceso a RDS y al broker para el caso del registro de datos y accesso al broker para el registro de ceibalitas.

2.2. DynamoDB

Para crear la tabla de dynamo para almacenar los errores es necesario indicarle la *hash key* a utilizar. En nuestro caso, el serial y el timestamp del error.

2.3. RDS

El proceso de creación de la base de datos es igual que cualquier otra base de datos SQL. Se establecen las prestaciones necesarias y luego de unos minutos la instancia queda creada, y es posible acceder utilizando la dirección y credenciales establecidas.

A continuación mostramos un dump para crear la tabla principal de la base de datos:

```
CREATE TABLE measurements_v2 (
    mac_addr macaddr NOT NULL,
    ip_addr cidr NOT NULL,
    "timestamp" timestamp without time zone NOT NULL,
    ap_mac_addr macaddr,
    rx_bytes bigint,
    tx_packets_overruns bigint,
    tx_packets_carrier bigint,
    tx_packets_errors bigint,
    tx_packets_dropped bigint,
    tx_packets_quantity bigint,
    rx_packets_overruns bigint,
    rx_packets_frame bigint,
    rx_packets_errors bigint,
    rx_packets_dropped bigint,
    rx_packets_quantity bigint,
    frequency real,
    missed_beacon bigint,
    charging boolean,
    invalid_misc bigint,
    uptime integer,
```

```
    battery_temp real,
    tx_excessive_retries integer,
    rssi integer,
    battery_power smallint,
    boot_time character varying(100),
    load_avg_5_min real,
    total_memory_kb bigint,
    free_memory_kb bigint,
    buffers_memory_kb bigint,
    cached_memory_kb bigint,
    total_swap_memory_kb bigint,
    free_swap_memory_kb bigint,
    root_dir_total_disk_space_kb bigint,
    root_dir_free_disk_space_kb bigint,
    home_dir_total_disk_space_kb bigint,
    home_dir_free_disk_space_kb bigint,
    serial_number character varying(100)
);
```

2.4. IoT Broker

Configurar el broker es la parte más manual del proceso. Una vez creado, se deben establecer las dos reglas que maneja, el registro de datos y de errores.

Para el registro de datos es necesario configurar la validación con el certificado proporcionado. Para esto utilizamos una policy que especifica que los dispositivos pueden publicar en $ceibalitas/ < CERTIFICATE-ID > /logs$. Esta validación se hace automáticamente en cada publicación de aquí en más. Una vez realizado esto sólo hace falta establecer la conexión con lambda, registrando un disparador para el topic $ceibalitas/ + /logs$.

El caso del registro de errores es mucho más simple, ya que existe una regla pre programada para guardar los datos en una tabla de dynamo. Se selecciona esta regla y se indica a qué tabla hay que mandar los datos.

Referencias bibliográficas

[1] D. Miorandi et al. Internet of things: Vision, applications and research challenges. Ad Hoc Networks, 10(7):1497–1516, September 2012. doi: 10.1016/j.adhoc.2012.02.016.

[2] Estadísticas de IoT por Gartner. https://www.gartner.com/newsroom/id/3598917. Accedido: Feb 2018.

[3] IBM Watson IoT. https://www.ibm.com/internet-of-things, . Accedido: Feb 2018.

[4] Microsoft Azure IoT Suite. https://azure.microsoft.com/en-us/suites/iot-suite/, . Accedido: Feb 2018.

[5] FIWARE IoT Stack. https://fiware-iot-stack.readthedocs.io/en/latest/, . Accedido: Feb 2018.

[6] Kaa IoT Platform. https://www.kaaproject.org/, . Accedido: Ene 2018.

[7] AWS IoT. https://aws.amazon.com/iot/, . Accedido: Feb 2018.

[8] Objetivo del Plan Ceibal. https://www.ceibal.edu.uy/es/institucional, . Accedido: Feb 2018.

[9] Dispositivos del Plan Ceibal. https://www.ceibal.edu.uy/es/dispositivos, . Accedido: Feb 2018.

[10] Infraestructura del Plan Ceibal. http://blogs.ceibal.edu.uy/tecnologia/nuestros-equipos/, . Accedido: Feb 2018.

[11] Cifras oficiales del Plan Ceibal. https://www.ceibal.edu.uy/es/articulo/ceibal-en-cifras, . Accedido: Feb 2018.

[12] I.F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: a survey. Computer Networks, 38(4):393 – 422, 2002. ISSN 1389-1286. doi: https://doi.org/10.1016/S1389-1286(01)00302-4. URL http://www.sciencedirect.com/science/article/pii/S1389128601003024.

[13] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. Computer Networks, 54(15):2787 – 2805, 2010. ISSN 1389-1286. doi: https://doi.org/10.1016/j.comnet.2010.05.010. URL http://www.sciencedirect.com/science/article/pii/S1389128610001568.

[14] Jennifer Yick, Biswanath Mukherjee, and Dipak Ghosal. Wireless sensor network survey. Computer Networks, 52(12):2292 – 2330, 2008. ISSN 1389-1286. doi: https://doi.org/10.1016/j.comnet.2008.04.002. URL http://www.sciencedirect.com/science/article/pii/S1389128608001254.

[15] Philips Hue Bridge. https://www.meethue.com/en-us/p/hue-bridge/046677458478. Accedido: Feb 2018.

[16] Nest. https://nest.com/. Accedido: Feb 2018.

[17] MQTT v3.1.1 official documentation. http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html, . Accedido: Ene 2018.

[18] AWS IoT protocols. https://docs.aws.amazon.com/iot/latest/developerguide/protocols.html, . Accedido: Feb 2018.

[19] Soporte de MQTT para Azure IoT. https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-mqtt-support, . Accedido: Mar 2018.

[20] IANA Service Name and Transport Protocol Port Number Registry. https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml?search=8883, . Accedido: Ene 2018.

[21] OASIS Advanced Message Queuing Protocol (AMQP) Version 1.0, Security. http://docs.oasis-open.org/amqp/core/v1.0/amqp-core-security-v1.0.html#section-security-layers. Accedido: Mar 2018.

[22] RabbitMQ. http://www.rabbitmq.com/. Accedido: Mar 2018.

[23] STOMP web socket plugin para RabbitMQ. https://www.rabbitmq.com/web-stomp.html. Accedido: Mar 2018.

[24] RFC 7228. https://tools.ietf.org/html/rfc7228. Accedido: Mar 2018.

[25] Jiehan Zhou et al. Create your own internet of things: A survey of iot platforms. pages 651–657, June 2013. ISSN 978-1-4673-6084-5. doi: 10.1109/CSCWD.2013.6581037.

[26] Kaa Transports. https://docs.kaaproject.org/display/KAA/Transports, . Accedido: Feb 2018.

[27] AWS IoT Thing Shadow Service. https://docs.aws.amazon.com/iot/latest/developerguide/iot-thing-management.html, . Accedido: Feb 2018.

[28] Azure IoT Identity Registry. https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-devguide-identity-registry, . Accedido: Feb 2018.

[29] Kaa Endpoint Provisioning and Registration. https://kaaproject.github.io/kaa/docs/v0.10.0/Programming-guide/Key-platform-features/Devices-provisioning-and-registration/, . Accedido: Feb 2018.

[30] AWS IoT Rules. https://docs.aws.amazon.com/iot/latest/developerguide/iot-rules.html, . Accedido: Feb 2018.

[31] Fiware IoT Complex Event Processing API. https://fiware-iot-stack.readthedocs.io/en/latest/cep_api/, . Accedido: Feb 2018.

[32] AWS IoT Thing Shadow Service. https://docs.aws.amazon.com/iot/latest/developerguide/iot-device-shadows.html, . Accedido: Feb 2018.

[33] Azure IoT Hub Device Twins. https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-devguide-device-twins, . Accedido: Feb 2018.

[34] AWS CLI. https://aws.amazon.com/cli/, . Accedido: Feb 2018.

[35] Documentación de Azure CLI. https://docs.microsoft.com/en-us/cli/azure/?view=azure-cli-latest, . Accedido: Abril 2018.

[36] Documentación de IBM Bluemix CLI. https://console.bluemix.net/docs/cli/index.html#overview, . Accedido: Abril 2018.

[37] Firmware Management using IBM Watson IoT Platform. https://www.ibm.com/developerworks/community/blogs/Informix%20Bit%20&%20Pieces/entry/Firmware_Management_using_IBM_Watson_IoT_Platform?lang=en, . Accedido: Feb 2018.

[38] Kaa supported platforms. https://kaaproject.github.io/kaa/docs/v0.10.0/Programming-guide/Using-Kaa-endpoint-SDKs/, . Accedido: Feb 2018.

[39] AWS Lambda. https://aws.amazon.com/documentation/lambda/, . Accedido: Nov 2017.

[40] Arquitectura de AWS API Gateway. https://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html, . Accedido: Ene 2018.

[41] Arquitectura de AWS API Gateway. https://aws.amazon.com/api-gateway/pricing/, . Accedido: Ene 2018.

[42] Costos de AWS RDS. https://aws.amazon.com/rds/postgresql/pricing/. Accedido: Ago 2017.

[43] Giuseppe DeCandia et al. Dynamo: Amazon's Highly Available Key-value Store. 2007.

[44] Costos de AWS DynamoDB. https://aws.amazon.com/dynamodb/pricing/. Accedido: Ago 2017.

[45] AWS IoT documentation. https://aws.amazon.com/documentation/iot/, . Accedido: Nov 2017.

[46] JWT. https://jwt.io/. Accedido: Feb 2018.

[47] OAuth. https://oauth.net/. Accedido: Feb 2018.

[48] State of the cloud report - Rightscale - 2017. http://www.offis.com.au/static/media/uploads/download_files/rightscale-2017-state-of-the-cloud-report.pdf. Accedido: Mar 2018.

[49] IBM Device Schema. https://console.bluemix.net/docs/services/IoT/im_schemas.html#iotrtinsights_task, . Accedido: Ene 2018.

[50] NodeRED. https://nodered.org/. Accedido: Ene 2018.

[51] NodeRED. https://github.com/Azure/azure-iot-protocol-gateway/blob/master/README.md, . Accedido: Ene 2018.

[52] Bluemix is now IBM Cloud. https://www.ibm.com/blogs/bluemix/2017/10/bluemix-is-now-ibm-cloud/, . Accedido: Feb 2018.

[53] KaaIoT Technologies has been established as the company to lead all Kaa innovations. https://www.kaaiot.io/company/kaaproject-expands-into-kaaiot-technologies-company/, . Accedido: Feb 2018.

[54] PostScapes IoT Cloud Platform Landscape. https://www.postscapes.com/internet-of-things-platforms/. Accedido: Ene 2018.

[55] State of IoT - 2015 Global Developer Study. https://www.progress.com/docs/default-source/default-document-library/progress/documents/papers/iot_surveyreport.pdf. Accedido: Ene 2018.

[56] Comparing 11 IoT Development Platforms. https://dzone.com/articles/iot-software-platform-comparison. Accedido: Ene 2018.

[57] V. Gazis, M. Görtz, M. Huber, A. Leonardi, K. Mathioudakis, A. Wiesmaier, F. Zeiger, and E. Vasilomanolakis. A survey of technologies for the internet of things. 6(2):1090–1095, Aug 2015. ISSN 2376-6492. doi: 10.1109/IWCMC.2015.7289234.

[58] 5 Things To Know About The IoT Platform Ecosystem. https://iot-analytics.com/5-things-know-about-iot-platform/, . Accedido: Ene 2018.

[59] K. J. Singh and D. S. Kapoor. Create your own internet of things: A survey of iot platforms. IEEE Consumer Electronics Magazine, 6(2):57–68, April 2017. ISSN 2162-2248. doi: 10.1109/MCE.2016.2640718.

[60] A. Botta et al. Integration of cloud computing and internet of things: A survey. Future Generation Computer Systems, 56(1):684–700, March 2016. doi: 10.1016/j.future.2015.09.021.

[61] sh Python Library. https://amoffat.github.io/sh/. Accedido: Ene 2018.

[62] subprocess Python Library. https://docs.python.org/2/library/subprocess.html. Accedido: Ene 2018.

[63] paho Python Library. https://www.eclipse.org/paho/. Accedido: Ene 2018.

[64] aws-iot-device-sdk-python Python Library. https://github.com/aws/aws-iot-device-sdk-python, . Accedido: Ene 2018.

[65] Documentación de Virtualenv. https://virtualenv.pypa.io/en/stable/. Accedido: Feb 2018.

[66] Debian Policy Manual, capítulos 3, 4 y 5. https://www.debian.org/doc/debian-policy/#document-ch-binary. Accedido: Feb 2018.

[67] aws-iot-device-sdk-python Python Library. https://github.com/jordansissel/fpm. Accedido: Ene 2018.

[68] Cron manpage. http://man7.org/linux/man-pages/man8/cron.8.html. Accedido: Ene 2018.

[69] AWS. https://aws.amazon.com/documentation/, . Accedido: Nov 2017.

[70] AWS Api Gateway. https://aws.amazon.com/documentation/apigateway/, . Accedido: Nov 2017.

[71] AWS IoT Policies. http://docs.aws.amazon.com/iot/latest/developerguide/iot-policies.html, . Accedido: Nov 2017.

[72] AWS Cloudwatch. https://aws.amazon.com/documentation/cloudwatch/. Accedido: Nov 2017.

[73] AWS Educate. https://aws.amazon.com/education/awseducate/, . Accedido: Ene 2018.

[74] Boto3. https://boto3.readthedocs.io/en/latest/. Accedido: Feb 2018.

[75] Google Cloud Functions. https://cloud.google.com/functions/docs/. Accedido: Feb 2018.

[76] Azure Functions. https://azure.microsoft.com/en-us/services/functions/, . Accedido: Feb 2018.

[77] Apache Openwhisk. https://github.com/apache/incubator-openwhisk. Accedido: Feb 2018.

[78] Casos de uso de Carriots. https://www.carriots.com/use-cases. Accedido: Jun 2017.

[79] AWS Lambda FAQ. https://aws.amazon.com/lambda/faqs/, . Accedido: Feb 2018.

[80] AWC EC2 Auto scaling group. https://docs.aws.amazon.com/autoscaling/ec2/userguide/AutoScalingGroup.html. Accedido: Feb 2018.

[81] Integraciones de Kaa. https://docs.kaaproject.org/display/KAA/Kaa+IoT+Platform+Home, . Accedido: Feb 2018.

[82] Datacenter mi nube de Antel. https://minubeantel.uy/index.php. Accedido: Feb 2018.

[83] Documentación de regresión lineal provista por Scikit Learn. http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html. Accedido: Jun 2017.

[84] Journal "Software: Practice and Experience

. https://onlinelibrary.wiley.com/journal/1097024x. Accedido: Mar 2018.

[85] Documentación de AWS S3. https://aws.amazon.com/documentation/s3/. Accedido: Feb 2018.

[86] Repositorio con el código utilizado en este proyecto. https://github.com/fosimani/proyectoiotceibal. Accedido: Abril 2018.

[87] Serverless framework. https://serverless.com/. Accedido: Mar 2018.

[88] Psycopg2. http://initd.org/psycopg/docs/. Accedido: Mar 2018.