UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Frontend y backend en Node.js para una aplicación IoT

Trabajo Final de Grado

Presentado en la facultad

Escola Tècnica Superior d'Enginyeria de Telecomunicació de Barcelona

Universitat Politècnica de Catalunya

por

Jonathan Camacho Torrens

En cumplimiento parcial

de los requerimientos para el grado en

Ingeniería Telemática

Tutor: Jose Luis Muñoz Tapia

Barcelona, Febrero 2018

Abstract

This project intends to study the state of the art of the IoT and create a temperature and humidity monitoring application that simulates a real IoT scenario.

Firstly, there will be a brief explanation about the IoT architectures and then the main problems facing the IoT will be addressed, such as the lack of standardization, the diversity of existing protocols or the lack of security.

Afterwards, an IoT application will be created, which will consist of a Raspberry Pi with a sensor that will monitor the temperature and humidity of a room, a server with a database to host the sensor information on it and a client to graphically represent the obtained data.

To allow the communication between the Raspberry and the server, a specific IoT cloud-based service will be used, such as Azure IoT Hub, so that a possible way to implement an application based on the IoT is offered in a simple and safe way.

Resum

Aquest projecte té la intenció d'estudiar l'estat de l'art del IoT i crear una aplicació de monitorització de la temperatura i la humitat que simuli un escenari IoT real.

Primerament es farà una breu explicació sobre les arquitectures IoT i després es tractaran els principals problemes als que s'enfronta l'IoT, com la falta d'estandardització, la diversitat de protocols existents o la falta de seguretat.

A continuació es crearà una aplicació IoT que consistirà en una Raspberry Pi amb un sensor que monitoritzarà la temperatura i humitat d'una sala, un servidor amb una base de dades on allotjar la informació i un client per a poder representar les dades obtingudes mitjançant gràfiques.

Per realitzar la comunicació entre la Raspberry i el servidor, s'utilitzarà un servei *cloud* específic per a l'IoT com és l'Azure IoT Hub, de manera que s'ofereix una possible manera d'implementar una aplicació basada en l'IoT de forma senzilla i segura.

Resumen

Este proyecto tiene la intención de estudiar el estado del arte del IoT y crear una aplicación de monitorización de temperatura y humedad que simule un escenario IoT real.

Primeramente se hará una breve explicación sobre las arquitecturas IoT y luego se tratarán los principales problemas a los que se enfrenta el IoT, como la falta de estandarización, la diversidad de protocolos existentes o la falta de seguridad.

A continuación se creará una aplicación IoT, que consistirá en una Raspberry Pi con un sensor que monitorizará la temperatura y la humedad de una sala, un servidor con una base de datos donde alojar la información y un cliente para poder representar mediante gráficas los datos obtenidos.

Para realizar la comunicación entre Raspberry y servidor, se utilizará un servicio *cloud* específico para el IoT como es Azure IoT Hub, de manera que se ofrece una posible manera de implementar una aplicación basada en el IoT de forma sencilla y segura.

Agradecimientos

Me gustaría dedicar unas líneas para agradecer a un conjunto de personas el apoyo y/o formación que me han dado y que han hecho que esto haya sido posible.

Para comenzar, quiero dar las gracias a todos los profesores que he tenido durante mi vida, en la escuela y la universidad, por formarme como profesional y como persona durante todos estos años de estudio.

En concreto, agradecerle a mi tutor José Luís Muñoz Tapia toda la ayuda prestada para poder realizar este proyecto y todo el soporte brindado durante el curso.

También quería darle las gracias a mi familia, por apoyarme en los momentos difíciles, por confiar en mí, por haber tenido paciencia conmigo y apoyarme en cualquier decisión que he tomado.

Por último y no menos importante, me gustaría agradecer a Everis la oportunidad que me ha dado de poder formar parte de su equipo de trabajo y poder realizar el proyecto en esta empresa. En concreto, darle las gracias a mi tutor de empresa Javier de la Cueva Orts por su apoyo moral y técnico durante la realización de la tesis, sin el cual esto no habría sido posible.

Historial de revisiones y registro de aprobación

Revisión	Fecha	Propósito
0	18/10/2017	Creación del documento.
1	27/11/2017	Revisión del documento.
2	16/01/2018	Revisión del documento.
3	23/01/2018	Revisión final y validación del documento.

LISTA DE DISTRIBUCIÓN DEL DOCUMENTO

Nombre	e-mail
Estudiante: Jonathan Camacho Torrens	jonacamm@gmail.com
Tutor UPC: Jose Luís Muñoz Tapia	jose.munoz@entel.upc.edu
Tutor Everis: Javier de la Cueva Orts	javier.de.la.cueva.orts@everis.com

Escrito por:		Revisado y aprobado por:	
Fecha	18/10/2017	Fecha	23/01/2018
Nombre	Jonathan Camacho Torrens	Nombre	Jose Luís Muñoz Tapia
Posición	Autor del proyecto	Posición	Tutor del proyecto

Tabla de contenidos

Lista de figuras

1. Introducción

1.1. Declaración de objetivos

Los objetivos de este proyecto estarán divididos en dos partes. Habrá una parte más teórica, donde se pretenderá conocer como son las arquitecturas IoT, las dificultades y retos que nos podemos encontrar a la hora de utilizar este tipo de arquitecturas y qué medidas podemos tomar para mejorar la seguridad de nuestras aplicaciones IoT. La segunda parte será más práctica y se pretenderá crear una aplicación IoT desde cero con el fin de monitorizar la temperatura y la humedad de una sala.

Por lo tanto, los principales objetivos del proyecto son:

1. Estudiar de forma teórica las arquitecturas IoT, los retos que presentan y posibles mejoras de seguridad a implementar en las aplicaciones.
2. Creación de un backend en Node.js que interactúe con un emisor de datos, un cliente y una base de datos.
3. Creación de un cliente en Angular que recoja datos y los represente gráficamente.
4. Creación de un cliente en Node.js que recoja datos de un sensor y los envíe a un servidor.

1.2. Requisitos y especificaciones

Los requisitos que debe cumplir la aplicación IoT construida son los siguientes:

- La Raspberry Pi (RP) debe ser capaz de comunicarse con el sensor de temperatura.
- La RP debe poder mandar información (cifrada o no) procedente del sensor al servidor.
- El servidor debe procesar los datos, almacenarlos en la base de datos y enviarlos a un cliente (frontend).
- El cliente deberá poder mostrar los datos en gráficas.
- Se debe verificar que los datos no han podido ser comprometidos por posibles atacantes.

Dichos requisitos se han puesto en desarrollo utilizando las siguientes especificaciones:

- Para la comunicación de la RP con el sensor, se utilizará un puerto paralelo a través de la placa Grovepi+.
- Para la comunicación de la RP con el servidor se utilizará el protocolo HTTP sobre WebSockets para la primera versión de la aplicación, y el protocolo MQTT sobre SSL/TLS para la segunda versión mejorada.
- La tecnología usada para hacer funcionar la RP y el backend en general será el lenguaje Node.js.
- Para construir el cliente, se utilizará Angular 2.
- Como base de datos se utilizará una tecnología NoSQL, en concreto MongoDB.

1.3. Plan de trabajo y diagram de Gantt

1.3.1. Paquetes de trabajo y tareas

Proyecto: Frontend y backend en Node.js para una aplicación IoT		
Estudio teórico de arquitectura y seguridad	Hoja 1 de 2	
Resumen: Estudio de la arquitectura y la seguridad tanto para tecnologías web como para IoT.	Fecha inicio planteada: 12/10/2017 Fecha fin planteada: 18/10/2017	
	Comienzo real: 12/10/2017 Fin real: 18/10/2017	
Tarea interna T1: Estudio de las arquitecturas: qué son y las diferencias web vs IoT. Tarea interna T2: Estudio de estrategias para mejorar la seguridad en entornos IoT.		

Construcción segunda versión aplicación	Hoja 1 de 2	
Resumen: Construcción de aplicación IoT y su correspondiente evaluación de seguridad.	Fecha inicio planteada: 28/11/2017 Fecha fin planteada: 18/12/2017	
	Comienzo real: 19/10/2017 Fin real: 30/11/2017	
Tarea interna T1: Construcción del frontend y del backend de		

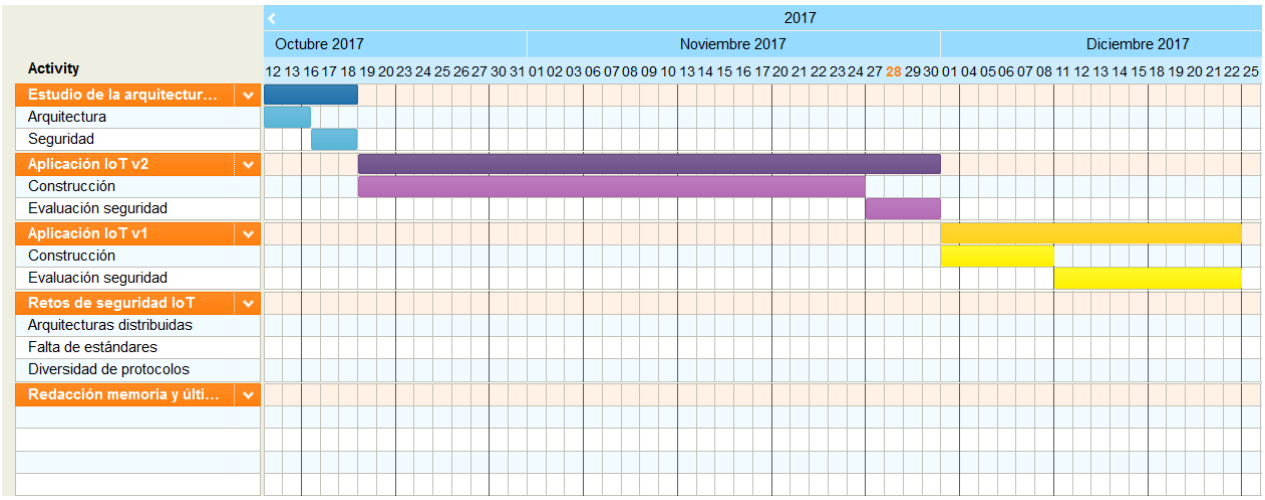
la aplicación. Tarea interna T2: Evaluación de seguridad de la aplicación.		

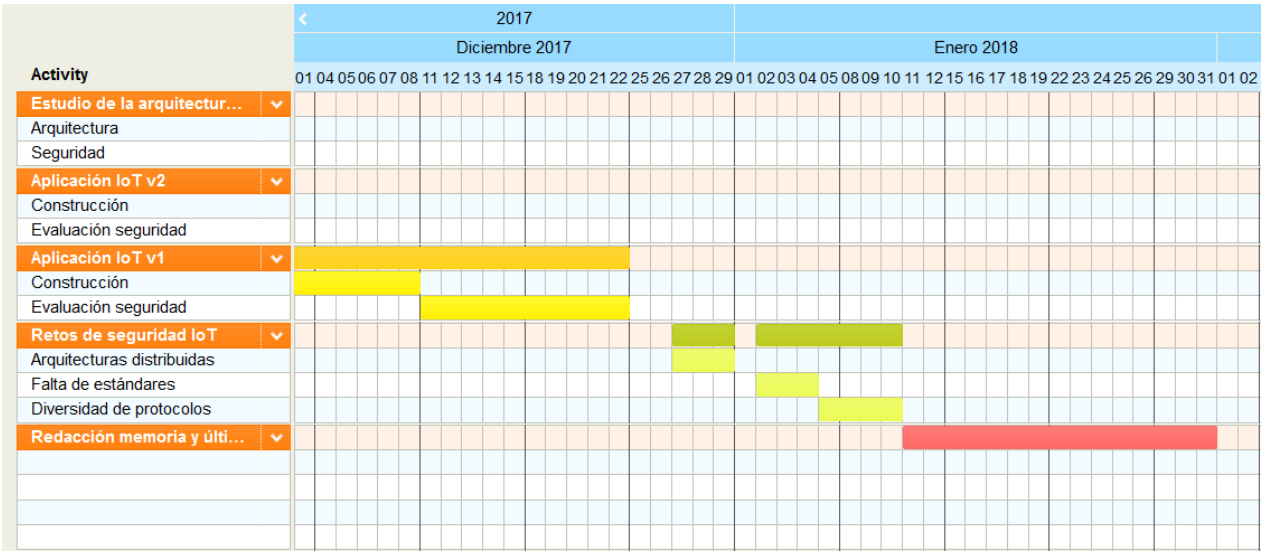
Construcción primera versión aplicación	Hoja 2 de 2	
Resumen: Construcción de aplicación IoT y su correspondiente evaluación de seguridad.	Fecha inicio planteada: 19/10/2017 Fecha fin planteada: 13/11/2017	
	Comienzo real: 01/12/2017 Fin real: 22/12/2017	
Tarea interna T1: Construcción del backend de la aplicación. Tarea interna T2: Evaluación de seguridad de la aplicación.		

Estudio teórico sobre los retos en la seguridad IoT	Hoja 2 de 2	
Resumen: Estudio de los retos que presenta la implementación del Internet de las Cosas.	Fecha inicio planteada: 14/11/2017 Fecha fin planteada: 27/11/2017	
	Comienzo real: 27/12/2017 Fin real: 10/01/2018	
Tarea interna T1: Estudio de las arquitecturas distribuidas. Tarea interna T2: Estudio de la falta de estándares de seguridad. Tarea interna T3: Estudio sobre la diversidad de protocolos existentes.		

Redacción memoria y últimas pruebas	Hoja 2 de 2	
Resumen: Redacción de la tesis, realización de pruebas y preparación de una demostración.	Fecha inicio planteada: 08/01/2018 Fecha fin planteada: 31/01/2018	
	Comienzo real: 11/01/2018 Fin real: 31/01/2018	
Tarea interna T1: Redacción de la tesis. Tarea interna T2: Revisión de la aplicación y últimos detalles. Tarea interna T3: Preparación de la presentación.		

1.3.2. Diagrama de Gantt





2. Estado del arte

Introducción IoT

El concepto de internet de las cosas fue propuesto por Kevin Ashton en el Auto-ID Center del MIT en 1999.

Es un concepto que surge de la interconexión de dispositivos a través de internet, intentando crear una capa digital que permita conectar objetos cotidianos entre sí para explotar los beneficios que se puedan conseguir con los datos obtenidos.

El ímpetu de las empresas por dominar un mercado tecnológico virgen ha generado un ecosistema heterogéneo compuesto por multitud de soluciones incapaces de interoperar entre ellas. De hecho, se estima que en el año 2020 habrá más de 20.000 millones de dispositivos conectados.

Esto provoca que la cantidad de aplicaciones IoT crezca a un ritmo altísimo, pero poniendo en riesgo algunos elementos, como por ejemplo la seguridad.

Para que esta tecnología pueda expandirse correctamente, será necesario tener en cuenta tres conceptos:

- Estandarización de las arquitecturas.
- Controlar la diversidad de protocolos existentes.
- Crear mejoras de seguridad.

2.1. Arquitecturas IoT

2.1.1. ¿Qué son?

Hoy en día, todavía no hay ningún organismo internacional oficial que haya propuesto un estándar sobre cómo construir una arquitectura que sea aceptado globalmente. De hecho, es sabido que hay muchos buscando una solución a este problema, como ya hemos nombrado anteriormente, lo que provoca que haya multitud de ideas y propuestas. Pero la realidad es que ninguna es más válida que otra, puesto que todas están en desarrollo a la espera de encontrar la mejor solución para este problema.

Aun así, cuando hablamos sobre las arquitecturas del Internet de las Cosas, es necesario nombrar al proyecto europeo más importante en este campo, el IoT-A (Internet-of-Things-Arquitecture). Este proyecto intenta poner en común ideas, conceptos y definiciones para tener una base teórica sobre la que construir las arquitecturas IoT.

Utilizaremos alguna de las bases y definiciones que se encuentran en dicho proyecto para, por ejemplo, comparar el modelo de comunicación del IoT con el actual.

Si bien es cierto que todavía a nivel global no se construye en base a las definiciones de este proyecto, sí que se sigue un patrón básico de construcción a la hora de diseñar un sistema basado en IoT.

Dicho sistema, basa su funcionamiento en 3 elementos principales:

2.1.1.1. Sensores y/o actuadores

Uno de los objetivos del Internet de las Cosas es poder recopilar, procesar y tratar información de un sistema o entorno de forma automática y para nuestro beneficio. Para realizar este proceso de recolección de información del medio, se utilizan dispositivos llamados coloquialmente “cosas”. Los dispositivos más utilizados son los sensores (ya sean de temperatura, humedad, presión, movimiento, etc) que nos permiten convertir magnitudes físicas o químicas en señales eléctricas, de forma que podamos guardar esos datos posteriormente.

Finalmente, estos sensores recogerán la información para dársela, por ejemplo, a un actuador, que utilizará la información eléctrica para realizar una acción. Entre algunos de los ejemplos de actuadores que podemos utilizar en el IoT se encuentran pantallas LCD, cámaras, impresoras, micrófonos, etc.

Otra opción, que suele ser la más común en muchos casos, es la de enviar la información recopilada por los dispositivos hacia un Gateway, el segundo de los elementos principales de esta arquitectura.

2.1.1.2. Gateways

Debido a la naturaleza de los sensores o “cosas”, que es muy diferente a la de los clientes web o de escritorio, necesitamos un elemento intermedio en esta arquitectura que actúe de proxy entre el mundo de los dispositivos y el centro de datos.

Por ejemplo, los sensores normalmente tienen unas capacidades muy limitadas en cuanto a conectividad de red. Es muy común que estos sensores utilicen el protocolo Bluetooth Low Energy (BLE) o que tengan conectividad utilizando el protocolo Zigbee, entre muchos otros. Pero la mayoría de esos protocolos tienen una cosa en común, y es que no se pueden conectar directamente a Internet.

Por lo tanto, se hace necesario el uso de un Gateway que pueda proporcionar una salida a redes externas utilizando, por ejemplo, Wifi o GSM. Con esta función, quedaría solucionado el problema de conectividad hacia internet, o hacia los sistemas backend encargados de recibir la información.

Aun así, este Gateway no funciona solamente como un simple proxy que envía la información al centro de datos, sino que además tiene otras funciones igualmente importantes. Entre ellas se encuentran el pre-procesado de los datos obtenidos por los sensores, la monitorización de dichos sensores y también puede actuar como proveedor de software y actualizaciones de seguridad para los dispositivos que están en su dominio.

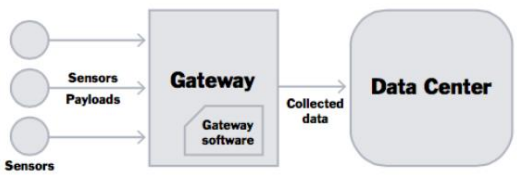


Figura 1. Elementos de una arquitectura IoT

En la aplicación creada en este proyecto, la función de filtro y pre-procesado la llevará a cabo la RP, mientras que de la ingesta de datos y del punto de acceso dentro de la red se encargará el servicio IoT Hub de la plataforma Azure.

2.1.1.3. Centro de datos

El centro de datos es uno de los elementos más importantes dentro de una arquitectura IoT. Al fin y al cabo, el objetivo de un sistema IoT no es simplemente captar información, sino utilizar esa información para obtener cierto conocimiento que pueda ser utilizado comercialmente o en beneficio propio. Para este fin, es necesario que las ingentes cantidades de información que se reciben por parte de los sensores establecidos en diferentes redes, sean almacenadas y tratadas de forma que se puedan aprovechar al máximo todos los datos obtenidos.

Estos centros de datos se enfrentan a varios problemas. Unos problemas que, por cierto, son relativamente nuevos, ya que coinciden con el auge del Big Data.

Aunque los Gateways filtren ciertos mensajes, la cantidad de información que reciben los centros de datos es ingente y el problema no se soluciona simplemente aumentando la capacidad de los discos.

Típicamente los sistemas de almacenamiento convencionales guardaban ficheros de formato similar, lo cual no generaba ningún problema. En el IoT, en cambio, hay que diferenciar dos tipos de ficheros. Existen los de gran tamaño, ya sean fotografías o vídeos capturados desde smartphones o cámaras de seguridad IP, y los de pequeño tamaño, como pueden ser ficheros de datos capturados por sensores. Estos sensores, pueden crear millones de archivos de pequeño tamaño que deben poder ser accesibles de forma aleatoria.

Por lo tanto, se necesitan dos tipos de sistemas de almacenamiento. Unos para la entrada/salida (I/O) de grandes archivos de manera secuencial, y otros para la entrada/salida de pequeños archivos de forma aleatoria.

A la hora de almacenar los datos, cuando hablamos del Internet de las Cosas, es muy habitual que los sistemas hagan uso de las tecnologías *cloud*.

2.1.2. Diferencias entre arquitecturas Web e IoT

Modelo de comunicación

Tal y como se ha comentado en apartados anteriores, al no haber un estándar concreto de cómo debe ser una arquitectura IoT, se realizará una comparación entre una arquitectura web tradicional y el modelo de comunicación que describe el proyecto europeo IoT-A.

A la hora de implementar protocolos que permitan comunicación entre máquinas en un entorno web, se hace uso del modelo o pila OSI, el cual está dividido en siete capas o niveles. La pila de comunicaciones propuesta para el IoT intenta tomar como referencia el modelo OSI, inspirándose también en el modelo TCP/IP.

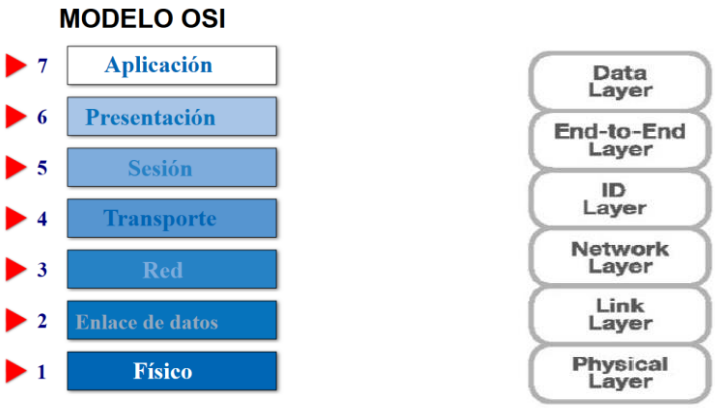


Figura 2. Modelo OSI vs Modelo OSI modificado para el IoT

Como se puede ver en la figura anterior, existen 5 capas para el modelo IoT que desarrollan las siguientes funciones:

Capa Física: Su definición es similar a la de la capa física de la pila OSI, por lo que intenta asegurar la interoperabilidad de las características físicas de las tecnologías de comunicación utilizadas en el sistema IoT, para asegurar la convivencia de las tecnologías actuales con las venideras.

Capa de Enlace de datos: Para que los sistemas IoT logren una interoperabilidad total, así como el soporte de tecnologías heterogéneas y un marco de seguridad integral, esta capa debe permitir la diversidad de protocolos. Pero al mismo tiempo, necesita proveer a las capas superiores unas capacidades e interfaces uniformes.

Capa de red: En esta capa, de nuevo, se proveen las mismas características que en la capa de la pila OSI correspondiente. Sin embargo, con el fin de conseguir una capacidad de gestión, una interoperabilidad y una escalabilidad globales, esta capa debe proporcionar un paradigma de comunicación común para cada solución de red posible.

Capa ID: Esta capa intenta ofrecer un marco de resolución común para el IoT. Además, los servicios de seguridad o autenticación explotarán esta capa para proveer un direccionamiento uniforme para los distintos dispositivos y tecnologías del IoT.

Capa Extremo a Extremo (*End-to-End):* Esta capa se encarga de funcionalidades de traducción, el soporte de proxies/gateways y el ajuste de los parámetros de configuración cuando se utilizan diferentes entornos de red.

Capa de Datos: Es el nivel más alto del modelo de comunicación IoT. Su propósito es asegurarse que toda información generada por cualquier objeto de un sistema IoT sea tratado de acuerdo al Modelo de Información, también propuesto por el IoT-A.

Paradigma de comunicación

En cuanto a los protocolos, existen algunas diferencias entre los que se utilizan para web y los destinados al IoT.

Los protocolos web, como ya sabemos, utilizan el paradigma cliente-servidor mientras que en el IoT, se suele utilizar el sistema publicar-subscribir. Aun así, ambos pueden utilizar los dos paradigmas.

Los protocolos **cliente-servidor** requieren que el cliente se conecte a un servidor y realice solicitudes. Esto quiere decir, que el servidor tiene los datos y es el cliente el que debe conocer de antemano acerca del servidor y ser capaz de conectarse.

En cambio, los protocolos **publicar-subscribir** requieren que los dispositivos se conecten a un gestor intermediario y publiquen la información. Entonces, los clientes pueden conectarse a este gestor y subscribirse para obtener la información. La ventaja principal es que este modelo desacopla al productor de datos del consumidor de datos.

Los protocolos cliente-servidor funcionan mejor cuando se conoce la infraestructura principal. Por ejemplo, si se sabe que un servidor esta en situado en la dirección IP x.x.x.x en el puerto 10000, el cliente puede conectarse y hacer peticiones.

En cambio, los protocolos publicar-subscribir son mejores cuando la infraestructura es desconocida. Por ejemplo, cuando los productores de datos (como los sensores en un coche) cambian de red por movilidad, o tienen una conectividad intermitente, es más fácil para ellos publicar sus datos cuando tienen línea y pueden hacerlo, despreocupándose así de estar localizables por cualquier cliente.

Aun así, cada red tiene sus necesidades e infraestructura y puede haber perfectamente sistemas IoT que se compongan de un cliente y un servidor utilizando protocolos como HTTP.

2.2. Seguridad IoT

Uno de los problemas a los que se enfrenta el IoT, es el haber avanzado en la implementación de esta tecnología en diferentes sectores sin tener en cuenta la seguridad desde un principio. Esto, junto con la heterogeneidad de las tecnologías que se utilizan en todos los niveles de la arquitectura, provoca que a día de hoy la seguridad en el ámbito IoT sea todavía una asignatura pendiente. Aun así, hay varias medidas que pueden tomar tanto consumidores como fabricantes para garantizar una mejor protección de los datos.

Recomendaciones básicas de seguridad

Por ahora, hasta que no se imponga un estándar acogido por todas las empresas que garantice unos mínimos en cuanto a seguridad, la securización de los sistemas del Internet de las Cosas recae sobre los fabricantes, los administradores de sistemas y los usuarios, principalmente. Desde ElevenPaths, empresa del grupo Telefónica dedicada a la innovación en soluciones de seguridad, se publicó en 2015 una serie de medidas que deberían tomar tanto los usuarios como los fabricantes para mantener una infraestructura lo más segura posible. Algunas de las más importantes son las siguientes:

Recomendaciones para usuarios y consumidores:

- Cambiar las contraseñas por defecto por unas robustas y utilizar el cifrado más robusto posible.
- Conectar dispositivos a una red separada, cuando sea factible.
- Deshabilitar o proteger el acceso remoto a dispositivos mientras no sea necesario.
- Investigar y aprovechar las medidas de seguridad implantadas en el dispositivo.
- Deshabilitar características no utilizadas.
- Instalar actualizaciones tan pronto estén disponibles.

Recomendaciones para fabricantes:

- Utilizar conexiones cifradas.
- Anonimizar los datos y recopilarlos sólo cuando sea estrictamente necesario.
- Requerir un cambio obligatorio de las contraseñas por defecto por otras robustas y no permitir contraseñas “quemadas” en el código.
- Permitir la configuración de reglas detalladas de control de acceso.
- Implantar medidas que dificulten ataques de fuerza bruta para adivinar credenciales de acceso.
- Verificación mutua de certificados SSL y de listas de revocación de certificados.
- Implementar medidas inteligentes de *fail-safe* cuando la conexión o la energía del dispositivo falla.
- Realizar análisis de seguridad del código fuente y ofuscarlos si es accesible para los usuarios.

Privacidad en las comunicaciones

Una de las grandes ventajas del IoT es la gran cantidad de información que se puede recolectar y aprovechar gracias a los avances del Big Data. Pero lo que a priori parece una ventaja puede convertirse en un inconveniente.

Las normas de protección de datos no avanzan al mismo ritmo que la tecnología, por lo que ante la expansión que se producirá en los próximos años en el mundo IoT, se hace necesario crear nuevas normativas para asegurar la protección de los datos personales.

Se pueden llevar a cabo mejoras en la privacidad en el IoT. Alguna de las posibles estrategias que pueden adoptar los fabricantes para intentar solventar este problema son:

- Bloquear las conexiones salientes (cuando sea posible) para evitar que cualquier observador pueda ver los flujos de datos de los dispositivos.
- Cifrar las peticiones DNS para evitar que un observador identifique los dispositivos.
- En ciertos escenarios, como en una *"casa inteligente"*, pasar todo el tráfico del hogar inteligente a través de una VPN, de forma que no se pueda relacionar el tráfico originado desde un asistente del hogar a dispositivos individuales.
- Inyectar tráfico para limitar la confianza de un observador al intentar identificar dispositivos, por ejemplo enmascarando patrones de tráfico interesantes.

2.2.1. Estrategias para mejorar la seguridad en entornos IoT

Una posible estrategia para asegurar las comunicaciones en el IoT es utilizando sistemas de cifrado para garantizar la seguridad de los datos que se transmiten.

Lo cierto es que, se use la técnica que se use, se debe tener en cuenta que uno de los principales problemas a la hora de securizar los sistemas IoT es la carencia de recursos computacionales presente en muchos de los dispositivos conectados a la red.

Una de las soluciones que se pueden implementar es la estrategia de aligerar el trabajo a los dispositivos finales y hacer que la red se ocupe de la mayor parte de todas estas tareas de gestión y securización.

Para llevar esta estrategia a cabo y hacer más segura la transmisión de datos, se presentan unos requerimientos necesarios:

1. Los dispositivos no deben tener puertos de entrada abiertos

En un servicio web común, si un servidor quiere mandar datos a un cliente o dispositivo, este debe estar escuchando. Hasta ahora este sistema ha funcionado correctamente, pero en el mundo IoT trabajar de este modo no es adecuado, ya que tener millones de dispositivos con puertos abiertos indefinidamente es un riesgo que no se debe tomar, ya que entre las posibles amenazas destacan las posibles infecciones de malware, robo de datos, ataques de denegación de servicio (DoS), entre otras.

Es por ello que los dispositivos conectados a una red segura deberían permitir solo las conexiones salientes. Esta medida disminuiría en gran medida las amenazas a las que se exponen estos dispositivos.

Para llevar a cabo estas medidas, es necesario utilizar la estructura publicar/subscribir que se ha explicado en apartados anteriores. Algunos protocolos que pueden usar una comunicación segura y confiable como son MQTT, CoAP, WebSockets o HTTP 2.0 son capaces de implementar el paradigma de comunicación publicar/subscribir entre dispositivos sin puertos abiertos. Para mantener la escalabilidad que necesita el IoT, las conexiones basadas en publicar/subscribir deberán ser gestionadas por servidores de alto rendimiento distribuidos por todo el mundo.

2. Cifrado extremo a extremo

Uno de los protocolos más utilizados para cifrar las comunicaciones es el protocolo SSL/TLS. Este protocolo protege el nivel superior de transmisión de datos entre dispositivos, cifrándolo punto a punto hasta llegar al otro extremo de su comunicación. Aunque SSL/TLS es adecuado para la transmisión segura de datos, los datos generados por los dispositivos IoT pueden ser vulnerados en algunos puntos de la comunicación.

Para garantizar la seguridad extremo a extremo, los datos se deberían cifrar utilizando el protocolo AES (Advanced Encryption Standard). El uso de este protocolo garantiza que solo los extremos de la comunicación con las claves de cifrado adecuadas podrán descifrar el mensaje.

El problema de utilizar el protocolo AES radica en la limitación que ofrece en algunos escenarios en los que quizás un punto intermedio necesita leer, filtrar o analizar parte de los datos o cabeceras. Para solventar este inconveniente, se puede utilizar un sistema de doble cifrado.

Este sistema de doble cifrado propone cifrar el cuerpo del mensaje con AES, pero el resto del mensaje, que puede contener datos clave para elementos intermedios de la comunicación, se cifra solamente en los puntos finales con TLS. Con esta estrategia, los desarrolladores y fabricantes de productos IoT pueden asegurar una comunicación segura y cifrada extremo a extremo permitiendo a su vez procesar y analizar los datos de cabeceras a mitad de camino.

3. Control de acceso basado en tokens

Aunque los protocolos AES y TLS solucionan el problema del cifrado de los datos transmitidos, otro problema muy importante al que se enfrentan los sistemas es tener un buen control de acceso sobre quién y qué puede transmitir y recibir datos.

En una tecnología como es el IoT, donde se espera que millones de dispositivos interconectados estén intentando escuchar los canales correctos, es extremadamente ineficiente e inseguro pedirle a esos mismos dispositivos que tengan que filtrar las conexiones y peticiones a las que no están subscritos. Es por esta razón por la que será la red la que se ocupará de realizar el mayor volumen de trabajo en cuanto a control de acceso.

Dentro del paradigma publicar/subscribir, se puede usar un sistema de control de acceso basado en tokens para distribuir tokens a los dispositivos y así garantizar el acceso a canales de datos específicos. Con este enfoque, se tiene un gran control sobre qué tokens se han creado, qué dispositivos han recibido estos tokens y a qué datos garantizan el acceso estos tokens. Esto también permite tener un control centralizado

sobre cuándo y cómo son revocados los tokens, permitiendo así cortar el flujo de datos a clientes que no han pagado una cuota, por ejemplo.

De esta manera, es la red la que controla el acceso a los dispositivos y la que les permite comunicarse en la red basándose en los tokens repartidos.

4. Monitorización del estado del dispositivo

Tanto en el ámbito industrial como en el del usuario final es realmente importante monitorizar el estado de conexión de los dispositivos. Si un dispositivo deja de enviar o recibir datos, el gestor de dicho dispositivo debe saberlo, ya que un dispositivo offline podría ser el resultado de una manipulación que se esté llevando a cabo, o que ha habido un problema, como un corte de energía.

Los metadatos de seguimiento requieren un canal de datos separado y seguro para monitorizar la presencia de los dispositivos, de modo que cada uno deberá tener un canal de publicación/subscripción para este fin y para enviar otro tipo de alertas. De este modo, si hay dispositivos que pasan a estar desconectados sin ninguna razón, la red podría enviar una notificación a un técnico para revisar el problema.

5. Facilidades para configurar y actualizar dispositivos

Una de las utilidades que tienen las comunicaciones a través de Internet es la de poder acceder a dispositivos remotamente y configurarlos y actualizarlos. Pero también es cierto que no todos los usuarios finales tienen los mismos conocimientos para poder realizar esta tarea.

En el mundo IoT, esto se debería tener muy en cuenta. Imaginemos un caso en el que un usuario compre un kit de 5 cámaras wifi de seguridad para su hogar. Lo que este cliente espera es conectar el dispositivo y que empiece a hacer su función, pero esto es algo muy poco común. En realidad, el usuario deberá configurar las cámaras y estar pendiente de todas las actualizaciones de seguridad que vayan apareciendo. Si bien es algo que puede parecer técnicamente necesario y correcto, es algo que exige un conocimiento por encima del que poseen la media de usuarios, lo que provoca que finalmente no realicen las actualizaciones pertinentes y se expongan a riesgos de seguridad.

Por lo tanto, se deberá aprovechar el paradigma publicar/subscribir para securizar los dispositivos IoT. Con este método, al conectar un dispositivo, este se anunciará a través de un canal de anuncio. Seguidamente, el servidor devolverá un canal privado por donde se podrán comunicar y unas reglas de acceso.

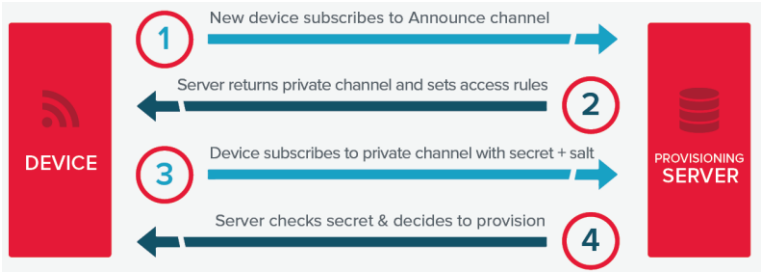


Figura 3. Intercambio de claves utilizando paradigma publicar/subscribir

De esta forma, el fabricante podrá usar el canal de publicación/subscripción para efectuar las actualizaciones necesarias y a su vez, hacer que el usuario se desentienda de todo el proceso y pueda disfrutar de una experiencia de conectar y usar.

La manera en la que se podrá realizar este proceso de actualización será mediante mensajes de broadcast emitidos por el servidor a través de los canales establecidos con los dispositivos para que estos respondan y descarguen el software si es necesario.

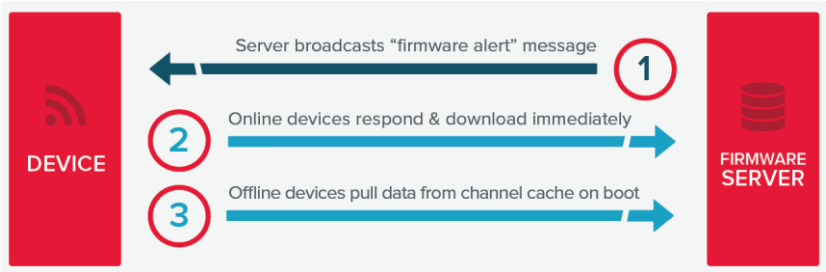


Figura 4. Actualizaciones de seguridad en sistemas publicar/subscribir

2.3. Retos y dificultades del IoT

El IoT no es una tecnología madura y aunque esté parcialmente desplegada, todavía está en desarrollo. En este apartado estudiaremos algunos de los retos más significativos a los que se enfrenta el IoT.

2.3.1. Arquitecturas distribuidas

Como ya hemos comentado en apartados anteriores, las arquitecturas IoT suelen tener los tres componentes principales: los sensores, los centros de datos y los gateways. Actualmente, estos últimos son los que se encargan de la mayoría de procesos de gestión, pero en realidad hay una tendencia hacia crear un sistema cada vez más distribuido.

En una **arquitectura IoT centralizada**, tanto los proveedores de datos como los consumidores de información se conectan a una entidad central (Gateway), por lo que es más sencillo gestionar la autenticación, donde los sensores tienen sus propios proveedores de identidad que establecen una relación de confianza con el Gateway de N-a-1, o las políticas de control de acceso, ya que los dispositivos no deben implementar ninguna lógica de control de acceso: envían la información a la entidad central en la que confían.

En cambio, en una **arquitectura IoT distribuida**, todas las entidades pueden tener la habilidad de recuperar, procesar, combinar y proveer información y servicios a otras entidades. Esto implica que hasta el más simple de los objetos y sensores deberá tener algún tipo de lógica de autenticación. Además, las políticas de control de acceso estarán distribuidas por todos los elementos, lo cual crea un ecosistema con una gran cantidad de políticas heterogéneas.

Además de estos problemas, se añade la dificultad en cuanto a la gestión de claves de seguridad. Esto es lógico, pues no todos los dispositivos tendrán los suficientes recursos como para poder establecer conexiones cifradas y seguras.

Por último, otro reto al que se enfrentan estas arquitecturas es la adaptabilidad. Dependiendo de varios factores como el nivel de criticidad de los datos, puede no ser necesario aplicar fuertes mecanismos de protección a un flujo de información concreto (por ejemplo, confidencialidad en el estado encendido/apagado de una farola).

Sin embargo, a pesar de todos estos retos que plantean las arquitecturas IoT distribuidas, hay ciertos aspectos que mejoran. Uno de ellos es la privacidad. Esto es así debido a que todos los dispositivos tienen mejor control sobre los datos que generan y procesan, permitiendo controlar la granularidad de los datos que enviarán.

La siguiente tabla ofrece a modo resumen las principales diferencias sobre como tratan los retos ambos tipos de arquitecturas:

Retos de seguridad	IoT Centralizada	IoT Distribuida
Identidad y Autenticación	N-a-1	N-a-N
Control de acceso	Políticas homogéneas	Políticas heterogéneas
Seguridad en redes y protocolos	Proveedor centralizado y conocido	Pares desconocidos
Privacidad	Menos flexible	Más flexible

Figura 5. Retos de seguridad según la distribución de dispositivos

2.3.2. Falta de estándares

Como ya se ha comentado en la introducción, uno de los mayores retos a los que se enfrenta el IoT es la falta de estandarización.

El hecho de ser una tecnología virgen y por desarrollar, hace que muchas empresas quieran explotar el IoT lo antes posible para conseguir liderar su sector. Pero si se quiere un Internet de las Cosas globalizado, potente y operativo, no puede haber “guerras” tecnológicas entre diferentes marcas. Y ahí es donde reside el problema. Las compañías que invierten en investigación sobre las tecnologías del IoT necesitan sacar un beneficio de ello y si se salen de su enfoque individualizado para adoptar uno más estandarizado,

podrían perder dinero. Por lo tanto, muchas compañías no tienen ninguna prisa por estandarizar sus sistemas.

Por esa razón, ya existen numerosos servicios y aplicaciones desplegadas, agravando cada vez más el problema, pues estos sistemas y dispositivos siguen corriendo en sus propias plataformas, creando un sistema cada vez más heterogéneo.

Por lo tanto, si a la enorme cantidad de plataformas disponibles para desarrollar aplicaciones IoT añadimos la gran diversidad de protocolos existentes, obtenemos un escenario que dificulta mucho a los desarrolladores crear aplicaciones que puedan ser interoperables.

Aun así hay empresas que intentan facilitar las cosas a los desarrolladores para mejorar sus productos. Por ejemplo, Apple ha hecho todo lo posible para poder interconectar objetos domésticos inteligentes de terceros e integrarlos en su "HomeKit", facilitando algunas APIs a desarrolladores. De esta manera, permiten a cada fabricante integrar su dispositivo con iOS y así poder controlar todos esos dispositivos desde una sola aplicación en lugar de tener una aplicación para cada dispositivo, cosa que también acaba mejorando la experiencia del usuario final.

2.3.3. Diversidad de protocolos

El problema que se desprende del gran número de protocolos existentes para las diferentes tecnologías que operan en Internet, es uno de los mayores retos a los que se enfrenta el desarrollo del IoT.

La ingente cantidad de protocolos dificulta en gran medida todos los aspectos de las comunicaciones, desde el descubrimiento de dispositivos, intercambio de información, seguridad de los datos transmitidos, formato de los datos, entre otros.

Solamente en la capa de enlace encontramos una gran cantidad de protocolos[1] que coexisten actualmente por todo Internet.

Además, no solo es problemática la cantidad de protocolos, sino la incompatibilidad que existe entre muchos de ellos para poder comunicarse, ya sea por operar en frecuencias diferentes o tener un alcance distinto.

[1] Para ver la magnitud de la cantidad de protocolos, ir a Anexo: 8.1 Tabla con ejemplos de protocolos de nivel de enlace.

3. Desarrollo del proyecto:

Introducción

Una vez estudiado el estado del arte en el que se encuentran las tecnologías del proyecto, se creará una aplicación IoT.

Para mostrar cómo es posible mejorar la seguridad y la conectividad en las aplicaciones IoT de forma sencilla, primeramente se creó una primera versión de la aplicación, la cual no implementaba ninguna medida de seguridad (algo más habitual de lo que pensamos en entornos reales) y cuya conectividad estaba limitada a una conexión por cable Ethernet. Para mejorarla, se implementó una segunda versión con conectividad a través de Internet utilizando una plataforma *cloud* específica para el IoT, la cual incorpora además medidas de seguridad[2] bastante robustas.

3.1. Primera versión de la aplicación

El escenario sobre el que se ha desarrollado la primera versión del proyecto es como el de la siguiente figura:

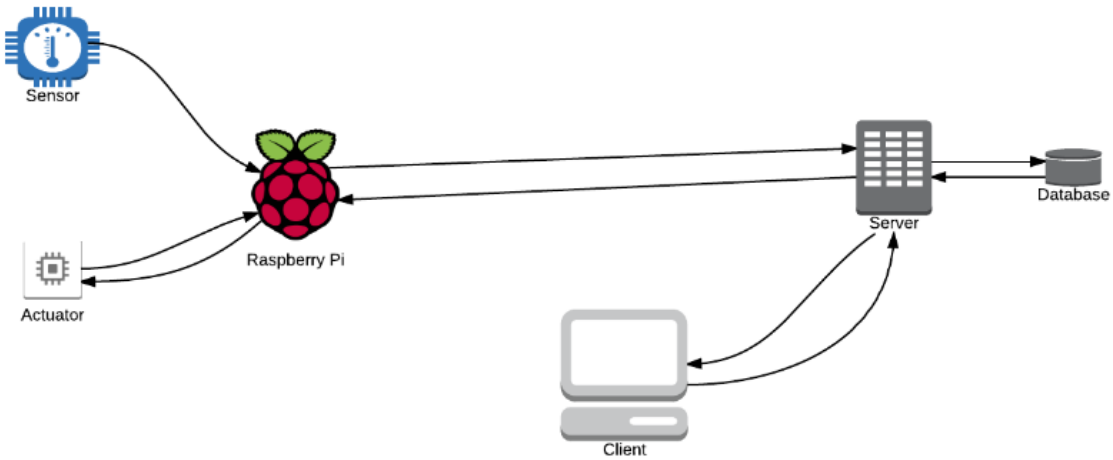


Figura 6. Arquitectura primera versión de la aplicación

Esta arquitectura tiene tres partes[3] principales:

1. Raspberry Pi y componentes
2. Sevidor y base de datos
3. Cliente Angular

[2] Para ver una comparativa en cuanto a seguridad de ambas versiones, ver Anexo: 8.2 Comparativa de seguridad

[3] Para ver un resumen de las especificaciones de los elementos de la arquitectura, ver Anexo: 8.3 Elementos de la arquitectura

Funcionamiento de la aplicación

Con los elementos ya definidos, explicaremos de forma general como se pone en marcha todo el sistema y su funcionamiento, siendo el siguiente:

El primer paso es iniciar la base de datos (MongoDB).

Seguidamente, se iniciará el servidor, el cual se pondrá a escuchar dos puertos. Un puerto (3000) estará dedicado a recibir peticiones de clientes mediante conexiones HTTP. El otro puerto (3001) servirá para establecer una conexión bidireccional con los clientes que lo soliciten utilizando WebSockets[4]. Este mismo puerto, también se utilizará para que la RP pueda conectarse y enviar los datos recogidos por el sensor para ser guardados en la base de datos.

Seguidamente, se deberá ejecutar la aplicación alojada en la Raspberry Pi, la cual pondrá en marcha el sensor y empezará a captar los datos. Paralelamente, se establecerá una conexión con el servidor para mandar la información recogida.

Por último se ejecutará el cliente, el cual establecerá una conexión con el servidor. Mediante esta conexión, se pedirán por un lado los datos guardados por el servidor en la base de datos y por otro lado los datos que vaya recibiendo el servidor en tiempo real. Estos datos se utilizarán para elaborar un par de gráficas sobre la temperatura y la humedad captadas por el sensor.

3.1.1. Desarrollo

Preparación del escenario de desarrollo

Para levantar el escenario mostrado anteriormente, se han tenido que llevar a cabo algunas consideraciones dada la cantidad de elementos en la arquitectura.

En cuanto a la localización de los elementos, la mayoría se encuentran en el portátil proporcionado por Everis, donde se realiza el proyecto. Hablamos del servidor y la base de datos, que corren en los puertos especificados anteriormente, además del cliente Angular que se ejecutará también desde el mismo PC. Por otra parte, tenemos la Raspberry Pi con los sensores y actuadores que por razones logísticas y con el fin de hacer un escenario un poco más real, se encuentran en casa del autor del proyecto.

Por lo tanto, dado que se ha trabajado en este proyecto desde las oficinas de Everis, para interactuar con la RP se ha utilizado una conexión cifrada utilizando un cliente de SSH. En concreto, el cliente utilizado es PuTTY, disponible para Windows y Linux. De esta manera, desde la oficina se ha podido programar la RP, levantar servicios, etc.

El desarrollo de la aplicación se basa, como ya se ha comentado anteriormente, en tres partes fundamentales. A continuación se explicará el desarrollo de cada una de las partes en cuanto a código y funcionamiento.

[4] Para saber por qué se ha utilizado WebSockets en lugar de HTTP, ver Anexo 8.4: WebSocket vs HTTP: ¿Por qué WebSocket?

Cliente Raspberry Pi

Esta parte del sistema es la responsable de captar la temperatura y la humedad que envíe el sensor y de entregarla al servidor. Además, se encargará de comunicarse y hacer funcionar los actuadores que tiene conectados, que en este caso será una pantalla LCD RGB y un LED. Toda esta parte de la aplicación estará programada en Node.js. Algunas de las partes más importantes de su implementación son las siguientes:

Interactuación con la pantalla LCD

Para poder utilizar este actuador, se utilizarán dos librerías:

- ***i2c-bus***: se utilizará para el intercambio de datos entre el bus de datos i2c de la placa y el sensor o la pantalla.
- ***node-grovepi***: nos permitirá utilizar las funcionalidades que ofrece la placa Grovepi+.

Para interactuar con la pantalla, se han utilizado tres funciones:

- ***setRGB***: nos permite cambiar el color de fondo de la pantalla. Lo utilizaremos para cambiar el color dependiendo de la temperatura actual.
- ***setCommand****:* nos permite realizar acciones sobre la pantalla, como por ejemplo resetearla.
- ***setText:*** con esta función podemos escribir texto en la pantalla. En nuestro caso, mostrar la temperatura y la humedad que indique el sensor.

Conexión con el servidor

En esta primera versión de la aplicación, tanto la RP como el servidor estarán en la misma red.

Para poder enviar los datos al servidor, utilizaremos la librería *socket.io-client*, la cual nos permitirá establecer una conexión mediante WebSockets.

```
// Connect to server using websockets at port 3001
var io = require('socket.io-client');
var socket = io.connect('http://192.168.1.18:3001', {reconnect: true});
```

Una vez se ha conectado al servidor, se crea una instancia del sensor y se ejecutarán las siguientes acciones hasta que el cliente decida desconectarse:

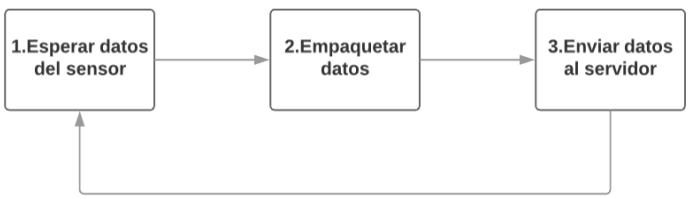


Figura 7. Bucle de acciones durante la ejecución del cliente

Cada vez que el sensor envíe un mensaje, lo captaremos y extraeremos lo que nos interesa: las mediciones de temperatura y humedad. Estos datos, los pondremos en formato JSON junto con otros metadatos y finalmente serán enviados a través del socket abierto.

```
var timestamp = Date.now();
var data = JSON.stringify({        ///put data into a JSON format
  deviceId: 'myFirstNodeDevice',
  timestamp: timestamp,
  temperature: temp,
  humidity: hum
});

socket.emit('msg', data);
```

Además, en función de la temperatura obtenida, se encenderá el LED y se pondrá de cierto color el fondo de la pantalla LCD para que resulte más visual la información obtenida.

Servidor y base de datos

Esta parte del sistema es la responsable de captar los datos que envía la Raspberry Pi, guardarlos en la base de datos (MongoDB), mostrarlos por pantalla y si hay algún cliente conectado se les envía a través de una conexión HTTP y a través de una conexión con WebSockets. Toda esta parte de la aplicación estará programada en Node.js.

Algunas de las partes más importantes de su implementación son las siguientes:

Inicialización del servidor

Al inicializar el servidor, se llevan a cabo dos funciones principales.

Primeramente, se abren los dos puertos (3000 y 3001) a través de los cuales estará escuchando posibles conexiones que más adelante hará el cliente Angular. A través del puerto 3000 se realizará una conexión mediante una API Rest y a través del puerto 3001 (cuya apertura se realiza en el archivo “socket.js”) la conexión será mediante el uso de sockets. El servidor se mantendrá a la espera hasta que alguien intente conectarse por estos puertos.

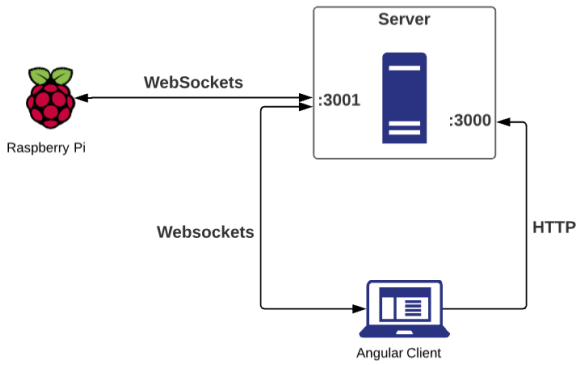


Figura 8. Esquema de protocolos y puertos

Por otro lado y de forma paralela, el servidor también escuchará a través del puerto 3001 posibles conexiones que pueda llevar a cabo la RP.

```
var port = process.env.PORT || 3000;

// Start the server
app.listen(port);
console.log('');
console.log('*****SERVER LAUNCHED*****');
console.log('Rest server listening on port ' + port);
console.log('');
```

```
var port = process.env.PORT_SOCKET || 3001;

// Open WebSocket at port 3001 to communicate with frontEnd and RaspberryPi
http.listen(port, function () {
  console.log('');
  console.log('Socket server listening on port ' + port);
  console.log('');
});
```

Para crear nuestra aplicación, utilizaremos el framework *Express* y para interactuar con la base de datos utilizaremos *Mongoose*, una herramienta de modelado de objetos para bases de datos MongoDB. Además, crearemos las rutas necesarias para acceder al servidor y a los datos de la base de datos. Dichas rutas serán *'/ ',* como raíz del servidor y '***api/readings***' como ruta donde se mostrarán las mediciones obtenidas de la RP.

Una vez se nos conecte la RP, estaremos a la espera de la llegada de mensajes. Cuando nos envíe un mensaje, este será guardado en la base de datos y enviado hacia el cliente Angular.

```
io.on('connection', function (socket){
    console.log('connection');

    socket.on('msg', handleMessage);
});
```

```
function handleMessage(message){

  jsObject = JSON.parse(message);

  saveReading(jsObject);

  emitReading(message);

}
```

Cliente Angular

Este cliente, a partir de los datos obtenidos del servidor devuelve unas gráficas representativas de la temperatura y la humedad obtenidas por el sensor. Para ello, se deberá conectar al servidor mediante una conexión HTTP por la que podrá recuperar información de la base de datos y otra conexión mediante el uso de sockets que permitirá obtener la información que envíe el sensor en tiempo real. Para que la representación de los datos sea más visual, se ha utilizado la librería blur-admin que ofrece unas plantillas para la representación de datos.

Debido a la incorporación de esta plantilla, se añadirán automáticamente una gran cantidad de archivos de configuración. Sin embargo, la lógica de la aplicación la encontramos en tres archivos: un servicio (*dashboard.service.ts*), un componente (*dashboard.component.ts*) y la plantilla html donde se representarán los datos (*dashboard.component.html*).

Dentro del servicio, tendremos la función *getChartConfiguration* que nos dará el aspecto visual de las gráficas.

Para recuperar los datos del servidor, tendremos dos funciones:

- *getLastHour*: realizará una petición *get* a la ruta /api/readings, recopilando así todas las lecturas del sensor guardadas en la base de datos durante la última hora.
- *getMessages*: crea y devuelve un observable que espera para recibir los mensajes.

Dado que tanto los datos que llegan en tiempo real como los guardados en la base de datos están en formato JSON, necesitaremos parsearlos para convertir los valores de temperatura y humedad a *int* para que las gráficas puedan interpretarlos y representarlos. Para ello se utilizarán las funciones *parseDataArray* y *parseDataElement*.

Estas funciones serán utilizadas por el componente, el cual tiene una función para cada gráfica:

- Gráfica estática

```
initRestChart(chart: any) {
  this.dashboardService.getLastHour().then(sensors => {
    chart.dataProvider = this.dashboardService.parseDataArray(sensors);
    chart.validateData();
  });
}
```

- Gráfica dinámica

```
initSocketChart(chart: any) {
  this.connection = this.dashboardService.getMessages().subscribe(message => {
    chart.dataProvider.push(this.dashboardService.parseDataElement(message));
    chart.validateData();
  });
}
```

Finalmente, se insertarán las gráficas en la plantilla HTML del componente:

```
<div class="row">
  <ba-card class="col-xlg-12 col-xl-12 col-lg-12 col-md-12 col-sm-12 col-xs-12" title="Temperature/Humidity (real time)" baCardClass="medium-card">
    <ba-am-chart baAmChartClass="dashboard-line-chart" [baAmChartConfiguration]="chartDataRest" (onChartReady)="initSocketChart($event)"></ba-am-chart>
  </ba-card>
</div>

<div class="row">
  <ba-card class="col-xlg-12 col-xl-12 col-lg-12 col-md-12 col-sm-12 col-xs-12" title="Temperature/Humidity (last hour)" baCardClass="medium-card">
    <ba-am-chart baAmChartClass="dashboard-line-chart" [baAmChartConfiguration]="chartDataSocket" (onChartReady)="initRestChart($event)"></ba-am-chart>
  </ba-card>
</div>
```

Dando como resultado:

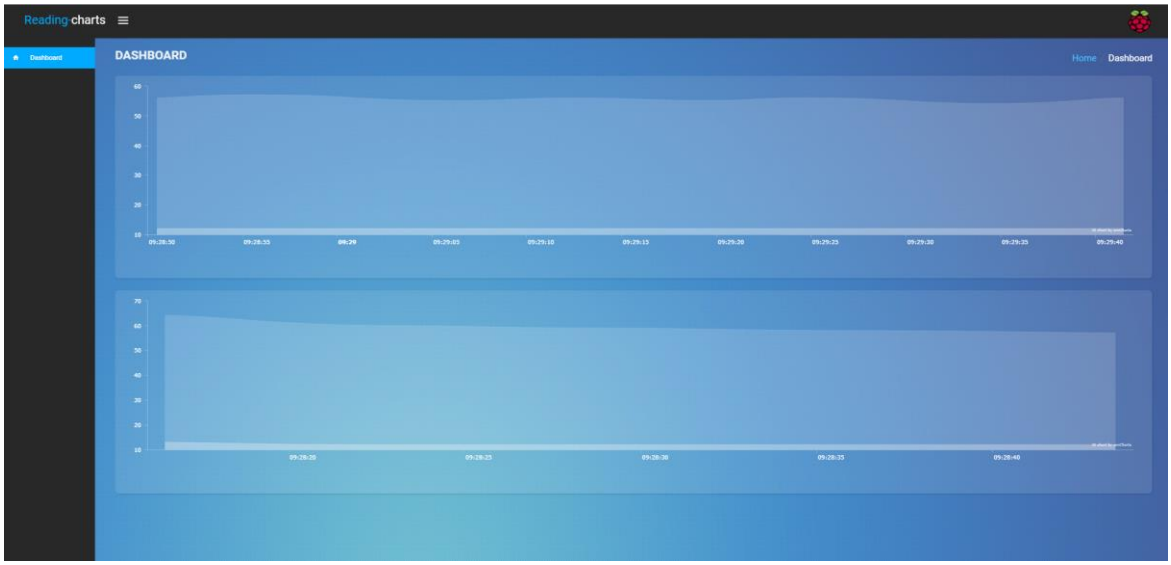


Figura 9. Cliente Angular en funcionamiento

Llegar a este resultado no fue sencillo, ya que al realizar la petición HTTP al servidor para obtener los datos, estos vienen como String en formato JSON. Al pasárselos a las gráficas, estas no daban ningún resultado y permanecían vacías, debido a que esperaban valores Integer y no Strings. Una vez descubierto el problema, se crearon las funciones mencionadas anteriormente para parsear los datos y darles el formato correcto para poder representarlos gráficamente.

3.2. Segunda versión de la aplicación

El escenario sobre el que se ha desarrollado la segunda versión de la aplicación es como el de la siguiente figura:

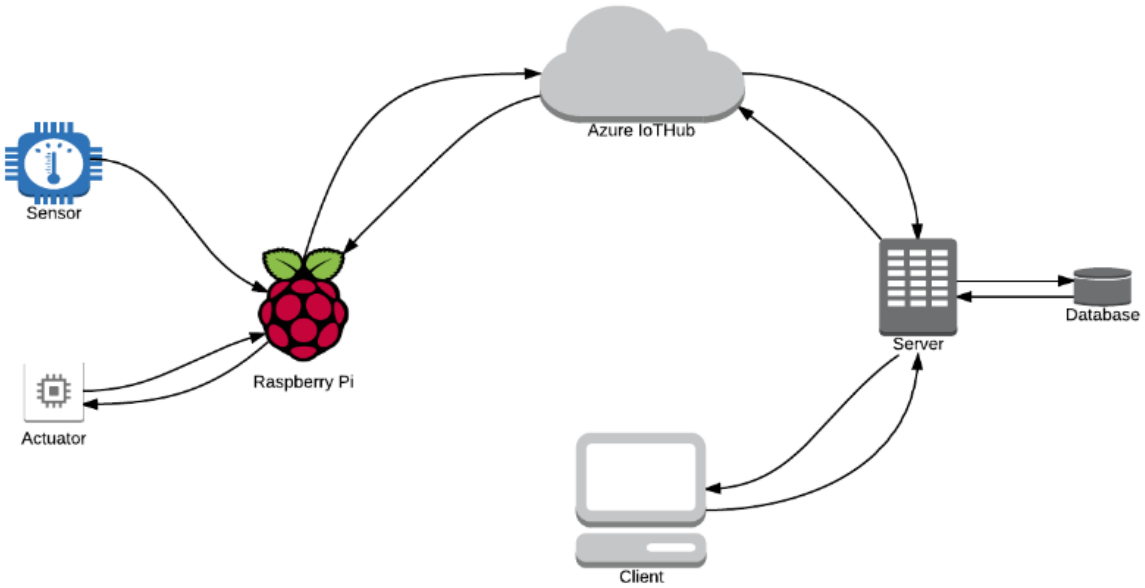


Figura 10. Arquitectura segunda versión de la aplicación

Esta arquitectura tiene cuatro partes principales:

1. Raspberry Pi y componentes
2. Servidor y base de datos
3. Cliente Angular
4. Azure IoT Hub

Los tres primeros elementos son exactamente los mismos que en la primera versión de la aplicación. En cuanto a la implementación del código que corre en cada elemento, la parte del cliente Angular se mantiene sin modificaciones, mientras que en la RP como en el servidor se han llevado a cabo ciertos cambios que se detallarán en el apartado de *Desarrollo*.

Por otra parte, aparece un nuevo elemento en esta arquitectura: el IoT Hub.

Azure IoT Hub

Azure IoT Hub es una Plataforma como Servicio (PaaS, *Platform as a Service*) que permite realizar comunicaciones seguras y confiables de manera bidireccional entre millones de dispositivos IoT y una solución en la nube. Con IoT Hub se pueden abordar retos de implementación como los siguientes:

- Conectividad y gestión de un gran volumen de dispositivos.
- Gran volumen de ingestión de datos.
- Comando y control de dispositivos.
- Refuerzo de la seguridad de dispositivos.

En este proyecto, se utilizarán varias librerías para comunicar el dispositivo con nuestra solución backend. Habrá dos partes diferenciadas, la comunicación dispositivo-a-nube y la comunicación nube-a-dispositivo.

Comunicación dispositivo-a-nube

Esta es la parte encargada de enviar los datos recogidos por el sensor hacia el servidor. En el elemento emisor de datos (nuestra RP), se utilizará el SDK para dispositivos gracias al cual se iniciará un cliente que será capaz de mandar datos al IoT Hub y recibirlos. Estos datos viajarán mediante el protocolo MQTT v3.1.1, cifrados bajo TLS.

Una vez dentro de Azure, los mensajes llegan a un *device-endpoint* o punto de acceso de dispositivos, ubicado en */devices/{deviceId}/messages/events*. Una vez allí, unas reglas de enrutamiento se encargarán de revisar las cabeceras de los mensajes para saber hacia dónde enrutarlos. Por defecto, los mensajes son enrutados hacia el *service-endpoint messages/events*, que es un punto de acceso de servicios compatible con EventHubs. Todos los mensajes que pasen por *messages/events* serán almacenados durante un día (valor por defecto) o hasta un máximo de siete días. Además, la longitud de los mensajes no puede exceder de los 256 KB, pudiéndolos agrupar en lotes para optimizar los envíos. Cada lote puede ser de hasta 256 KB.

Por último, este *endpoint* enviará los datos al backend, el cual tiene implementado un cliente del servicio EventHub dedicado a la ingesta de datos. Estos datos viajarán mediante el protocolo AMQP 1.0, también cifrados bajo TLS.

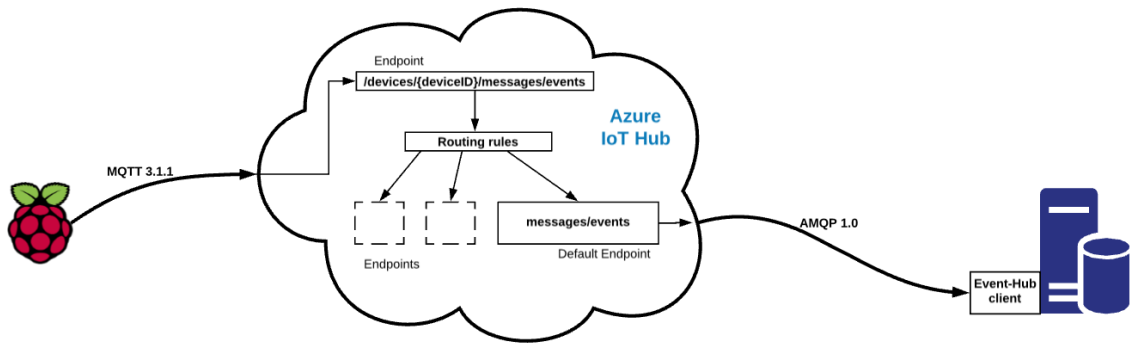


Figura 11. Funcionamiento interno Azure IoT Hub

Comunicación nube-a-dispositivo

Esta es la parte encargada de enviar datos del servidor al dispositivo. Para ello en el servidor se utiliza la librería 'azure-iothub', con la cual se pueden enviar mensajes hacia el IoT Hub.

Una vez dentro de Azure, los mensajes se envían a través del *service-endpoint /messages/devicebound*. Entonces, nuestro dispositivo podrá recibir los mensajes a través del *device-endpoint /devices/{deviceId}/messages/devicebound*.

Cuando los datos salgan de Azure, se dirigirán hacia el dispositivo mediante el protocolo MQTT v3.1.1 sobre TLS.

3.2.1. Desarrollo

En este apartado se explicará brevemente la implementación[5] de cada una de las partes de esta versión.

Cliente Raspberry Pi

Para poder mandar los datos hacia el servidor, en esta versión será necesario que los mensajes pasen previamente por el IoT Hub. Las modificaciones que se han realizado son las siguientes:

Librerías y variables de entorno

A continuación veremos las librerías y las variables necesarias para la conexión de la RP con el IoT Hub.

'**azure-iot-device-mqtt**': librería necesaria para el envío de mensajes hacia el IoT Hub utilizando el protocolo MQTT.

'**azure-iot-device**': librería que contiene los componentes principales del SDK para dispositivos de Azure IoT. Utilizaremos el componente Message para mandar mensajes al IoTHub y también la utilizaremos para establecer el protocolo de comunicación entre dispositivo y Azure.

connectionString: son las claves necesarias para la autenticación del dispositivo con el IoT Hub. Las obtenemos en la plataforma de Azure al crear el servicio.

client: instanciamos un objeto de tipo cliente que será el encargado de comunicarse con el IoT Hub.

```
// Set Azure IoT variables
var clientFromConnectionString = require('azure-iot-device-mqtt').clientFromConnectionString;
var Message = require('azure-iot-device').Message;
var connectionString = 'HostName=****.azure-devices.net;DeviceId=****;SharedAccessKey=****';
var client = clientFromConnectionString(connectionString);
```

[5] Para ver el código completo de cada parte, ver anexos: 8.5 Código Raspberry Pi, 8.6 Código servidor, 8.7 Códgio Cliente Angular.

Envío de mensajes

Para enviar los datos obtenidos del sensor, se insertan los valores de temperatura y humedad en un objeto JavaScript y este se pasa a un string JSON, tal como se hizo en la primera versión. Además, en esta segunda versión, se meterán los datos en un objeto de tipo *Message* y se enviará hacia el IoT Hub mediante el uso de la función *sendEvent.*

```
var timestamp = Date.now();
var data = JSON.stringify({
  deviceId: 'myFirstNodeDevice',
  timestamp: timestamp,
  temperature: temp,
  humidity: hum
});
var message = new Message(data);

// Send message to Azure IoT Hub
client.sendEvent(message);
```

Recepción de mensajes

La RP, a parte de enviar los datos sobre las mediciones recogidas por el sensor, también puede recibir datos u órdenes. En nuestro caso, recibirá una respuesta por cada mensaje enviado al servidor. Esta respuesta llevará la información necesaria para encender el led en función de la temperatura recogida.

```
client.on('message', function(msg) {
  if (msg.data == 'on'){
    led.turnOn();
    status = 1;
  }
  else {
    led.turnOff();
    status = 0;
  }
});
```

Servidor y base de datos

Librerías y variables de entorno

En esta versión de la aplicación, se añadirán una serie de librerías que se utilizarán para la comunicación del servidor con el IoT Hub. A la hora de interactuar con la base de datos y el cliente de Angular el procedimiento será exactamente el mismo que en la primera versión.

Las librerías utilizadas son las siguientes:

'**azure-iothub**': proporciona las funcionalidades necesarias para poder enviar datos del servidor a Azure.

'**azure-iot-common**': de esta librería se aprovecha la clase Message, con la cual podremos mandar comandos o mensajes de telemetría entre un elemento y el servicio IoT Hub.

'**azure-event-hubs**': se trata de la librería que se encarga de toda la ingesta de datos que provengan de Azure.

```
var EventHubClient = require('azure-event-hubs').Client;
var iothub = require('azure-iothub').Client;
var Message = require('azure-iot-common').Message;
var Reading = require('../models/reading');

// Declare variables
var port = process.env.PORT_SOCKET || 3001;
var connectionString = process.env.CONNECTION_STRING || 'HostName=****.azure-devices.net;SharedAccessKeyName=****;SharedAccessKey=****';
var targetDevice = 'myFirstNodeDevice';
```

Conexión con IoT Hub

Una vez configurados los parámetros y las claves para la conexión con el IoT Hub, se procederá a abrir dos clientes: *client* y *serviceClient*. Esto es así, porque no es posible mandar y recibir mensajes desde un mismo cliente. El cliente *client* se crea a partir de la librería 'azure-event-hubs', por lo tanto será el encargado de toda la ingesta de datos que provengan de Azure. Por otro lado, el cliente *serviceClient*, creado a partir de la librería 'azure-iothub', será el encargado de enviar mensajes a Azure.

Gestión de mensajes

Por cada mensaje que llegue al servidor, se guardará en la base de datos, se enviará al cliente Angular conectado si lo hubiera y por último se enviará una respuesta a la RP con la temperatura obtenida para que esta pueda encender o no el LED.

Cliente Angular

En esta segunda versión de la aplicación el cliente Angular no ha sido modificado respecto a la primera versión.

4. Resultados

A continuación, se mostrará el funcionamiento de la aplicación en cada una de las partes:

```
pi@raspberrypi:~/Desktop/hub-raspberry-client $ node TemperatureHumidityDeviceDisplayRound2.js
Client connected
Starting GrovePi board initialization
info GrovePi.board GrovePi is initing
========================================

TEMP = 18.39 °C        HUM = 34 %

Sending message: {"deviceId":"myFirstNodeDevice","timestamp":1516623847758,"temperature":18,"humidity":34}
========================================
^Cending
info GrovePi.board GrovePi is closing
pi@raspberrypi:~/Desktop/hub-raspberry-client $
```

Figura 12. Envío de mensajes con la RP.

```
C:\Users\jcamacht\Desktop\codigoFinal\node-backend>node server.js

*****SERVER LAUNCHED*****
Rest server listening on port 3000


Socket server listening on port 3001


Ready to SEND messages to devices


=========================
== CONNECTED TO AZURE ==
=========================
Ready to RECEIVE data from devices

Created partition receiver: 0
Created partition receiver: 1

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Message received:
{"deviceId":"myFirstNodeDevice","timestamp":1516623847758,"temperature":18,"humidity":34}

Message sent to WebSocket

Temperature:  18
Sending message to Device: on

Reading added to the database
^C
C:\Users\jcamacht\Desktop\codigoFinal\node-backend>
```

Figura 13. Recepción de mensajes en el servidor.

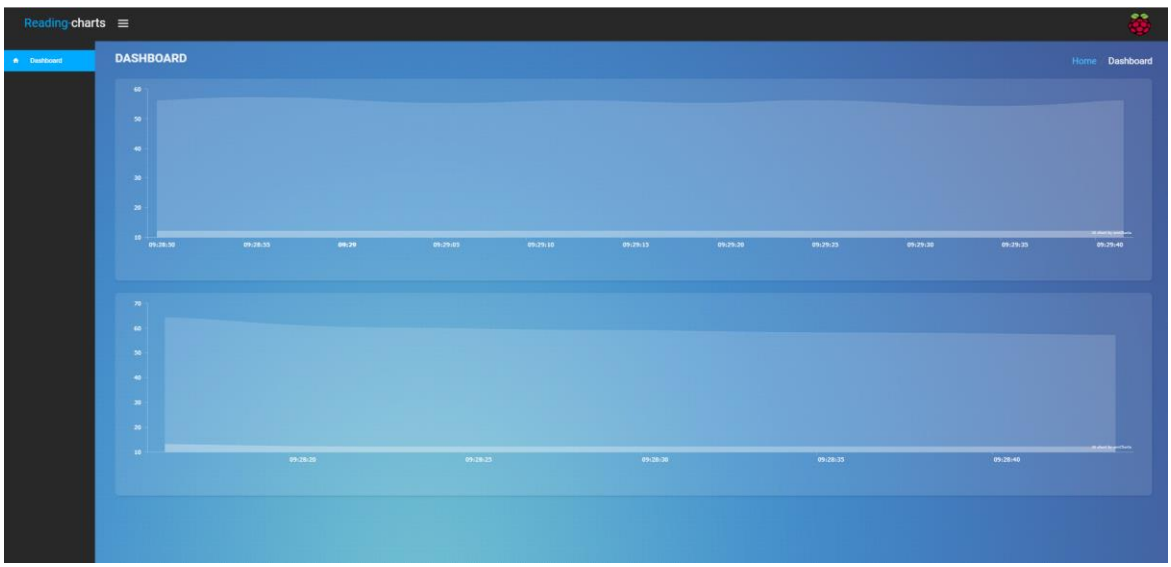


Figura 14. Representación de datos en el cliente Angular

5. Presupuesto

El número de horas invertidas en este proyecto han sido aproximadamente las 720h estipuladas para la realización de un TFG en empresa. Teniendo en cuenta que la remuneración se ha pagado a 8 €/hora, el coste del personal asciende a 5760 €.

Para el desarrollo de la aplicación, se han comprado diversos elementos hardware cuyo coste se muestra a continuación:

- Portátil de empresa (Dell Latitude E5540): 862 €
- Raspberry Pi 3: 40 €
- Placa Grovepi+: 30 €
- Led Grovepi: 2 €
- Pantalla LCD (LCD RGH Backlight): 12 €
- Sensor Temperatura/Humedad (DHT11): 6€

Por lo tanto, el coste total del proyecto sería de:

$\mathbf{C_{TOTAL}} = C_{personal} + C_{Hardware} = 5760\ € + 952\ € = \mathbf{6712\ €}$

Una vez desarrollado el prototipo, si se quisieran desplegar diferentes unidades para una monitorización en varios puntos, cada unidad tendría un coste de **90 €**.

6. Conclusiones y futuro desarrollo:

Este proyecto me ha permitido adquirir una base de conocimientos sobre el IoT, gracias a que se han tratado los objetivos propuestos inicialmente. Unos objetivos que por un lado, tratan de mostrar las dificultades que representa el desarrollo de aplicaciones IoT y por otro, trata de mostrar una manera simple de desarrollar una aplicación utilizando una solución tecnológica específica para este tipo de tecnologías de manera sencilla y segura.

Como ya hemos visto, la aplicación que se ha creado en este proyecto es un prototipo que monitoriza la temperatura y la humedad a través de un sensor. En un futuro se pretende utilizar este prototipo para monitorizar también la calidad del aire, la cantidad de ruido, cantidad de luz, entre otras magnitudes, de la oficina LivingLab de Everis. Aunque el precio del proyecto es elevado, una vez diseñado, el precio de cada módulo es de 70€ sin contar los sensores, los cuales se pueden añadir por un precio de alrededor de 6€ cada uno, pudiendo tener un módulo de monitorización con muchos sensores por una cantidad relativamente pequeña de dinero.

No hay que olvidar que no se trata de un simple sensor que envía datos, sino que por debajo tiene una Raspberry Pi, con lo cual tenemos varios sensores conectados a un pequeño ordenador con el cual se pueden llevar a cabo infinidad de aplicaciones. El límite estará en nuestra imaginación.

7. Bibliografía:

[1] Henryk Konsek, “The Architecture of IoT Gateways”. Disponible en: https://dzone.com/articles/iot-gateways-and-architecture.

[2] Andreas Nettsäter, “Internet-of-Things Architecture”. Disponible en: http://www.meet-iot.eu/deliverables-IOTA/D1_3.pdf.

[3] Pubnub, “A new approach to IoT Security”. Disponible en: https://www.pubnub.com/static/papers/IoT_Security_Whitepaper_Final.pdf.

[4] Microsoft, “Azure IoT Hub developer guide”. Disponible en: https://www.pubnub.com/static/papers/IoT_Security_Whitepaper_Final.pdf .

[5] R. Roman, J. Zhou, and J. Lopez, “On the features and challenges of security and privacy in distributed internet of things”, Computer Networks, vol.57, pp. 22662279, 2013. Disponible en: https://www.nics.uma.es/pub/papers/roman2013iot.pdf

[6] Glow Labs, “Table Comparing Wireless Protocols for IoT Devices”. Disponible en: http://glowlabs.co/wireless-protocols/.

8. Anexos:

8.1. Tabla con ejemplos de protocolos de nivel de enlace

Protocolos IoT	Frec. (EU)	Alcance	Consumo	Seguridad
ZigBee	2.4GHz, 868 MHz (EU)	30-100 m	bajo	Cifrado
Z-Wave	868MHz (EU)	30-100 m	bajo	Cifrado
Bluetooth 4.0+	2.4GHz	60 m	medio	Cifrado
Bluetooth 5	2.4GHz	250 m	medio	Cifrado
Bluetooth Low Energy (BLE)	2.4GHz	60 m	bajo	Cifrado
Wi-Fi	2.4GHZ/5GHz	35-70 m	alto	Opcional
Wi-Fi-ah (HaLow)	900MHz	900 m	bajo	???
Thread	2.4GHz	30 m	bajo	Cifrado
DigiMesh	2.4GHz, 868 MHz (EU)	~32 km	bajo	Cifrado
MiWi	2.4GHz	250 m	bajo	Cifrado
EnOcean	868 MHz (EU) 315 MHz	9-30 m	"Battery Free"	Cifrado
6LoWPAN	2.4GHz	115 m	bajo	Opcional
Weightless (W, N, P)	white-spaces, 915MHz, 868MHz, 780MHz, 470 MHz, 433 MHz, 169 MHz	2 km (P), 5 km (W, N)	bajo (N), medio (W, P)	Cifrado
mcThings	2.4GHz	200 m	bajo	Cifrado
LoRa	150MHz-1GHz (muchas opciones)	hasta 32 km	bajo	Cifrado
SigFox	900Mhz (US) 868 MHz (EU)	~32 km	bajo	Cifrado
LTE Cat-M1	1.4MHz	~32 km	bajo	Espectro bajo licencia
NarrowBand-IoT (Cat M2)	Por debajo de 1GHz	~32 km	bajo	Cifrado
3G and 4G Cellular (US)	700 MHz, 800 MHz, 850MHz, 1700MHz, 1900MHz, 2100MHz, 2300MHz, 2500MHz	~32 km	alto	Cifrado

8.2. Comparativa de seguridad

A la hora de realizar el análisis, se ha considerado que, como el cliente Angular simplemente hace peticiones a un servidor e interpreta los datos, no es un elemento que sea representativo de un escenario IoT real, sino que se podría encontrar en cualquier arquitectura web corriente. Por lo tanto, el cliente Angular se encuentra sin ningún tipo de seguridad incorporada en ambas versiones.

En cambio, la parte que incluye el sensor, el Gateway (RP), el servidor y la plataforma *cloud* (Azure), al ser un posible escenario IoT real se ha decidido realizar aquí la evaluación de la seguridad.

Evaluación de la seguridad de la aplicación v1.0

En esta primera versión de la aplicación, no se ha implementado absolutamente ninguna medida de seguridad. Este escenario puede parecer exagerado hasta cierto punto, ya que contiene varios fallos triviales que un desarrollador experimentado no debería hacer. Quizá sea difícil encontrar un escenario tan inseguro en tantos aspectos, pero lo cierto es que es más común de lo que parece el hecho de encontrarnos con una o varias de las vulnerabilidades que se explicarán a continuación en sistemas desarrollados y ya desplegados por todo el mundo.

Cifrado del payload

Al arrancar la aplicación de la RP, se establece la conexión con el servidor y se empiezan a mandar los datos. La transmisión se realiza utilizando el protocolo WebSockets, el cual manda la información sin cifrar. Esto es muy peligroso, pues se pueden llevar a cabo ataques MITM (Man In The Middle) esnifando el tráfico en un punto intermedio de la comunicación y viendo el contenido de las comunicaciones. Para corroborarlo, se ha hecho la prueba utilizando Wireshark y efectivamente, podemos ver en la siguiente figura como podemos acceder a la información enviada hacia el servidor.

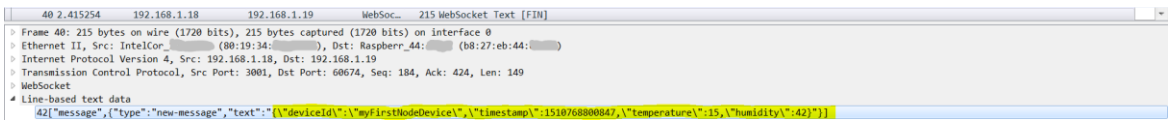


Figura 15. Mensaje captado con Wireshark sin cifrar

Autenticación del dispositivo

Tal como dicta su definición, la autenticación se utiliza para confirmar que una entidad es quien dice ser. Para poder tener autenticación es necesaria, como condición previa, la existencia de identidades biunívocamente identificadas de tal forma que se permita su identificación.

En el caso de esta primera versión de la aplicación, ni el servidor está identificado como tal ni la RP tiene la capacidad de identificarse y asegurar al servidor que es quien dice ser. El problema reside en que en esta arquitectura, la única entidad que debería tener acceso al servidor es la RP, pero con esta implementación sin autenticación, cualquier

usuario con acceso a la red en la que se encuentra el servidor puede robar los datos alojados, inyectar nuevos datos o incluso destruirlos.

Credenciales y acceso al dispositivo IoT

Este apartado es uno de los más críticos a los que se enfrenta el IoT, además de muchos otros sistemas en general. Uno de los errores más frecuentes a la hora de utilizar y/o administrar un dispositivo, es no revisar la configuración y dejar los valores por defecto.

En nuestro caso, donde tenemos una RP, el problema es evidente ya que podemos aprovecharnos de esta "vulnerabilidad". Las Raspberry Pi tienen en todos sus modelos como credenciales por defecto el usuario "pi" y la contraseña "raspberry". Esto ha sido utilizado por diferentes atacantes y cierto malware, como el llamado "Linux.MulDrop.14", el cual se aprovechaba de estas credenciales por defecto para entrar a las RP y utilizarlas para minar criptomonedas.

Tanto para los afectados por el malware anteriormente citado como para nuestro caso, la solución es simplemente cambiar el usuario y poner una contraseña más robusta.

Debido a la naturaleza de los dispositivos IoT, estos pueden estar repartidos por cualquier lugar, hasta en los puntos más inaccesibles, lo cual implica que quizá solo se podrá acceder a ellos remotamente.

El problema es que si se quiere administrar el dispositivo, se suele dejar un puerto abierto escuchando posibles peticiones, normalmente el puerto 22 (SSH), el cual es uno de los más buscados y atacados.

Una de las formas de poder proteger nuestra RP de posibles atacantes es mediante el uso del golpeo de puertos (*Port Knocking)*. Este mecanismo nos permite mantener el puerto 22 cerrado, evitando así posibles intentos de conexión indeseados. Los usuarios legítimos, deberán "golpear" o enviar un mensaje a ciertos puertos en un orden concreto, abriendo así el puerto deseado, en este caso el que contenga el servicio SSH.

Control de acceso

Otra de las configuraciones poco seguras que ofrece esta primera versión, al igual que muchas en Internet, es no tener un control de acceso robusto frente a quién puede acceder a nuestros recursos o los métodos HTTP que pueden usar en nuestra aplicación. En este caso, se permite el acceso de cualquier dominio a nuestros recursos, pudiendo ejecutar cualquiera de las peticiones HTTP (GET, POST, OPTIONS, PUT, PATCH, DELETE) dejando una configuración poco segura.

Evaluación de la seguridad de la aplicación v2.0

En esta segunda versión, se ha utilizado Azure para comunicar la RP con el servidor. Este hecho ha provocado que el sistema esté sujeto a la seguridad que implementa este servicio.

Cifrado del payload

Si volvemos a capturar el tráfico, veremos como ahora el mensaje viaja cifrado. Esto se debe a que Azure IoT soporta TLS 1.2, TLS 1.1 y TLS 1.0, en este orden. En nuestro caso, el mensaje irá cifrado mediante TLS 1.0, tal como se muestra en la siguiente imagen.

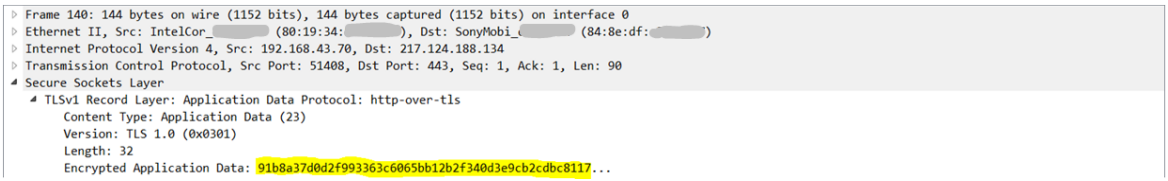


Figura 16. Mensaje cifrado captado con Wireshark

Autenticación del dispositivo

En la versión anterior no teníamos ningún sistema de autenticación. Con la incorporación del servicio IoT Hub se pueden utilizar claves de autenticación, las cuales permiten autenticar cada dispositivo con el IoT Hub añadiendo las claves en el código.

Además, para los dispositivos que tengan Windows 10 IoT Core, se puede configurar un módulo TPM (*Trusted Platform Module*) con el cual se puede proveer un almacenamiento seguro de claves y generación de tokens. De esta manera, las claves no estarán escritas en el código, sino que estarán guardadas en un servidor de forma más segura. Además, los tokens generados tienen un periodo de validez corto, reduciendo así el impacto en seguridad si alguno de ellos es filtrado.

Control de acceso

En esta versión de la aplicación, se ha utilizado la funcionalidad que nos ofrece CORS (Cross-Origin-Resource-Sharing). De este modo, podemos aceptar solamente peticiones que provengan de la dirección que queramos. En nuestro caso, de la misma máquina desde donde el cliente lanzará sus peticiones. Además, permitiremos únicamente peticiones del tipo 'GET'.

```
// Add headers
app.use(function(req,res,next){
  // Website you wish to allow to connect
  res.setHeader('Access-Control-Allow-Origin', 'http://localhost:4200');

 // Request methods you wish to allow
 res.setHeader('Access-Control-Allow-Methods', 'GET');

 next();
});
```

8.3. Elementos de la arquitectura

Raspberry Pi

La Raspberry Pi es un computador de placa reducida desarrollado en Reino Unido con el objetivo de estimular la enseñanza de ciencias de la computación en las escuelas.

En este caso se trata de una Raspberry Pi 3 Modelo B, la cual posee un chip fabricado por Broadcom (BCM2837), con una CPU quad-core ARM Cortex-A53 a 1,2 GHz y con 1GB de RAM. Incorpora conectividad Wifi y Bluetooth (4.1 Low Energy). También incluye 40 pines GPIO, 4 puertos USB, conector de 3.5mm, Ethernet, microHDMI y un slot para microSD.

El sistema operativo que se le ha instalado es Raspbian, el cual está basado en Debian.

Placa GrovePi+

GrovePi+ es un sistema modular construido pensando en la facilidad de uso, con el cual se pueden añadir a la Raspberry Pi diferentes módulos sin necesidad de soldar ninguna placa.

Este sistema se agrega a la Raspberry Pi utilizando los 40 pines GPIO y ya estará listo para que la RP y los módulos se puedan comunicar entre sí.

DHT11: Sensor de temperatura y humedad

El sensor utilizado para medir la temperatura y la humedad desde la Raspberry Pi es el DHT11. Las características principales son:

- Rango de medida
  - Temperatura: 0 ~ 50 ºC
  - Humedad: 20% - 90% RH
- Exactitud
  - Temperatura: ±2 ºC
  - Humedad: ±5 % RH
- Sensibilidad
  - Temperatura: 1 ºC
  - Humedad: ±1 % RH

Led: Grove Led socket kit

Led con conexión para la placa Grovepi+ con potenciómetro incorporado para controlar la luminosidad.

Grove LCD RGB Backlight: Pantalla LCD

Con el fin de poder ver los resultados enviados por el sensor, dispondremos de una pantalla LCD que hará de actuador y la cual podremos programar para que muestre dichos datos, otro tipo de texto e incluso cambiar el color de fondo dependiendo de la temperatura que envíe el sensor.

Servidor y base de datos

Como backend de esta aplicación, se utilizará un servidor programado en Node.js. Este servidor estará alojado en una máquina personal.

La base de datos que utilizará el servidor para guardar los datos obtenidos será MongoDB, una base de datos NoSQL muy utilizada en sistemas IoT.

Cliente

El cliente de esta aplicación está programado en Angular 2 y se ejecutará en una máquina personal. El objetivo de este será representar de forma gráfica los datos obtenidos del sensor conectado a la Raspberry Pi.

8.4. WebSocket vs HTTP: ¿Por qué WebSocket?

Para la comunicación entre la RP y el servidor en la primera versión de la aplicación, se pensaron dos posibles maneras de llevarlo a cabo: mediante HTTP o WebSockets.

El protocolo HTTP se rige por un paradigma de tipo petición/respuesta. Cada uno de estos mensajes tiene una cabecera que puede llegar a ser de algunos KB. En cambio, WebSockets realiza un *handshake* mediante HTTP para dejar una conexión bidireccional abierta, permitiendo así el intercambio de mensajes sin apenas utilizar bytes para cabeceras.

En esta aplicación, debido a la baja velocidad a la que el sensor envía los datos, era prácticamente indiferente el uso de una u otra tecnología, pero dado que se están estudiando aplicaciones IoT se ha preferido utilizar WebSockets por las ventajas que ello conlleva en aplicaciones de este tipo.

Un posible escenario real podría ser un sensor que envíe datos sobre la temperatura de una máquina industrial. Este tipo de sensores pueden enviar miles de mediciones por segundo, con lo cual el uso de peticiones HTTP sería muy ineficiente para la red. Por este motivo, se ha decidido utilizar WebSockets.

8.5. Código Raspberry Pi

```
'use strict';

// GrovePi variables initialization
var i2c = require('i2c-bus');
var GrovePi = require('node-grovepi').GrovePi;
var Commands = GrovePi.commands;
var Board = GrovePi.board;
var DHTDigitalSensor = GrovePi.sensors.DHTDigital;
var sleep = require('sleep');

var DISPLAY_RGB_ADDR = 0x62;
var DISPLAY_TEXT_ADDR = 0x3e;

// Set Grovepi variables
var board;
var led = new GrovePi.sensors.DigitalOutput(3);
var status = 0;

// Set Azure IoT variables
var clientFromConnectionString = require('azure-iot-device-mqtt').clientFromConnectionString;
var Message = require('azure-iot-device').Message;
var connectionString = 'HostName=XXXX.azure-devices.net;DeviceId=XXXX;SharedAccessKey=XXXX=';
var client = clientFromConnectionString(connectionString);

// RGB LCD screen functions
function setRGB(i2c1, r, g, b) {
  i2c1.writeByteSync(DISPLAY_RGB_ADDR,0,0);
  i2c1.writeByteSync(DISPLAY_RGB_ADDR,1,0);
  i2c1.writeByteSync(DISPLAY_RGB_ADDR,0x08,0xaa);
  i2c1.writeByteSync(DISPLAY_RGB_ADDR,4,r);
  i2c1.writeByteSync(DISPLAY_RGB_ADDR,3,g);
  i2c1.writeByteSync(DISPLAY_RGB_ADDR,2,b);
}

function textCommand(i2c1, cmd) {
  i2c1.writeByteSync(DISPLAY_TEXT_ADDR, 0x80, cmd);
}

function setText(i2c1, text) {
  textCommand(i2c1, 0x01); //Clear display
  sleep.usleep(50000);
  textCommand(i2c1, 0x08 | 0x04); //display on, no cursor
  textCommand(i2c1, 0x28); //2 lines
  sleep.usleep(50000);
  var count = 0;
  var row = 0;
  for (var i = 0, len = text.length; i < len; i++) {
    if(text[i] === '\n' || count === 16) {
      count = 0;
      row ++;
      if (row === 2)
        break;
      textCommand(i2c1, 0xc0);
      if(text[i] === '\n')
        continue;
    }
    count++;
    i2c1.writeByteSync(DISPLAY_TEXT_ADDR, 0x40, text[i].charCodeAt(0));
  }
}
```

```
// Handle request from Azure
var connectCallback = function (err) {
  if (err) {
    console.log('Could not connect: ' + err);
  } else {
    console.log('Client connected');

    client.on('message', function(msg) {
      if (msg.data == 'on'){
        led.turnOn();
        status = 1;
      }
      else {
        led.turnOff();
        status = 0;
      }
    });

    console.log('Starting GrovePi board initialization');

    board = new Board({
      debug: true,
      onError: function(err) {
        console.log('TEST ERROR');
        console.log(err);
      },
      onInit: function(res) {
        if(res) {
          var dhtSensor = new DHTDigitalSensor(7, DHTDigitalSensor.VERSION.DHT11, DHTDigitalSensor.CELSIUS);

          // Start reading from sensor
          dhtSensor.on('change', function(res) {  //res[0] = heat, res[1] = humidity, res[2] = temperature
            if (res[2] < 60 && res[1] < 100){
              var temp = Math.round(res[2]);
              var hum = Math.round(res[1]);
              console.log('');
              console.log('TEMP = ' + res[2] + ' HUM = ' + res[1]);
              console.log('');
              console.log('----------------------');


              var timestamp = Date.now();
              var data = JSON.stringify({
                deviceId: 'myFirstNodeDevice',
                timestamp: timestamp,
                temperature: temp,
                humidity: hum
              });
              var message = new Message(data);
              console.log('Sending message: ' + message.getData());
              console.log('======================');

              // Send message to Azure IoT Hub
              client.sendEvent(message);

              // Send text to display
              var i2c1 = i2c.openSync(1);
              setText(i2c1, 'Temp: ' + temp + ' C' + '\n' + 'Hum: ' + hum + '%');
```

```
            //Change display color
            if (temp>=30){
              setRGB(i2c1,255,0,0);
            }
            else if (temp<=20){
              setRGB(i2c1,0,0,255);
            }
            else{
              setRGB(i2c1,255,255,0);
            }
            i2c1.closeSync();

            sleep.sleep(5);
          }
        });
        dhtSensor.watch(500); // milliseconds
      }
      else {
        console.log('TEST COULD NOT START');
      }
    }
  });
  board.init();
 }
}

// Close script, conections and shutdown display
function onExit(err){
  console.log('ending');
  var i2c1 = i2c.openSync(1);
  setText(i2c1,'');
  setRGB(i2c1,0,0,0);
  i2c1.closeSync();
  led.turnOff();
  board.close();
  process.removeAllListeners();
  process.exit();
  if (typeof err != 'undefined')
    console.log(err);
}

// Open client and start sending information to Azure IoT Hub
client.open(connectCallback);

process.on('SIGINT', onExit);
```

8.6. Código servidor

server.js

```
// Libraries
var express = require('express');
var bodyParser = require('body-parser');
var mongoose = require('mongoose');

// Variables
var mongo_string = process.env.MONGO_STRING || 'mongodb://localhost/myFirstNodeDevice';
var port = process.env.PORT_REST || 3000;
process.env.NODE_ENV = 'DEV';

// Controllers
var readingController = require('./controllers/reading');

// Utilities
var socket = require('./utilities/socket');

// Connect to MongoDB
mongoose.connect(mongo_string);

// Create our Express application
var app = express();

app.use(bodyParser.json());

// Add headers
app.use(function(req,res,next){
  // Website you wish to allow to connect
  res.setHeader('Access-Control-Allow-Origin', 'http://localhost:4200');

 // Request methods you wish to allow
 res.setHeader('Access-Control-Allow-Methods', 'GET');

 next();
});

// Create our Express router
var router = express.Router();

router.route('/readings')
 .get(readingController.getReadings);

router.get('/', function(req,res){
 res.send("Welcome to the server. You can receive last measures going to /api/readings. ");
})

// Register all our routes with /api
app.use('/api', router);

// Start the server
app.listen(port);
console.log('');
console.log('*****SERVER LAUNCHED*****');
console.log('Rest server listening on port ' + port);
console.log('');
```

socket.js

```
// Import Libraries
var app = require('express')();
var http = require('http').Server(app);
var io = require('socket.io')(http);

var EventHubClient = require('azure-event-hubs').Client;
var iothub = require('azure-iothub').Client;
var Message = require('azure-iot-common').Message;
var Reading = require('../models/reading');

// Declare variables
var port = process.env.PORT_SOCKET || 3001;
var connectionString = process.env.CONNECTION_STRING || 'HostName=XXXX.azure-devices.net;SharedAccessKeyName=XXXX;SharedAccessKey=XXXX';
var targetDevice = 'myFirstNodeDevice';

// Open WebSocket at port to communicate with frontEnd
http.listen(port, function(){
    console.log('');
    console.log('Socket server listening on port ' + port);
    console.log('');
});

// Open Azure IoT Hub connection and listen to it
var client = EventHubClient.fromConnectionString(connectionString);
client.open()
    .then(client.getPartitionIds.bind(client))
    .then(function(partitionIds) {
        console.log('');
        console.log('==========================');
        console.log('== CONNECTED TO AZURE ==');
        console.log('==========================');
        console.log('Ready to RECEIVE data from devices');
        console.log('');
        return partitionIds.map(function (partitionId) {
          return client.createReceiver('$Default', partitionId, { 'startAfterTime' : Date.now()}).then(function(receiver) {
            console.log('Created partition receiver: ' + partitionId);
            receiver.on('errorReceived', handleError);
            receiver.on('message', handleMessage);
          });
        });
    })
    .catch(handleError);

// Open Azure IoT Hub connection and send messages to it
var serviceClient = iothub.fromConnectionString(connectionString);
serviceClient.open(function(err) {
    if(err) {
        console.error('Could not connect: ' + err.message);
    } else {
        console.log('');
        console.log('Ready to SEND messages to devices');
        console.log('');
    }
});

// Functions

function saveReading(message) {
    var reading = new Reading();

    reading.deviceId = message.deviceId;
    reading.timestamp = message.timestamp;
    reading.temperature = message.temperature;
    reading.humidity = message.humidity;

    reading.save(function(err) {
        if(err) console.log('An error has occurred');
        else console.log('Reading added to the database');
    });
}
```

```
function emitReading(message) {
    io.emit('message', { type: 'new-message', text: message });
    console.log('Message sent to WebSocket');
    console.log('');
}

function sendMessasgeToDevice(temperature) {
    var message = new Message("");
    message.ack = 'full';
    message.messageId = "MyMessageId";

    if (temperature > 22) message.data = 'off';
    else if (temperature <= 22) message.data = 'on';

    console.log('Temperature: ', temperature);
    console.log('Sending message to Device: ' + message.getData());
    console.log('');

    //serviceClient.send(targetDevice, message, printResultFor('send'));
    serviceClient.send(targetDevice, message);
}

function handleMessage(message) {
    var jsonString = JSON.stringify(message.body);

    console.log('')
    console.log('********************************')
    console.log('Message received: ');
    console.log(jsonString);
    console.log('');

    saveReading(message.body);

    emitReading(jsonString);

    sendMessasgeToDevice(message.body.temperature);
}

function handleError(err) {
    console.log(err.message);
}
```

8.7. Código cliente Angular

Debido a que la plantilla utilizada para generar las gráficas contiene un gran número de archivos, se mostrarán únicamente los que son importantes para la lógica de la aplicación.

dashboard.component.js

```
import { Component, OnDestroy } from '@angular/core';
import { DashboardService } from './dashboard.service';

import * as moment from 'moment';

import 'style-loader!./dashboard.component.scss';

@Component({
  selector: 'app-dashboard',
  templateUrl: './dashboard.component.html',
})

export class DashboardComponent implements OnDestroy {

  public chartDataRest: Object;
  public chartDataSocket: Object;
  public connection: any;

  constructor(private dashboardService: DashboardService) {
    this.chartDataRest = this.dashboardService.getChartConfiguration([]);
    this.chartDataSocket = this.dashboardService.getChartConfiguration([]);
  }

  ngOnDestroy() {
    this.connection.unsubscribe();
  }

  initRestChart(chart: any) {
    this.dashboardService.getLastHour().then(sensors => {
      chart.dataProvider = this.dashboardService.parseDataArray(sensors);
      chart.validateData();
    });
  }

  initSocketChart(chart: any) {
    this.connection = this.dashboardService.getMessages().subscribe(message => {
      chart.dataProvider.push(this.dashboardService.parseDataElement(message));
      chart.validateData();
    });
  }
}
```

dashboard.service.js

```
import { Injectable } from '@angular/core';
import { Http, Headers } from '@angular/http';
import { BaThemeConfigProvider, colorHelper, layoutPaths } from '../../theme';

import { environment } from '../../../environments/environment';

import { Observable } from 'rxjs/Observable';
import 'rxjs/add/operator/toPromise';
import * as io from 'socket.io-client';

@Injectable()
export class DashboardService {

  private firstDate = new Date();
  private socket;

  constructor(
    private http: Http,
    private _baConfig: BaThemeConfigProvider,
  ) {}

  getChartConfiguration(data) {
    const layoutColors = this._baConfig.get().colors;
    const graphColor = this._baConfig.get().colors.custom.dashboardLineChart;

    return {
      type: 'serial',
      theme: 'blur',
      marginTop: 15,
      marginRight: 15,
      responsive: {
        'enabled': true,
      },
      dataProvider: data,
      categoryField: 'timestamp',
      categoryAxis: {
        minPeriod: 'ss',
        parseDates: true,
        gridAlpha: 0,
        color: layoutColors.defaultText,
        axisColor: layoutColors.defaultText,
        dateFormats: [
          { period: 'fff', format: 'JJ:NN:SS' },
          { period: 'ss', format: 'JJ:NN:SS' },
          { period: 'mm', format: 'JJ:NN' },
          { period: 'hh', format: 'JJ:NN' },
          { period: 'DD', format: 'MMM DD' },
        ],
      },
      valueAxes: [
        {
          minVerticalGap: 50,
          gridAlpha: 0,
          color: layoutColors.defaultText,
          axisColor: layoutColors.defaultText,
        },
      ],
```

```
    graphs: [
      {
        id: 'g0',
        bullet: 'none',
        useLineColorForBulletBorder: true,
        lineColor: colorHelper.hexToRgbA(graphColor, 0.3),
        lineThickness: 1,
        negativeLineColor: layoutColors.danger,
        type: 'smoothedLine',
        valueField: 'temperature',
        fillAlphas: 1,
        fillColorsField: 'lineColor',
      },
      {
        id: 'g1',
        bullet: 'none',
        useLineColorForBulletBorder: true,
        lineColor: colorHelper.hexToRgbA(graphColor, 0.15),
        lineThickness: 1,
        negativeLineColor: layoutColors.danger,
        type: 'smoothedLine',
        valueField: 'humidity',
        fillAlphas: 1,
        fillColorsField: 'lineColor',
      },
    ],

    chartCursor: {
      categoryBalloonDateFormat: 'DD MM YYYY - HH:NN:SS',
      categoryBalloonColor: '#4285F4',
      categoryBalloonAlpha: 0.7,
      cursorAlpha: 0,
      valueLineEnabled: true,
      valueLineBalloonEnabled: true,
      valueLineAlpha: 0.5,
    },
    dataDateFormat: 'DD MM YYYY - HH:NN:SS',
    export: {
      enabled: true,
    },
    creditsPosition: 'bottom-right',
    zoomOutButton: {
      backgroundColor: '#fff',
      backgroundAlpha: 0,
    },
    zoomOutText: '',
    pathToImages: layoutPaths.images.amChart,
  };
}

getLastHour() {
  const path = `${environment.base_url}/readings/`;
  return this.http.get(path)
    .toPromise()
    .then(response => response.json());
}

getMessages() {
  const observable = new Observable(observer => {
    this.socket = io(environment.socket_url);
    this.socket.on('message', (data) => {
      observer.next(JSON.parse(data.text));
    });
    return () => {
      this.socket.disconnect();
    };
  })
  return observable;
}
```

```
  parseDataArray(data) {
    const parsedData = [];

    for (const element of data) {
      parsedData.push(this.parseDataElement(element));
    }
    return parsedData;
  }

  parseDataElement(element) {
    const newDate = new Date( this.firstDate );
    newDate.setTime( element.timestamp * 1 );
    return {
      timestamp: newDate,
      temperature: parseInt(element.temperature),
      humidity: parseInt(element.humidity),
    };
  }
}
```

dashboard.component.html

```
<div class="row">
  <ba-card class="col-xlg-12 col-xl-12 col-lg-12 col-md-12 col-sm-12 col-xs-12" title="Temperature/Humidity (real time)" baCardClass="medium-card">
    <ba-am-chart baAmChartClass="dashboard-line-chart" [baAmChartConfiguration]="chartDataRest" (onChartReady)="initSocketChart($event)"></ba-am-chart>
  </ba-card>
</div>

<div class="row">
  <ba-card class="col-xlg-12 col-xl-12 col-lg-12 col-md-12 col-sm-12 col-xs-12" title="Temperature/Humidity (last hour)" baCardClass="medium-card">
    <ba-am-chart baAmChartClass="dashboard-line-chart" [baAmChartConfiguration]="chartDataSocket" (onChartReady)="initRestChart($event)"></ba-am-chart>
  </ba-card>
</div>
```

Glosario

IoT: Internet of Things

RP: Raspberry Pi

HTTP: Hypertext Transfer Protocol

MQTT: Message Queue Telemetry Transport

AMPQ: *Advanced Message Queuing Protocol*

BLE: *Bluetooth Low Energy*

GSM: Global System for Mobile communications

AES: *Advanced Encryption Standard*

SSL: *Secure Sockets Layer*

DoS: Denial of Service

DNS: Domain Name System

VPN: Virtual Private Network

OSI: Open System Interconnection