UNIVERSIDAD DE VALLADOLID
E.T.S.I. TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

Soluciones de edge computing para IoT: análisis y desarrollo de un prototipo

Autor:
D. Álvaro Puente Escudero
Tutores:
D. Ignacio de Miguel Jiménez
D. Ramón José Durán Barroso
Valladolid, julio 2022

Título:	**Soluciones de *Edge Computing* para IoT: análisis y desarrollo de un prototipo**
Autor:	**D. Álvaro Puente Escudero**
Tutores:	**D. Ignacio de Miguel Jiménez** **D. Ramón José Durán Barroso**
Departamento:	**Teoría de la Señal y Comunicaciones e Ingeniería Telemática**

Tribunal:

Presidente:	**Ignacio de Miguel Jiménez**
Vocal:	**Ramón José Durán Barroso**
Secretario:	**Juan Carlos Aguado Manzano**
Suplente:	**Noemí Merayo Álvarez**
Suplente:	**Evaristo J. Abril Domingo**

Fecha:

Calificación:

Resumen

El mercado del procesamiento en la nube (cloud computing) está dominado por tres grandes plataformas: Amazon Web Services (AWS), Microsoft Azure y Google Cloud. El aumento de la importancia de la Internet de las Cosas (loT, Internet of Things), la necesidad de acercar los recursos de cómputo a los dispositivos y a los usuarios finales para reducir la latencia (dando lugar a la computación en el borde o edge computing), y la necesidad de analizar los datos recopilados, están haciendo que dichas plataformas ofrezcan productos específicos enfocados a esas cuestiones.

En este Trabajo Fin de Grado se analizan las posibilidades de estas tres plataformas orientadas a loT y edge computing, y se describe además una prueba experimental utilizando los módulos de una de las plataformas analizadas, AWS. Concretamente, se ha desarrollado

un prototipo de un caso de uso relacionado con la conducción autónoma. Se ha empleado un microcontrolador Arduino como dispositivo electrónico que iría embebido en un vehículo, y una Raspberry Pi actuando como servidor de borde. El microcontrolador recopila información sobre la distancia a la que se encuentran obstáculos (y si ha habido un choque o no), y la envía al servidor de borde. El servidor de borde procesa esa información y envía la respuesta de vuelta al microcontrolador junto con información ambiental. Además, el servidor de borde también se comunica con la nube para notificar de los eventos de choque.

Palabras Clave

Computación en el borde, Conducción Autónoma, AWS IoT Greengrass, Solución IoT, Componente

Abstract

The cloud computing market is dominated by three major platforms: Amazon Web Services (AWS), Microsoft Azure and Google Cloud. The increasing importance of the Internet of Things (IoT), the need to bring computing resources closer to devices and end users to reduce latency (leading to edge computing), and the need to analyze the data collected, are causing these platforms to offer specific products focused on these issues.

In this Final Degree Project we analyze the possibilities of these three platforms oriented to loT and edge computing, and we also describe an experimental test using the modules of one of the analyzed platforms, AWS. Specifically, a prototype of a use case related to autonomous driving has been developed. An Arduino microcontroller has been used as an electronic device that would be embedded in a vehicle, and a Raspberry Pi acting as an edge server. The microcontroller collects information about the distance to obstacles (and whether or not there has been a crash), and sends it to the edge server. The edge server processes that information and sends the response back to the microcontroller along with environmental information. In addition, the edge server also communicates with the cloud to notify of crash events.

Keywords

Edge Computing, Autonomous Driving, AWS IoT Greengrass, IoT Solution, Component

AGRADECIMIENTOS

A mis tutores, por la confianza depositada en mí y por el apoyo recibido.

A la Universidad de Valladolid, por todos los medios que me ha facilitado.

A mis padres, Santiago y Beatriz, y mis hermanos, Almudena y Juan, por su incesante ayuda. Sin ellos no habría sido posible.

A mis compañeros de clase y amigos, en especial a Fernando, Alfonso, Alberto y Mario, por hacer aun más agradable esta etapa.

Al Colegio Mayor Peñafiel, por ser un pilar en mi desarrollo como persona.

A mis abuelos, porque sé que estarían orgullosos de mi trabajo.

Este trabajo ha sido posible gracias a las becas de colaboración del Ministerio de Educación y Formación Profesional.

Este trabajo ha sido financiado por la Consejería de Educación de la Junta de Castilla y León y el Fondo Europeo de Desarrollo Regional (proyecto VA231P20), y el Ministerio de Ciencia e Innovación y la Agencia Estatal de Investigación (proyecto PID2020-112675RB-C42 financiado por MCIN/AEI/10.13039/501100011033)

Índice general

Índice de figuras

Glosario de acrónimos

AI Artificial Intelligence.

eMBB Enhanced Mobile Broadband.

GPOS General Purpose Operating System.

IoAV Internet of Autonomous Vehicles.

IoT Internet of Things.

IoV Internet of Vehicles.

LCD Liquid Cristal Display.

LED Light Emitting Diode.

M2M Machine to machine.

MEC Mobile *Edge Computing*.

MIMO Multiple Input, Multiple Output.

ML Machine Learning.

mMTC Massive Machine-Type Communications.

MQTT MQ Telemetry Transport.

PLS Physical Layer Security.

RTOS Real Time Operating System.

SWIPT Simultaneous Wireless Information and Power Transfer.

uRLLC Ultra-Reliable and Low Latency Communitations.

V2I Vehicle to Infraestructure.

V2V Vehicle to Vehicle.

V2X Vehicle to Everything.

CAPÍTULO 1

Introducción

1.1. Contexto

El continuo desarrollo tecnológico que hemos experimentado estas últimas décadas en aspectos como capacidad, velocidad de procesamiento y de red ha permitido conectar no solo móviles, ordenadores o similares sino también sensores, electrodomésticos u cualquier otro componente que pudiera pensarse. Esto plantea serios problemas con la conectividad ya que, a pesar de haber mejorado considerablemente, puede resultar insuficiente al tener semejante cantidad de dispositivos conectados a Internet.

Para 2023 se espera que el número de dispositivos conectados a redes IP sea tres veces la población mundial (29,3 mil millones) entre los cuales 14,7 mil millones serán dispositivos IoT (comparados con los 6,1 mil millones que suponen en 2018). Esto implica un crecimiento del 59,25 % con respecto a 2018 [aia20]. Además, los datos generados para 2025 se estiman en torno a los 163 zettabytes en 2025 según la International Data Corporation (IDC) [RGR17]. Esto deja en evidencia la necesidad de nuevas tecnologías que permitan un tratamiento eficiente de los datos. La computación en la nube puede no ser una buena solución para el manejo de todos los dispositivos IoT pues la cantidad de datos que generan puede llegar a saturar tanto la capacidad de la red como la capacidad de procesamiento.

Es aquí donde entra en juego el paradigma computacional *Edge Computing* que, a grandes rasgos, permite el tratamiento de los datos cerca de donde se generan, es decir, en el borde de la red. Esta tecnología cumple con las "Tres Leyes del Internet of Things" [CH20]:

- *Leyes de la física*. Existe, cada vez más, una necesidad de obtener respuestas en tiempo real sobre todo para los dispositivos autónomos. Una comunicación con la nube provoca, para muchas soluciones IoT, retardos prohibitivos debido a limitaciones físicas. Por tanto, sería deseable un procesamiento local que permitiese sortear esos problemas relacionados con el retardo.
- *Leyes de la economía*. Un enorme volumen de datos puede generar cuellos de botella y elevados costes de procesamiento en la nube dando lugar a pérdidas de rendimiento. Las empresas asumirían un elevado coste derivado del almacenamiento y transmisión de datos a la nube.
- *Leyes de la tierra*. Las restricciones legales y geográficas pueden dificultar la recopilación y transferencia de datos. Además, algunas partes del mundo no tienen la infraestructura para admitir la conectividad IoT, lo que limita la disponibilidad de la nube.

Dejando claro, pues, la necesidad permanente de servicios de borde para cumplir esta leyes. Con tecnologías como *Edge Computing* se cumplen los requisitos de retardo al situarse cerca del origen de los datos y sin tener necesidad de una conexión a Internet. Además, el ancho de banda disponible entre el servidor de borde y los dispositivos generadores de datos esta compartida por menos dispositivos que si fuese una conexión a la nube. El procesamiento puede realizarse en estos servidores de borde sin necesidad de conectarse a la nube. Esto supone un ahorro en costes mayor cuanto mayor sea el volumen de datos que tratamos en el borde y no en la nube. Incluso este procesamiento puede servir como "filtro" que únicamente envíe a la nube datos útiles que justifique el uso de una conexión a la nube. Al tener este servidor de borde en nuestro dominio, nuestra información también queda bajo nuestro dominio lo que nos permite una mayor confidencialidad y cumplimiento de normativas relativas al procesamiento en las diferentes geografías.

Por tanto, el futuro del Internet de las Cosas (IoT) depende del desarrollo de este tipo de tecnologías que evitan en buena medida un uso de recursos, compartidos por muchas entidades generadoras y receptoras de datos, que provoquen ciertos picos de congestión lo que es vital para el funcionamiento de las redes. Además, este tipo de tecnologías da lugar a una nueva amplia gama de casos de uso del IoT al incorporar una serie de ventajas novedosas en este sector.

1.2. Objetivos

El objetivo principal de este Trabajo de Fin de Grado es conseguir un desarrollo de una solución IoT basada en el paradigma computacional *Edge Computing* y lograr implementarla de forma eficaz, segura y útil para el entorno donde se pretender implantar. No solo nos centraremos en la implementación sino que también analizaremos las repercusiones que tiene sobre el entorno considerado.

Sin embargo, para conseguir este objetivo es preciso realizar previamente una serie de tareas:

- Conocer con detalle la tecnología *Edge Computing* para comprender su funcionamiento. Una vez tenemos una idea clara de lo que supone, podemos elaborar una serie de posibles casos de uso genéricos en los cuales esta tecnología es realmente beneficiosa. Seleccionaremos uno de los casos de uso en base a criterios de utilidad, usabilidad y presupuestarios. El caso de uso se concretará en una idea que va a fundamentar nuestro prototipo.
- Seleccionar el servicio a partir del cual podremos desarrollar y poner en funcionamiento nuestro prototipo. Compararemos distintos proveedores de servicios que proporcionan un cierto tipo de plataformas que pueden ser útil para desarrollar el prototipo. Seleccionaremos aquel proveedor que otorgue mejores condiciones a nuestra solución IoT.
- Describir la solución que se pretende implementar. En este paso, haremos una descripción de los elementos que forman parte de esta y diseñaremos la arquitectura de la solución IoT.

En la Figura 1.1 puede observarse de forma esquemática los objetivos previamente expuestos y qué se pretender conseguir en este trabajo.

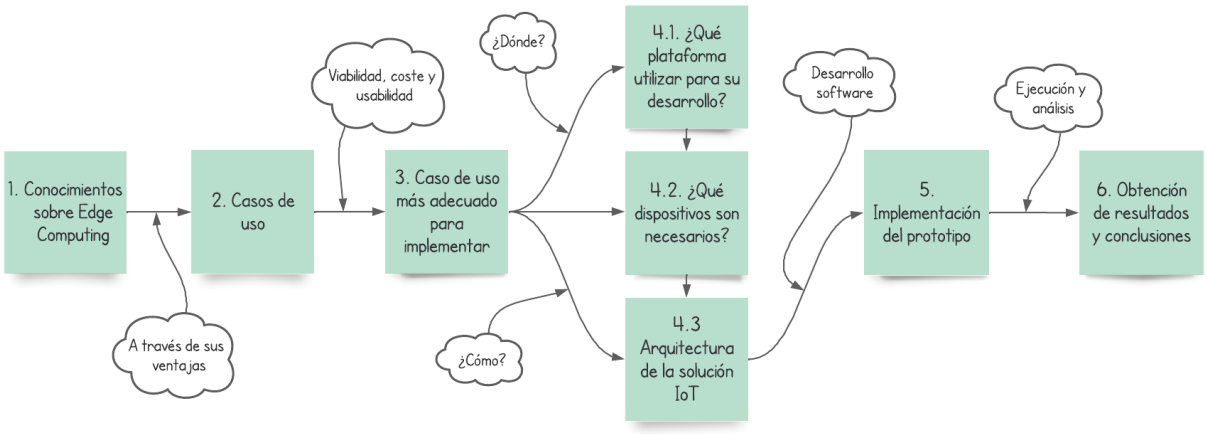


Figura 1.1: Esquema conceptual de los objetivos a perseguir

1.3. Estructura de la memoria del Trabajo Fin de Grado

Nuestro trabajo constará de diez capítulos para lograr los objetivos marcados.

- *Capítulo 1.* Explicamos los objetivos que debe cumplir este trabajo.
- *Capítulo 2.* Tratamos explicar de la forma más concisa posible en qué consiste *Edge Computing* centrándonos en sus principales características
- *Capítulo 3.* Enfocamos este capítulo a los inconvenientes que tiene esta tecnología si no se usa en los casos de uso para los que esta pensado.
- *Capítulo 4.* Hacemos hincapié en los casos de uso que pueden resultar adecuados para implementar esta tecnología.
- *Capítulo 5.* De entre todos los casos de uso que hemos visto en el capítulo anterior, nos vamos a centrar en uno de ellos. Para este caso de uso seleccionado, describiremos más detalladamente en qué consiste y pensaremos dos funcionalidades que trataremos de implementar a través de un prototipo.
- *Capítulo 6.* En este capítulo, se exponen las herramientas que deben converger para poder diseñar, desarrollar e implementar las funcionalidades que pretendemos incluir en nuestro prototipo.
- *Capítulo 7.* En este apartado, exponemos los elementos que vamos a utilizar para desarrollar el prototipo. También compararemos los diferentes servicios en la nube que pueden servir para implementar una solución IoT basada en *Edge Computing* y seleccionaremos, en base a ciertos criterios, cual es el más óptimo.
- *Capítulo 8.* En este apartado haremos el diseño hardware para representar todas las conexiones que deben llevarse a cabo para formar la solución IoT y el diseño software mediante diagramas de flujo.
- *Capítulo 9.* En este capítulo mostraremos todo el procedimiento y desarrollo que es necesario, a nivel de software y hardware, para crear el prototipo con las funcionalidades comentadas.

- *Capítulo 10.* Finalmente, llevamos a cabo un análisis de los resultados que obtenemos del prototipo y los ponemos en contraste con lo que se espera de una solución IoT basada en *Edge Computing*

CAPÍTULO 2

¿Qué es Edge Computing?

Como vimos en la introducción, *Edge Computing* es un paradigma computacional que se podría definir, en pocas palabras, como aquel que permite realizar un procesamiento cercano a donde se generan los datos, esto es, en el borde de la red [JCS18]. De esta manera, se logran mitigar varios problemas [KSDN22]:

- Menor utilización del ancho de banda disponible con la nube ya que buena parte del procesado se realiza en el borde. En este sentido, lo que se logra es una reducción de los datos enviados a la nube y en consecuencia, una reducción de costes derivados de la transmisión de una cantidad elevada de datos a la nube y también del coste subyacente que se requiere para el procesamiento y almacenamiento en la nube de dichos datos.
- Es posible garantizar un retardo máximo acotado. A través de una computación en la nube no podemos asegurar un retardo mínimo principalmente porque hay un uso compartido de los recursos tanto a nivel de ancho de banda como de procesamiento en la nube con lo que pueden producirse picos de tráfico que impidan un servicio rápido y eficiente. Además, la propia distancia con los servidores en la nube suele ser suficiente para proporcionar un retardo muy elevado para ciertas aplicaciones con este tipo de requisitos. Con un servidor en el borde, si mantenemos una conexión con este estable, con un ancho de banda capaz de cubrir las necesidades de la solución IoT y con un procesamiento capaz de soportar la tasa máxima de datos que pueda proceder de la solución IoT, podemos garantizar un retardo máximo de respuesta para peticiones de dicha solución. Esto abre un abanico de nuevas aplicaciones de IoT.
- Se evitan problemas legales referidos al tráfico de datos según en qué geografía. El procesamiento al hacerse en el borde, actúa siempre bajo las restricciones legales donde se sitúa. Además, buena parte de los datos generados por los dispositivos IoT están estrechamente relacionados con la vida de los usuarios. Imaginémonos cámaras en el interior de una casa transmitiendo datos a la nube, el riesgo de una filtración de información es elevado. *Edge Computing* trataría mejorar aun más la confidencialidad de esos datos evitando tener que enviarlos a la nube. También es cierto que surgen nuevos retos de seguridad con esta tecnología.
- Ahorro de energía. Los centros de datos ubicados en la nube consumen mucha energía. El incremento del número de dispositivos conectados a la red incrementará mucho la energía consumida. Si el número de dispositivos aumenta mucho podría incluso generarse cuellos de botella debidos a un consumo máximo de energía, por ello, *Edge Computing* se presenta como una solución factible para reducir este consumo al hacer buena parte del procesamiento en el entorno exterior de la red, evitando que buena parte del tráfico

llegue a los centros de datos o circule por el núcleo de la red.

En este punto, convendría analizar con más detalle a que nos referimos con el concepto de "borde" que hemos estado utilizando hasta ahora. Para entender bien este concepto, primero deberíamos entender que es necesario una arquitectura diferente a la utilizada con la computación en la nube. Con este nuevo planteamiento, lograríamos alcanzar el propósito de la tecnología *Edge Computing* que es "establecer un paradigma computacional descentralizado y potencialmente jerárquico para respaldar el desarrollo y las operaciones de los sistemas distribuidos" [CH20]. Esto supone un nuevo salto en la forma de conectividad que hasta ahora planteábamos. ¿Por qué? pues porque la introducción de tal cantidad de dispositivos de distinta naturaleza a la red hace necesario plantear otro tipo de estructura para poder soportar una mayor generación de datos sin saturar la red.

Nuestro objetivo, para lograr una visión amplia acerca de *Edge Computing*, consistirá en entender sus características fundamentales: descentralización y jerarquización.

2.1. Un sistema descentralizado y jerarquizado

Para entender qué supone un sistema descentralizado, es preciso comprender los distintos tipos de redes que existen y sus principales características. En la Figura 2.1, se muestran los diferentes tipos más básicos de redes que definió Paul Baran en el año 1962 [Bar62]. A pesar de no ser muy preciso con lo que se entiende actualmente por una red distribuida, nos es útil para entender conceptualmente aspectos básicos de estos tres tipos de redes.

Un sistema centralizado es aquel en el que los distintos componentes están conectados a un componente central. Tiene el gran inconveniente de que, si desaparece el nodo central, la totalidad del sistema deja de funcionar. En la práctica, se suele utilizar una mezcla de varias topologías en estrella. Además, las estaciones pueden estar situadas a diferentes distancias con el nodo central lo que hace que haya diferentes calidades de conexión. Puede haber riesgo de congestión en el nodo central si todas las estaciones deciden enviar una gran cantidad de datos al nodo central.

Un ejemplo de este sistema es la red de televisión. Desde un mismo foco, donde se recopilan los datos (cámaras de televisión), los datos (imágenes) son emitidas a las distintas televisiones.

Por tanto, este tipo de sistema no define las características buscadas por *Edge Computing*. Es susceptible de generar retardos indeterminados que dependen del estado de la red y del nodo central según el volumen de datos existente en la red en dicho momento lo que resulta incompatible con la masividad de datos generados por los dispositivos IoT.

Un sistema distribuido tal y como define Andrew S. Tanenbaum es "una colección de computadores independientes que aparece a sus usuarios como un único sistema independiente" [TS08]. Sin irnos muy a fondo, se trataría de un sistema en el que no existe un nodo central o principal que actúa como gestor. Se trata de computadores autónomos que actúan conjuntamente para proporcionar un determinado sistema. La forma en que interactúan entre sí dependería del servicio que pretenden proporcionar. Este tipo de sistema oculta a los usuarios la manera en que se comunican y lo hace de forma coherente y uniforme independientemente de dónde y cuándo se produzca la interacción. Son sistemas fácilmente escalables como consecuencia directa de tener ordenadores autónomos y de mantener ocultos como participan entre sí.

Un ejemplo de sistema distribuido es la computación en la nube donde un conjunto de

recursos virtualizados, alojados en el centro de datos de un proveedor en la nube, son utilizados para aprovechar una serie de servicios. Desde el punto de vista del usuario, parece que trata con recursos de su propia máquina. Sin embargo, estos recursos son, además, compartidos por otros usuarios. Estos servicios se pueden adaptar a las necesidades del cliente permitiendo escalabilidad.

Un sistema distribuido tampoco se adecúa a lo que *Edge Computing* pretende conseguir. Para una solución IoT basada en *Edge Computing*, los dispositivos deberían conectarse a un mismo nodo que proporcione el procesamiento en la zona cercana, no debe depender de otros dispositivos cercanos para realizar su procesamiento ya que, si no, no se pueden asegurar ciertos requisitos de retardo pues dependerían del número de dispositivos cercanos y de su capacidad de procesamiento en dicho momento. Esto no quita que pueda haber cierta comunicación entre dichos dispositivos pero dicha comunicación no debe ser determinante para el buen funcionamiento del servicio que se proporciona.

Esto, de nuevo, nos lleva a la necesidad de un nodo que centralice estas funciones pero no una totalidad o gran cantidad de nodos sino aquella parte que se encuentra cercana a dicho nodo para cumplir con las "Tres leyes del Internet de las Cosas" que vimos en la introducción. Por todo ello, surgen los sistemas descentralizados.

Un sistema descentralizado se entiende como el proceso donde la autoridad central delega funciones a otra serie de entidades y estos a su vez podrían delegar otra serie de funciones de menor jerarquía a otras entidades. De esta manera, se genera una jerarquía en la que se proporciona y acepta el control de otros niveles y se mantiene una autonomía a nivel de servicio [CH20]. Como podemos ver, hay una cierta relación entre descentralización y jerarquización aunque, no tiene porque darse, podrían estar todas las entidades en el mismo nivel de control.

Un arquitectura descentralizada es la que mejor se adapta a la tecnología *Edge Computing*, las funciones que típicamente las proporciona la nube, como el almacenamiento y procesamiento, la cual podemos representar como la autoridad central, son delegadas a dispositivos que se localizan en el borde de la red. De esta manera, se procesan únicamente los datos procedentes del área (ahora de reducido tamaño con respecto a la cobertura de un centro de datos) al que esté dando servicio el dispositivo de borde. Con esto, logramos dar un servicio rápido y sin posibles congestiones en cada área. Si convendría matizar aquí que, cuando nos referimos a delegar el procesamiento de datos, nos referimos a la transferencia de las funciones de procesamiento más básicas o sencillas a estos dispositivos del borde de la red. Esto no quita que el servidor de borde, por diversos motivos, delegue funciones de procesamiento a la nube. Este procesamiento que se delega, no tiene que ver con datos con las características básicas de *Edge Computing* (baja latencia y cantidad) y que generalmente necesitan de servicios que los dispositivos en el borde no pueden proporcionar.

Por tanto, puede haber intercambio de datos bidireccional entre la autoridad central y los dispositivos de borde. Es más, lo que estamos definiendo como autoridad central bien puede ser un centro de datos en la nube. *Edge Computing* necesita una gran potencia de computación y soporte de almacenamiento masivo de un centro de computación en la nube, y el centro de computación en nube también necesita el modelo de computación de borde para procesar datos masivos y de privacidad. La computación en la nube y *Edge Computing* son complementarios no sustitutivos [SPX19].

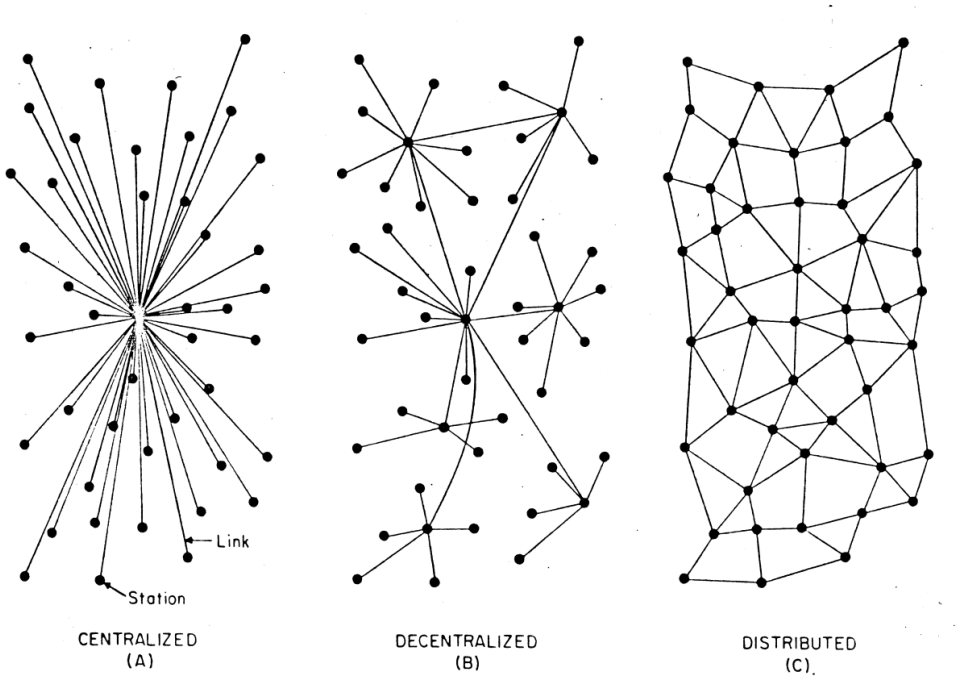


Figura 2.1: Centralized, decentralized and distributed networks. Tomado de [Bar62] (p. 4), por P. Baran

2.2. El borde

El "borde" está siendo una de las palabras más repetidas a lo largo del trabajo, y es un concepto imprescindible para entender *Edge Computing*. Sin embargo, hasta ahora lo hemos estado definiendo de forma muy vaga como zona más externa de la red, aquella que se encuentra cerca del origen y consumo de los datos. Si bien es cierto, la realidad es que el borde se incluye varias ubicaciones como [McK22]:

- Perímetro local, donde la mayor parte de los datos son generados.
- Perímetro de la red de las telecomunicaciones, con centros de datos de operadores y oficinas centrales (en las estaciones base de las redes 5G/6G).

Como podemos ver en la Figura 2.2, podemos ver un esquema de las distintas ubicaciones donde puede estar situado un servidor de borde. Su localización dependería del tipo de servicio que se pretendiese dar.

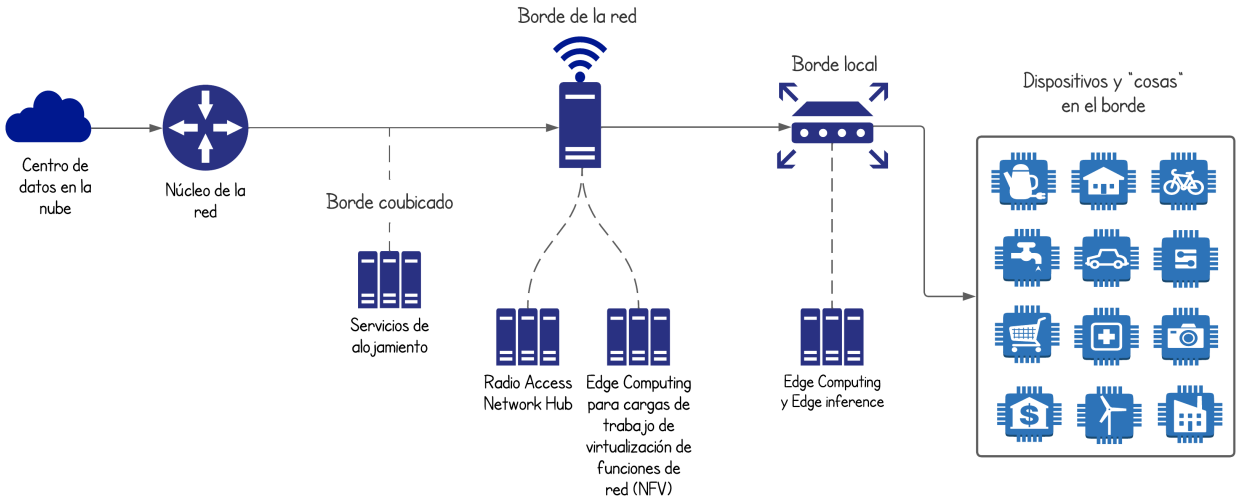


Figura 2.2: Esquema de una red, a partir de [McK22]

En conclusión, *Edge Computing* es un paradigma computacional que aboga por un sistema descentralizado y fuertemente jerarquizado al utilizar sistemas de procesamiento en el borde de la red que se encargan del primer procesado de los datos en un determinado área y parte de estos pueden ser llevados a un nivel superior para procesar y almacenar en centros de datos con mayor capacidad de procesado y almacenamiento. En la Figura 2.3 se muestra un pequeño esquema de lo que supone a nivel de modelado este paradigma.

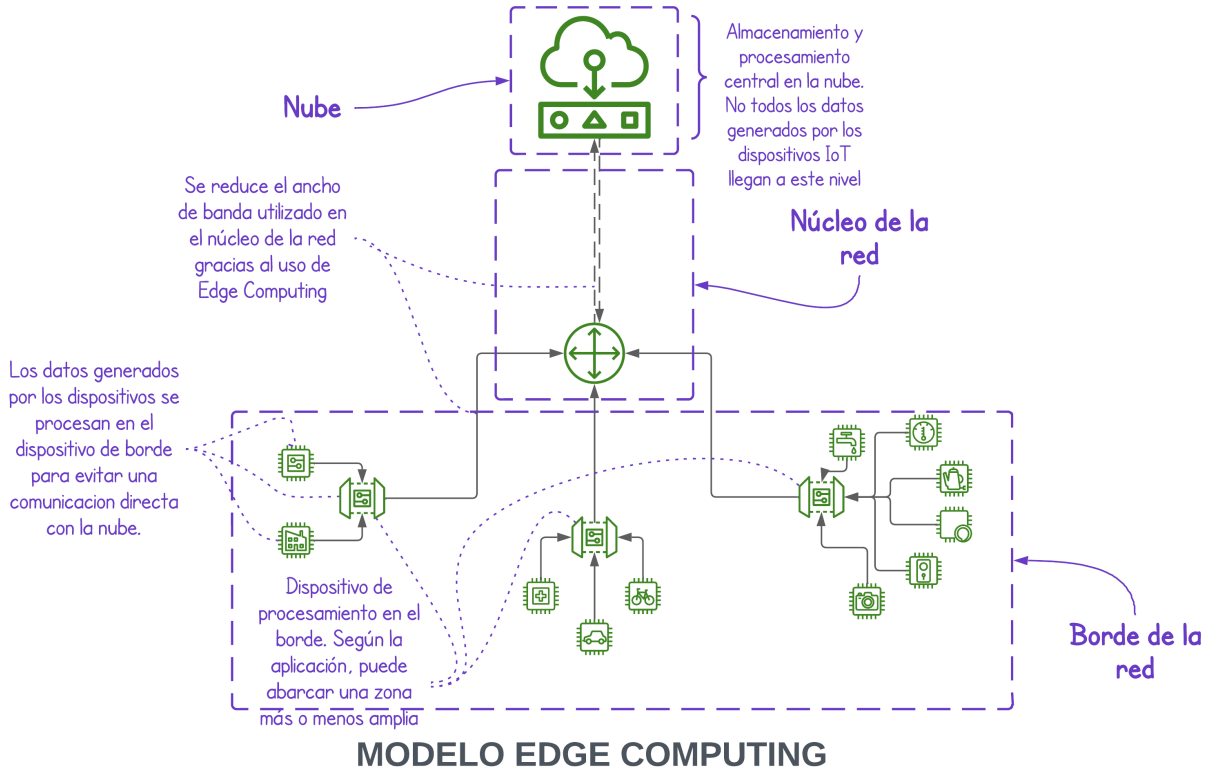


Figura 2.3: Modelo *Edge Computing*

CAPÍTULO 3

Edge Computing: No todo son ventajas

Edge Computing, a pesar de ser una tecnología que aporta un novedoso catálogo de aportaciones, posee ciertas desventajas que hace que no sea factible como solución a cualquier problema. Existen razones por las que es preferible una solución basada en la nube antes que una basada en *Edge Computing*. Veamos algunos de los aspectos a tener cuenta a la hora de determinar si utilizar *Edge Computing* resulta factible para una determinada solución IoT.

3.1. Almacenamiento

Hay que considerar que, a pesar de que el tiempo de subida de datos en un servidor de borde es breve, el almacenamiento en este supone ocupar recursos que son limitados, pues, a diferencia de la nube, donde los recursos de almacenamiento pueden considerarse como ilimitados, no dispone de un gran espacio como lo pueden tener un centro de datos. Su almacenamiento debería ser temporal para las operaciones que se estuviesen realizando en tiempo real o aquellos que resultasen esenciales para otorgar un determinado servicio. Almacenar recursos en estos servidores llegaría a provocar que acabasen saturando en poco tiempo.

Además, como los servidores de borde pueden ejecutarse en diferentes organizaciones, es difícil garantizar la integridad, protección, evaluación del anonimato, el no repudio y la actualización de los datos originales [YLH+18].

3.2. Procesamiento

Si bien el procesamiento de los servidores de borde puede resultar adecuado para atender la demanda de datos de los dispositivos IoT, su limitada capacidad computacional en lo referido a procesamientos demasiado complejos y cuya escalabilidad es un problema desafiante. Puede que para un determinado tipo de procesamientos más laboriosos sea necesario hacer optar por procesamientos en la nube, los cuales ofrecen recursos computacionales complejos y escalables.

3.3. Transmisión y ancho de banda

Una de las principales características que proporciona *Edge Computing* es su baja latencia ante el envío gran cantidad de datos. Esta característica en sí misma no supone una desventaja

pero sí conviene analizar aquello que dejamos de percibir a cambio de esta característica, como son reducciones en almacenamiento y procesamiento. Es posible que ciertas soluciones IoT no requieran de requisitos de latencia pero sí de procesamientos más complejos y de elevados almacenamientos de datos.

Como podemos ver, no todas las características que proporciona *Edge Computing* resultan óptimas para todos los problemas. Será tarea del diseñador de la solución IoT determinar, en base a las características que necesita en su servicio, si es conveniente aplicar un servicio basado en computación en la nube o en *Edge Computing*. Recordemos también que, es posible combinar ambos paradigmas para llevar a cabo nuestra solución. En la Figura 3.1 se muestra un resumen de las características de cada tecnología.

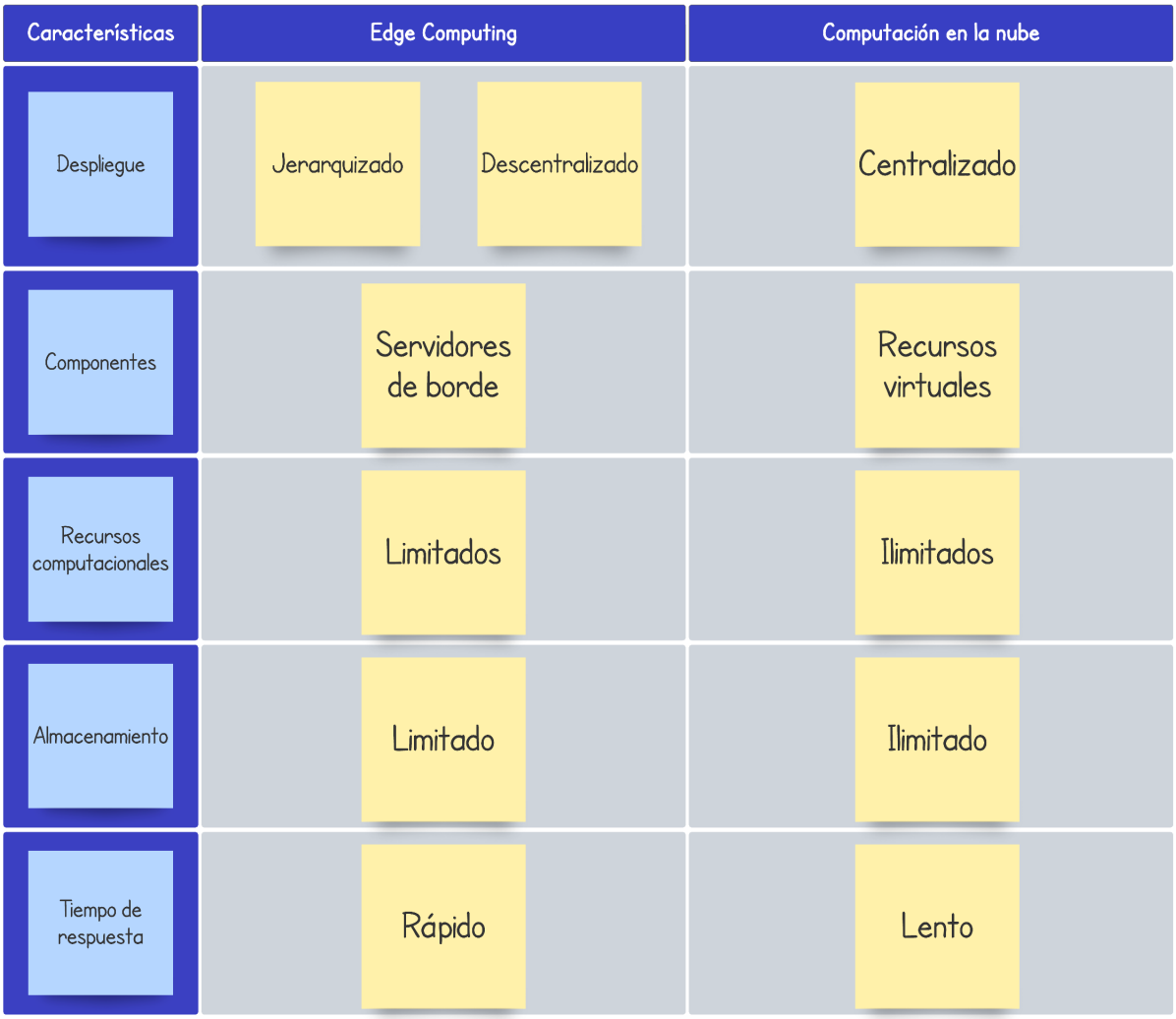


Figura 3.1: Características de *Edge Computing* y computación en la nube

CAPÍTULO 4

Casos de uso

Como hemos estado viendo, *Edge Computing* proporciona una nueva forma de procesar los datos. Lo hace cerca de donde se generan los datos mediante una estructura descentralizada y jerarquizada. Esta novedosa estructura soluciona una serie de problemas que con la computación en la nube no tenían solución. Los beneficios que obtenemos por solucionar estos problemas nos llevan a utilizar esta tecnología para un determinado tipo de aplicaciones. Esto junto al desarrollo de otro tipo de tecnologías como la red de acceso móvil 5G, la cual resulta determinante para el desarrollo del IoT y nos proporciona tres nuevos escenarios de aplicación:

- *Enhanced Mobile Broadband (eMBB).* Proporcionar una tasa binaria máxima de 10 Gbps, determinante para soportar un gran cantidad de datos procedentes de los sensores.
- *Massive Machine-Type Communications (mMTC).* Proporciona un millón de conexiones por kilometro cuadrado, determinante para sensorizar masivamente las ciudades.
- *Ultra-Reliable and Low Latency Communitations (uRLLC).* Proporciona retardos del orden de 1 ms, determinante para mantener una conexión de ultra-baja latencia con el dispositivo de borde.

Además, la continua mejora de los dispositivos de procesamiento en cuanto a memoria, CPU, ... complementan la labor proporcionada por *Edge Computing* para conseguir determinados fines. Los principales campos de aplicación de esta tecnología [KSDN22], y que analizaremos a continuación, son los vehículos autónomos, las ciudades inteligentes, la automatización industrial, la funcionalidad de la red, los videojuegos, la entrega de contenidos, el sector financiero, la realidad aumentada o virtual y el sector sanitario.

4.1. Vehículos autónomos

El propósito de este novedoso campo de aplicación consiste en el manejo independiente de un vehículo, esto es, que ningún usuario interactúe en los aspectos relativos a su movilidad. Para que esto sea posible, es necesario un elevado grado de sensorización que permita determinar todos los parámetros que generalmente percibe el usuario para actuar sobre el vehículo, para poder ser procesados y obtener como resultado las órdenes que permitan al vehículo moverse de forma eficaz, segura y eficiente.

En concreto, los sistemas de conducción autónoma integran estrechamente muchas tecnologías, como la detección, la localización, la percepción, la toma de decisiones así como la interacción fluida con plataformas en la nube para generación de mapas de alta definición

(HD) y el almacenamiento de datos [LLT+19].

Edge Computing entra en escena por varios motivos, el primer motivo es porque es necesario un procesamiento externo centralizado de las condiciones cambiantes de la carretera y para que las decisiones, tomadas por cada vehículo, se envíen a estos servidores con el fin de operar de forma efectiva y comunicarse con el resto de integrantes del sistema. Otro motivo es que ese procesamiento debe producirse en tiempo real, el buen funcionamiento de la aplicación requiere de una conducción muy segura y para ello el vehículo debe actuar rápidamente. Un procesamiento en el borde garantizaría una latencia suficientemente pequeña, como para que el vehículo reaccione rápidamente ante cualquier imprevisto, y debería ofrecer un ancho de banda suficiente para la transmisión de los datos. Los vehículos autónomos necesitan procesar 2 Gb/s en tiempo real [LLT+19].

El ecosistema informático de la conducción autónoma, que puede verse en la Figura 4.1, esta formado por un sistema de computación de borde el cual integra los módulos funcionales en tiempo real, como localización, percepción, planificación y control, etc. El vehículo se comunica con el servidor de borde, y con la nube central a través de las redes 3G/4G/5G. Los vehículos pueden comunicarse entre sí (Vehicle to Vehicle), por ejemplo, a través de redes DSRC [LLT+19], pues una conducción segura requiere también de una buena comunicación entre los vehículos cercanos.

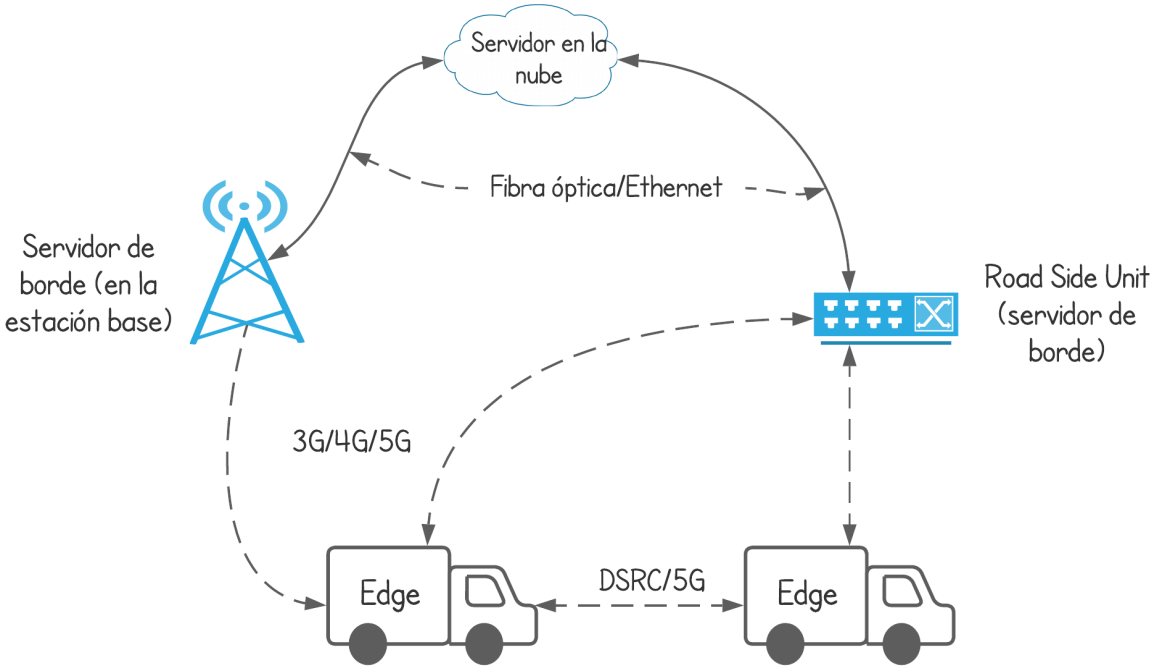


Figura 4.1: Arquitectura *Edge Computing* para conducción autónoma, a partir de [LLT+19]

4.2. Ciudades inteligentes

Las ciudades son un gran núcleo de generación de datos. Hasta ahora buena parte de ellos podían procesarse en servidores situados en la nube ya que esta es escalable y proporciona una gran capacidad. Estos datos proceden de diferentes fuentes, tal y como vemos en la Figura 4.2,

como edificios, vehículos, medidores ambientales, dispositivos inteligentes, ...y uno de los fines es utilizar los datos para administrar los recursos disponibles de forma eficiente y sostenible. Sin embargo, hasta ahora existía cierta limitación en el tiempo de respuesta, inherentes a la computación en la nube, para llevar a cabo determinadas acciones. Esto provocaba un limitación en la calidad y variedad de servicios.

Con *Edge Computing*, podemos dar respuestas en tiempo real lo que amplia la gama de servicios posibles. Esto permitirá un mayor desarrollo de las ciudades y en consecuencia, una mejora de la calidad de vida de sus ciudadanos. En palabras de [KYT+20], "*Edge Computing* extiende los recursos de computación en nube al borde de la red y ofrece conocimiento del contexto, baja latencia, soporte de movilidad, escalabilidad, por nombrar algunos. Por lo tanto, para abordar las limitaciones de la computación en la nube, para habilitar entornos de ciudades inteligentes en tiempo real, la computación de borde es una solución viable".

Así pues, los principales objetivos para la consecución de una ciudad inteligente pueden resumirse en:

- Infraestructura de red que permita la conectividad de los diferentes dispositivos, que deben ser poder ser identificables, autónomos, localizables y legibles.
- Variedad de tecnologías a las que prestar una asidua atención.
- Interfaces inteligentes entre las cosas y las personas.

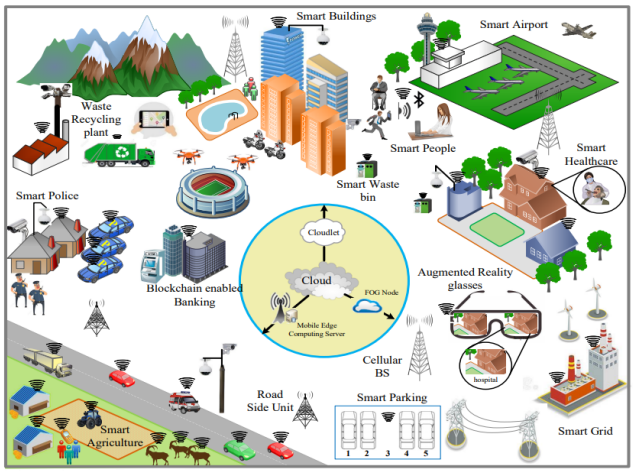


Figura 4.2: Una visión general de la ciudad inteligente habilitada por *Edge Computing*, tomado de [KYT+20] (p. 2)

4.3. Automatización industrial

La automatización industrial se podría definir como el despliegue de la tecnología *Edge Computing* a lugares industriales, que gracias a sus características, mejoran en los siguientes aspectos [WABN20]:

- Mantenimiento o reducción del tiempo de inactividad. Con *Edge Computing* procesamos datos en tiempo real de la maquinaria y a través de modelos de machine learning en el borde, podemos detectar anomalías en su funcionalidad y tomar medidas preventivas antes que correctivas. Lograríamos aumentar el intervalo de tiempo entre las inspecciones de mantenimiento in situ.

- Rendimiento u optimización del proceso. A través de *Edge Computing* podríamos diseñar algoritmos que predigan defectos en la fabricación y así aumentar el rendimiento de la producción. Como analizamos datos en tiempo real, cada proceso de transformación puede ajustarse al nivel general de fabricación evitando cuellos de botella y maximizando la producción.
- Optimización, reducción y control de la mano de obra. *Edge Computing* nos proporcionan herramientas de análisis de la producción en tiempo real. De esta manera, podemos canalizar las cargas de trabajo en aquellas zonas donde más hace falta. Consecuentemente, conseguimos una gran mejora en la productividad, lo que se traduce en un ahorro en horas trabajadas que pueden dedicarse a otros aspectos.
- Mejora de la calidad. Con sistemas predictivos de calidad, de reconocimiento de defectos del producto y métricos sobre los productos producidos en tiempo real gracias a aplicarse en el borde.
- Mejora de la gestión de datos. Esto está directamente relacionado con la reducción del coste que supone el almacenamiento, procesamiento y transferencia de datos. Esto incluye también una mejora en el acceso a los datos, que junto con la visualización en el borde, mejoran la toma de decisiones.

4.4. Funcionalidad de la red

Como dijimos previamente, la explosión de datos y dispositivos no hace nada más que crecer cada año. Estos dispositivos, para poder comunicarse en la red, deben estar identificados a través de una dirección IP. La red actual, con este crecimiento, no va a lograr mantener su proceso de comunicación de forma eficiente debido a una mayor utilización del limitado ancho de banda. Un servidor de borde podría actuar como un primer frente para atender solicitudes y realizar determinados procesamiento evitando que así buena parte de los datos no penetren el núcleo de la red. Así, se racionaliza el uso del ancho de banda.

4.5. Videojuegos

El sector de los videojuegos tiene un impacto global en todo el mundo. Tan solo en España el número de videojugadores ha superado la cifra de 15 millones [Vá20]. Si nos centramos en aquellos juegos de carácter multijugador, tienen un impacto en el uso de la red pues tienen un elevado consumo de ancho de banda y mucha sensibilidad al retardo. Con servidores de borde se podría reducir el ancho de banda y retardo mejorando la experiencia del usuario. El problema del retardo y el ancho de banda se solucionaría completamente si fuesen videojuegos multijugador que abarcasen el área que tiene asignada el servidor de borde; si no, estaríamos ante una solución parcial pues todavía datos críticos, necesarios para el funcionamiento en línea, deben ser enviados a la nube.

4.6. Entrega de contenidos

Con servidores de contenido, como música, vídeos o acceder a una página web, en el borde reduciríamos mucho la respuesta del servidor para obtener este servicio y además lo recibiríamos

a una velocidad mayor. En resumidas cuentas, la calidad de servicio mejoraría.

4.7. Sector financiero

La red del sistema financiero se caracteriza por realizar un número elevado de transacciones las cuales deben ser totalmente seguras. Incorporar *Edge Computing* en este sector supondría mejorar la velocidad de las transacciones, que en muchos casos resulta crucial para evitar contratiempos para firmas y cliente, además de otorgar mejoras adicionales de seguridad.

4.8. Realidad aumentada/realidad virtual

"La realidad aumentada facilita una apariencia real a los usuarios mediante la superposición de elementos digitales sobre objetos/entornos del mundo real" [KSDN22]. Esta superposición necesita de un procesamiento que hasta ahora se hace en la nube y en consecuencia, tiene cierto retardo. De nuevo, con servidores en el borde este retardo puede ser drásticamente reducido mejorando la experiencia del usuario.

4.9. Sector sanitario

Edge Computing proporciona novedosas oportunidades para este sector ya que es capaz de otorgar un procesamiento más rápido y de forma local, lo que otorga un mayor grado de seguridad como ya vimos. El aspecto de seguridad tiene bastante peso ya que la información de los pacientes debe mantenerse totalmente confidencial. Además, es posible atender cualquier urgencia de forma inmediata sin depender de que la conexión con Internet sea estable. Podrían también crearse aplicaciones, que no requieran de un procesamiento complejo, que midiesen ciertos parámetros de nuestro cuerpo con el fin de prevenir enfermedades y tomar medidas de inmediato. En definitiva, *Edge Computing* favorece el mantenimiento de una vida más saludable y segura.

Estos son solo algunos de los casos de uso más sonados para esta tecnología pero abarca más campos como el del hogar, oficinas, energético/sostenibilidad, comercio y muchos otros más.

CAPÍTULO 5

Nuestro caso de uso

Hasta ahora hemos visto una amplia gama de sectores donde la tecnología *Edge Computing* tiene o puede tener una incidencia destacada debido a sus novedosos beneficios. Ahora centraremos el estudio en el caso concreto donde desarrollaremos nuestro prototipo.

El caso de uso que aplicaremos de forma práctica está relacionado con la conducción autónoma. A continuación, explicaremos más a fondo en que consiste, por qué hemos optado por este caso de uso y finalmente, qué funcionalidades implementaremos de forma práctica.

5.1. Fundamentos de la conducción autónoma

5.1.1. Introducción

La idea de lograr una mundo más sostenible ha generado un cierto desapego en los países desarrollados a la hora de poseer un automóvil en favor de soluciones de movilidad integradas [LLT+19]. Esto provocará un cambio en las tecnologías de transporte que hoy en día utilizamos y que se verán potenciadas por el uso de combustibles alternativos. En este proceso, se espera que los vehículos semiautónomos y autónomos crezcan de manera gradual así como en aspectos de comunicaciones vehiculares. Por lo tanto, puede anticiparse que un futuro paso evolutivo en las comunicaciones vehiculares sea el Internet de los Vehículos Autónomos (IoAV).

Antes de comenzar una discusión sobre IoAV, vale la pena describir el concepto más amplio de IoV y resaltar sus deficiencias existentes. A diferencia de IoT, donde los dispositivos informáticos pasivos forman una red de sensores inteligentes accesibles a través de Internet, IoV implica mucha más movilidad e intervenciones humanas. La seguridad humana y la comodidad son los factores indispensables en el diseño de IoV. Además, existen diferentes niveles de comunicaciones en IoV:

- Comunicaciones intravehiculares.
- Comunicaciones entre vehículos (V2V)
- Comunicaciones de vehículo a infraestructura (V2I)
- Comunicaciones con todo (V2X)

La variedad de comunicaciones viene, además, acompañada con problemas de escalabilidad y de cobertura para la comunicación. Para garantizar un uso eficiente de los recursos, los

vehículos se agrupan en las denominadas nubes vehiculares en los que los datos dentro de una nube vehicular se comparten, procesan y cargan con frecuencia en la nube [LLT+19]

IoAV, se presenta entonces como un sistema con las siguientes características:

- *Organización*. Al eliminar de toda responsabilidad de conducción al humano, IoAV debe actuar de forma adecuada ante diversos escenarios adversos como son condiciones ambientales, fallos de software o ante actividades maliciosas en la red.
- *Configuración*. Un sistema IoAV se configura cumpliendo con las políticas de alto nivel que especifican las autoridades de control de tráfico.
- *Optimización*. Una solución IoAV debe ser capaz de optimizar sus decisiones a través del contexto en el que se encuentra, es decir, debe ser capaz de tomar decisiones a través de la monitorización y experimentación.
- *Protección*. Debe de minimizar el efecto de fallos en cascada así como dar seguridad ante cualquier ataque malicioso. Debe ser autosuficiente para lidiar con los posibles ataques. Para ello, debe anticiparse a los problemas, analizando el comportamiento de los vehículos y sensores.

5.1.2. Tecnologías habilitadoras

A pesar de que todavía, con la tecnología actual, no es posible desarrollar una solución IoAV completa, existe una multitud de avances prometedores que podrá llegar a proporcionar los medios suficientes para obtener los beneficios completos de IoAV [JWJZ18]. Entre estas posibles tecnologías habilitadoras se encuentran:

- *Detección inteligente*. Se requieren tres tipos de sensores inteligentes: de detección, ambientales y de retrodispersión tal y como se ilustra en la Figura 5.1.
- *Computación en la nube*. Los proveedores de computación en la nube ofrecen servicios que permiten desplegar soluciones *Edge Computing*, indispensable para una comunicación V2I.
- *Big Data Vehicular*. Se trataría de extraer la información que hay detrás de los datos obtenidos por los sensores, vehículos y otros agentes [Lit17].
- *Técnicas de seguridad*. Una solución IoAV debe ser muy robusta frente ataques maliciosos puesto que es vital lo que hay en juego. Esto se complicaría aún más en ambientes donde conviven vehículos convencionales, semiautónomos y autónomos. Así pues ha surgido la seguridad de capa física (PLS, *Physical Layer Security*) para evitar transmisión de datos falsificados y una mejora del rendimiento en la confidencialidad. Esto, unido con tecnología blockchain, puede construir modelos confiables para IoAV [JWJZ18].
- *Comunicaciones mmWave*. Como debe realizarse un intercambio de comunicación rápido y de un gran volumen de datos, pues se deben descargar mapas e imágenes en tiempo real, resulta imperante la necesidad de unos enlaces de comunicación inalámbrica del orden de los Gbps no solo para V2I sino también para enlaces V2V. Utilizando bandas de onda milimétrica, disponibles en una red de acceso 5G, sería posible lograr este objetivo.

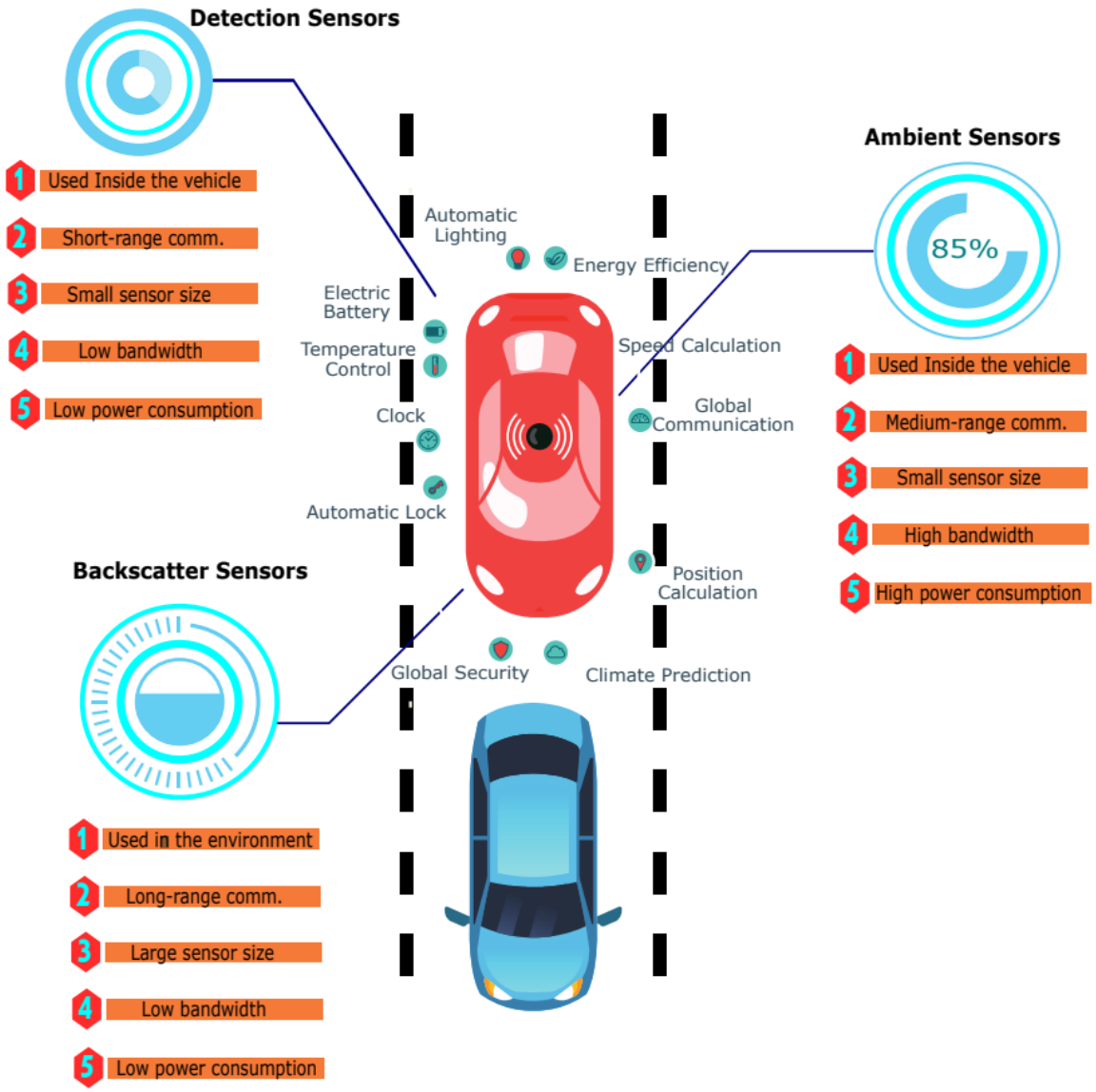


Figura 5.1: Sensorización necesaria para la movilidad autónoma de los vehículos, tomado de [LLT+19] (p. 5)

5.1.3. Arquitectura IoAV

En esta sección, adoptamos un enfoque ascendente y brindamos una vista panorámica de la arquitectura propuesta de IoAV. A diferencia de las arquitecturas propuestas para IoV, consideramos dividir IoAV en tres capas, es decir, capa física, capa virtual y capa de gestión. Como se ilustra en la Figura 5.2, cada capa tiene una contribución clave para habilitar los servicios de IoAV y al mismo tiempo permite la construcción de una plataforma para el intercambio de información en tiempo real. Las tecnologías de la capa física garantizan una comunicación sostenible, de baja latencia y de alta velocidad de datos para la capa virtual, lo que proporciona un mecanismo flexible para administrar los recursos según las políticas de nivel superior descritas en la capa de administración. [LLT+19]

Capa física

En esta capa encontramos la estructura sobre la que se consolida la comunicación para IoAV. En este sentido, las técnicas de acceso múltiple cuasiortogonales y no ortogonales son los dos principales contendientes para satisfacer las demandas futuras de comunicaciones vehiculares espectralmente eficientes [LNPW14].

Los arrays de antenas a gran escala Massive MIMO (*Multiple Input, Multiple Output*) permiten una mayor flexibilidad y precisión en la que los vehículos podrán comunicarse mediante multiplexación en frecuencia en el espectro de ondas milimétricas (mmWaves).

Las técnicas de comunicación ecológicas, como la transferencia inalámbrica simultánea de información y energía (SWIPT, *Simultaneous Wireless Information and Power Transfer*), tienen el potencial de cargar pequeños sensores a lo largo de la carretera o autopista, allanando así el camino para la carga de sensores sin batería [JWJZ18].

Tecnologías de comunicación entre vehículos (V2V) puede mejorar aún más el desempeño de IoAV.

Puede observarse, de nuevo, que la red de acceso 5G es un perfil candidato para sentar las bases de la capa física ya que este tipo de tecnologías forman parte de ella o son compatibles.

Capa virtual

En esta se implementan, sobre hardware de uso general, los paradigmas computacionales proporcionados por proveedores de computación en la nube como *Edge Computing*. Se trata de crear una infraestructura de procesamiento y almacenamiento descentralizado y jerarquizado. Esta capa debe lograr también que haya una fuerte coordinación entre el borde y la nube.

Capa de gestión

Se trata de la capa que administra los distintos recursos que están distribuidos geográficamente. La utilización eficiente de estos recursos garantizaría el buen funcionamiento de las diferentes infraestructura informáticas. En concreto, debe exponer diferentes vistas de la infraestructura y administrar los recursos de la nube y el borde

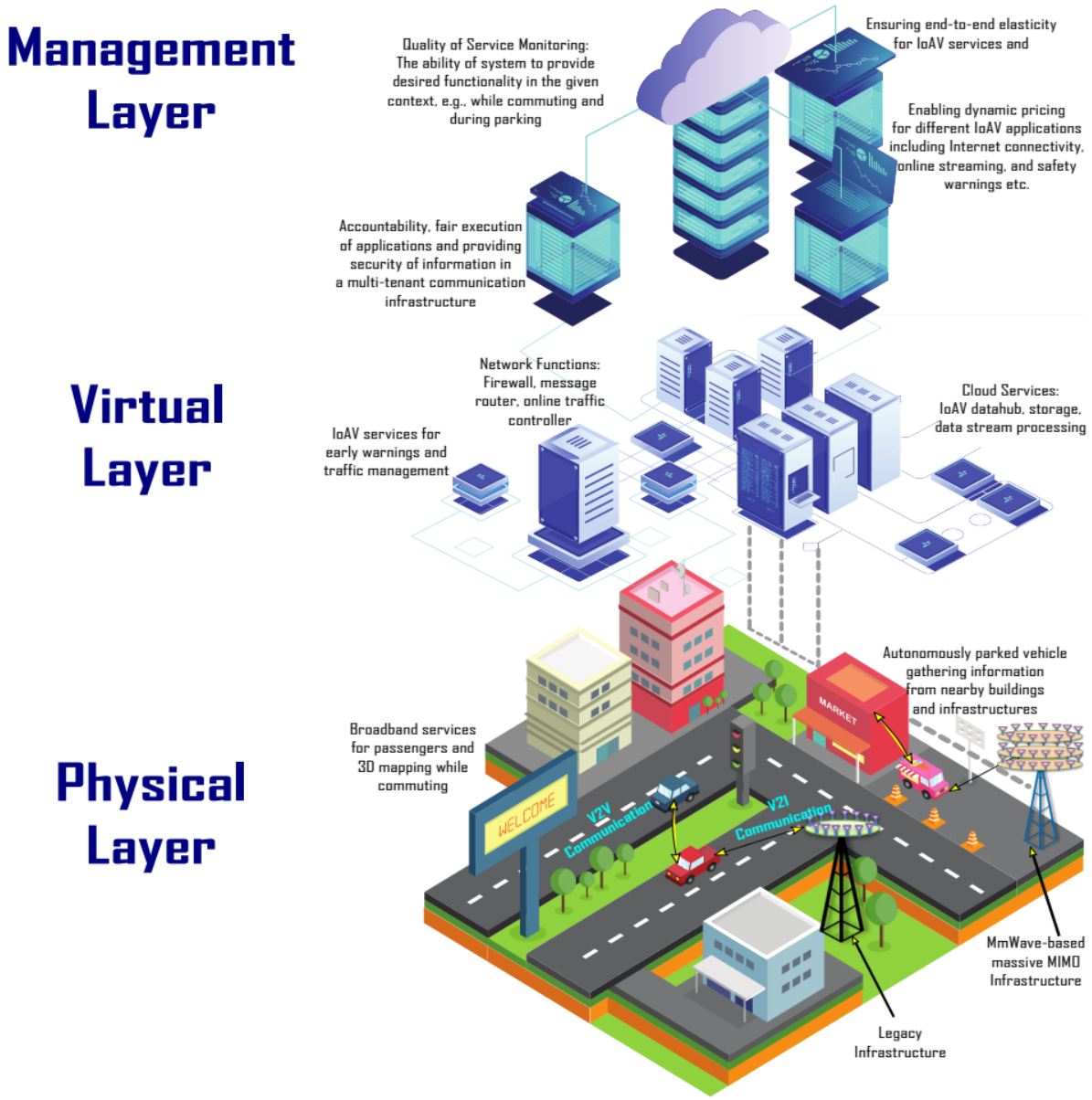


Figura 5.2: Arquitectura IoAV, tomado de [LLT+19] (p. 7)

5.1.4. Desafíos

Como ya hemos dicho, este tipo de tecnología aun no esta plenamente desarrollada y que requiere de comunicación con todo tipo de dispositivos (V2X) para reducir costes. Los principales desafíos a los que se enfrenta son [LLT+19]:

Decisiones cooperativas

La decisión cooperativa, que podemos definir como el intercambio de información de vehículos con el fin de que tomen ciertas decisiones a partir de esta, debe adaptarse a una gran diversidad de entornos mediante una cobertura de corto alcance lo que supone un gran reto. Un traspaso proactivo y la asignación de recursos pueden ser soluciones factibles. Además esto puede verse mejor realizable con conexiones 5G [aec18].

Detección cooperativa

La detección cooperativa está centrada en compartir información entre vehículos (V2V) e infraestructura (V2I). Es, por tanto, aquí donde los problemas relacionados con la transmisión de información de los sensores del vehículo sale a la luz.

El primer problema tiene que ver con garantizar una comunicación en tiempo real. La transmisión de elevadas cantidades de datos al servidor de borde, que surge para solucionar este tipo de problemas, puede llegar a provocar cierto retraso. Por ejemplo, un vehículo con diez cámaras de alta resolución pueden generar 2 giga-píxeles de datos por segundo mientras que se requieren 250 billones de operaciones informáticas por segundo para procesar los datos rápidamente [GLZ21]. La solución pasaría por dotar a los servidores de borde de un gran poder computacional.

Además de aquí se podría derivar otro problema, y es que, durante esa transmisión es posible que el vehículo haya salido del rango de servicio de un determinado servidor lo que provocaría que el vehículo se quedase momentáneamente fuera de servicio y las operaciones que se estaban realizando no lleguen. En un servicio que debe ser vital asegurar una completa recepción de toda la información dadas sus implicaciones en la vida real, así pues, no podemos permitir que cierta información no llegue. Por tanto, el vehículo en estos casos debe, por una parte, siempre conectarse a aquel servidor que ofrezca potencialmente una mayor cobertura y debe haber comunicación entre servidores para que estos datos, que no se han podido transmitir, se envíen a aquel servidor que ahora este ofreciendo el servicio al vehículo. Esto se conoce como migración del servicio y se puede producir cuando un vehículo se vaya alejando del alcance de un servidor y se vaya adentrando en el alcance de otro.

El otro problema tiene que ver con la compensación dinámica del coste de los sensores de infraestructura y sensores del vehículo, que podrá ser solventada a través del mercado del automóvil.

5.2. La conducción autónoma, un gran abanico de posibilidades

Existen diversas razones por las que nos hemos decantado por este sector.

Una de ellas es por su elevada flexibilidad a la hora de desarrollar diferentes soluciones IoT. Para poder llevar a cabo de manera satisfactoria un modelo de conducción autónoma, es necesario que converjan una amplia variedad de sistemas, cada uno de ellos destinado a cubrir una funcionalidad para su correcto funcionamiento. Como nuestro tiempo resulta limitado, esta división de las diversas funcionalidades nos permite llevar a cabo un número limitado de ellas y, además, como hay una gran variedad, elegir aquellas que mejor se ajusten a nuestros requisitos, límites, nivel de aprendizaje y presupuesto.

Otra razón es la capacidad que nos otorgan las diversas funcionalidades de la conducción autónoma para realizar ciertas analogías y simplificaciones de lo que verdaderamente se implementa. Por ejemplo, pensemos en el procesador integrado de un coche autónomo, el cual posee una elevada capacidad de computación para poder llevar a cabo todas las tareas que requiere el coche tanto para tareas rutinarias de comprobación como para aquellas que requiere de enviar y procesar información en el borde y atender la respuestas de manera prioritaria. Si bien toda este desarrollo puede ser muy extenso en conceptos y desarrollo, podemos

realizar un primera aproximación de sus principales características utilizando un procesador más sencillo que permitiese desarrollar, con sensores de menor complejidad, funcionalidades concretas basadas en *Edge Computing* de conducción autónoma.

5.3. Funcionalidades que implementaremos

Dentro de nuestro sector, como ya hemos dicho, existen diversas funcionalidades que podemos implementar, de manera separada, para cubrir alguno de los aspectos de la conducción autónoma. A la hora de seleccionar las funcionalidades que implementaremos, nos vamos a basar en los siguientes criterios:

- *Nivel de aprendizaje*. La funcionalidad implementada debe servirnos, en buena medida, como un ejemplo práctico que demuestre todos los conceptos teóricos que hemos visto previamente sobre esta tecnología. Debe permitirnos comprender la tecnología *Edge Computing* desde un punto de vista práctico.
- *Requisitos técnicos*. La orientación de este trabajo no va a consistir en implementar una solución utilizada en este sector debido a su elevada complejidad, pues, requeriría de conocimientos técnicos más ambiciosos entre otras razones. Buscamos desarrollar una funcionalidad en la que pueda establecerse una analogía con una solución utilizada en la conducción autónoma, que siendo más sencilla, nos permita extraer unas conclusiones a las que se obtendrían con una implementación real.
- *Presupuesto*. Directamente relacionado con el aspecto anterior. Como en este trabajo se pretende realizar una primera aproximación práctica a la tecnología *Edge Computing*, no es factible económicamente implementar una solución realista para conducción autónoma. Se trata de poder representar algunos aspectos básicos a través de dispositivos de bajo coste.

  En los siguientes apartados describiremos las funcionalidades que se han implementado, un detector de proximidad y de choque y un sistema monitorizador de las condiciones ambientales.

5.3.1. Detector de proximidad y choque

Descripción

Los sensores ultrasónicos han formado parte en los vehículos desde hace ya muchos años como dispositivos para medir distancias entre objetos cercanos al sensor cuando el coche se mueve a una baja velocidad. En un contexto de conducción autónoma, se pueden utilizar para posicionar los obstáculos del vehículo cuando este se encuentra parado o con una velocidad anormalmente reducida.

Un ejemplo de uso de esta funcionalidad podría ser en contextos de estacionamiento del vehículo. El vehículo necesita saber continuamente su posición con respecto a los objetos cercanos para ir desplazándose hasta lograr estacionar sin provocar ningún choque. Parecería razonable pensar que esta funcionalidad podría actuar sin necesidad de *Edge Computing*, sin embargo, hemos de comprender que estamos en un contexto del que forman parte un número elevado de vehículos por lo que es necesario enviar la información del entorno a un servidor

de borde que permita adecuar dicha información con el contexto del resto de vehículos y así enviar líneas de actuación al vehículo que han sido condicionadas por la actuación del resto de vehículos del entorno logrando evitar accidentes.

Adecuación a nuestros recursos

Si bien la complejidad de diseñar una funcionalidad completa de posicionamiento del vehículo mediante ultrasonidos, más aun si implementamos funciones complementarios como bien puede ser el desarrollo de un sistema de aparcamiento autónomo, es posible ver una pequeña utilidad de este sistema utilizando la tecnología *Edge Computing*. Al fin y al cabo, nuestro trabajo se centra en la utilización de esta tecnología y no de hacer una funcionalidad muy compleja por motivos explicados anteriormente, aunque, sí que es cierto que el hacer una funcionalidad más sencilla no va a permitir explorar en profundidad todos los aspectos de *Edge Computing*.

Por tanto, realizar un sistema de aproximación de objetos a un sensor de ultrasonidos en el cual dicha información sobre la distancia al objeto se envía al servidor de borde para procesarla y enviar de vuelta una cadencia y frecuencia que debe tomar la alarma según la distancia puede sernos, de por sí, útil para analizar aspectos básicos de esta tecnología. Además, si la distancia del objeto y el sensor ultrasonido es muy pequeña y hay un contacto con un sensor que detecta estos, se considerará un choque y activará a mayores un LED y dicho choque se registrará en la nube.

De esta manera, hemos conseguido los tres objetivos para que sea factible implementar esta funcionalidad. El coste, tanto en tiempo, por tener una menor complejidad, como monetario ya que se reduce a la adquisición de unos pocos sensores. Ya no tenemos que implementar una solución completa de posicionamiento ultrasónico sino un único detector que emule un mínimo de toda esa solución. Simulamos una situación, a grandes rasgos, utilizada por los vehículos autónomos: el posicionamiento ultrasónico mediante detección de un objeto en una dirección. También simulamos una situación de choque, que, si bien en un caso realista esa información podría utilizarse para establecer una comunicación a través de Internet con los servicios de emergencia, nosotros registraremos el choque, simulado, en la nube.

5.3.2. Condiciones ambientales

Descripción

No es difícil pensar que en la conducción autónoma las condiciones ambientales del entorno son importantes. Pueden resultar cruciales para disparar ciertas funciones en el coche como bien puede ser reducir la velocidad cuando existen condiciones adversas incluso impidiendo la circulación de vehículos en caso de desastres naturales o proporcionar servicios adicionales al usuario que hace uso del coche autónomo como puede ser activar el aire acondicionado en casos de temperaturas bajas o altas.

En esta funcionalidad vamos a pretender realizar una solución contraria a la anterior, es el servidor el que recoge información y la comunica al vehículo. Como las condiciones ambientales son prácticamente iguales a lo largo del rango que puede abarcar un servidor de borde, es posible en él incorporar toda una serie de sensores ambientales, desde los de mayor a menor coste, que envíen esta información a todos los vehículos de la zona. De esta manera, el coche autónomo tendría una menor complejidad ya que no requeriría de sensores ambientales, se

trata de una información que puede estar centralizada en torno al servidor de borde. También, reducimos a la mitad la latencia ya que no se trata de una comunicación petición-respuesta sino en un solo sentido. Esta información ambiental, como ya hemos visto, se utilizará también como condicionante en órdenes que enviará el servidor de borde a los vehículos para su adecuado funcionamiento.

Adecuación a nuestros recursos

De nuevo, implementar un sistema ambiental completo tiene una elevada complejidad que se escapa del trabajo, más aún si aplicamos estos factores para realizar determinados acciones según el valor de estos factores. Sin embargo, reducir el número de sensores ambientales para hacer una acción concreta en el vehículo simulado podría servirnos para obtener las principales conclusiones de este tipo funcionalidades.

Por tanto, un registro continuo de la temperatura y humedad en el servidor de borde y, posteriormente, envíe dichos datos a una pantalla del vehículo simulado, con el fin de dar esa información al usuario del coche, ya nos resulta útil para nuestro estudio

Con esta funcionalidad básica cumplimos, de nuevo, con los tres requisitos para hacer factible la solución IoT. El coste monetario se reduce a una serie de sensores y una pantalla LCD. También, el coste en tiempo se reduce pues se adapta al horizonte temporal de este trabajo ya que reducimos la complejidad de la solución IoT. Se trata de una funcionalidad que logramos emular satisfactoriamente ya que no requiere de los complejos elementos de una solución realista pero logran emular sus rasgos básicos. Los conceptos prácticos que aprendemos son distintos a la funcionalidad anteriormente explicada y están relacionados con la tecnología que es objeto de análisis en este trabajo.

CAPÍTULO 6

¿Cómo implementar la solución?

Para poder implementar una solución *Edge Computing* es necesario cubrir aspectos de [WABN20]:

- *Hardware*
- *Software*
- *Servicio.*

En el mercado cada uno de estos segmentos es proporcionado por diferentes compañías. A continuación, veremos los principales proveedores de cada ámbito y seleccionaremos aquellos productos que utilizaremos en la solución.

6.1. Hardware

En este ámbito se proporcionan los dispositivos inteligentes para poder llevar a cabo una solución *Edge Computing*. Podemos subdividirlos en [WABN20]:

- *Sensores.* Los sensores son piezas de hardware que se encargan de proporcionar datos de una determinada magnitud física. Pueden o no convertir esta magnitud en una señal digital e incluso pueden contener hardware que envíe información cuando detecte una determinada anomalía. Por ejemplo, un sensor podría ser un medidor de temperatura.
- *Dispositivos.* Incluirían múltiples sensores y otro tipo de hardware relacionado para llevar a cabo una determinada función. Por ejemplo, un dispositivo podría ser un robot.
- *Controladores.* Se trataría de todo aquel hardware que se encargaría de correr funciones de control. Los avances en el hardware informático y las tecnologías de virtualización están permitiendo que los controladores inteligentes ejecuten sistemas operativos abiertos que puedan ejecutar aplicaciones que no son de control, como la analítica y la visualización.
- *Computadoras.* Se trata de dispositivos con un sistema operativo (por ejemplo, Windows o basado en Linux) localizados fuera de los centros de datos.
- *Centros de datos en el borde.* Se trataría de los servidores que se encargarían de realizar la computación en el borde. Por tanto, están en la misma ubicación que los sensores que generan datos.
- *Equipamiento de red.* Se trataría del hardware que permite absorber toda la cantidad de

datos que generan los dispositivos incluidos en la solución *Edge Computing*.

6.2. Software

En este ámbito se proporcionan las herramientas para recopilar, controlar y gestionar los datos que se generan en los dispositivos de borde. Podemos subdividirlos en [WABN20]:

- *Software de infraestructura*. Se trata de sistemas operativos que ejecutan las aplicaciones de borde. Pueden ser RTOS (Real Time Operating System), para ejecución en tiempo real o GPOS (General Purpose Operating Systems) los cuales ejecutan aplicaciones donde el tiempo no es un aspecto crítico.
- *Aplicaciones*. Se trata de herramientas de análisis de datos utilizadas para realizar todo tipo de funciones analíticas, desde la construcción o ejecución de modelos ML / AI hasta simples motores de reglas.
- *Plataformas*. Como vimos en apartados anteriores, las soluciones basadas en una computación exclusiva en la nube no van a ser óptimos en servicios ofrecidos por Internet de las Cosas. Conforme al crecimiento del número de dispositivos y de soluciones en tiempo real, el ancho de banda y la latencia pasan a ser parámetros importantes, las principales plataformas del procesamiento en la nube como Amazon Web Services, Microsoft Azure y Google Cloud Platform ofrecen productos enfocados a estos requisitos acercando a el procesamiento al borde, es decir, han creado productos que permiten implementar soluciones *Edge Computing*. Se trata, por tanto, de aplicaciones software que permiten desplegar y administrar las aplicaciones que se ejecutan en el borde.

6.3. Servicios

Los servicios son ofrecidos por empresas para asistir a los usuarios en tareas de diseño, implementación y operación. Existen diversos tipos de servicios, como la consultoría, que trataría de ayudar al usuario en la búsqueda de oportunidades de computación; de tipo implementación, que trataría de customizar y desarrollar soluciones *Edge Computing*; y de tipo operacional, que se encargaría de la gestión de la solución día a día [WABN20].

CAPÍTULO 7

Herramientas para el desarrollo de la solución IoT

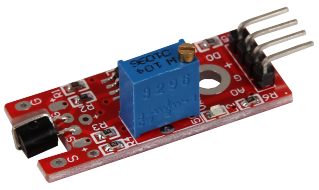
En este apartado, seleccionaremos los distintos componentes que van a conformar la solución IoT para nuestro caso de uso. Para ello, dividiremos los componentes que conforman la solución IoT de la misma forma que lo hemos explicado en el capítulo anterior, es decir, vamos a hacer una selección de nuestros componentes hardware, software y servicios más adecuados para nuestra solución. Si es preciso, compararemos las diversas alternativas que haya.

7.1. Componentes hardware

Para poder implementar la solución hardware vamos a requerir de sensores [Ele20], (que permitan recopilar información sobre proximidad, contacto, humedad y temperatura), dispositivos de visualización (que nos permitirán percibir el funcionamiento de la solución IoT), de un controlador (que centralice la información del coche emulado para enviarla conjuntamente al servidor de borde) y de un centro de datos que actúe como servidor de borde realizando un determinado procesado para poder llevar a cabo las funcionalidades mencionadas en el Capítulo 5.

7.1.1. Sensores y dispositivos de visualización

Mental touch sensor (KY-036). Este sensor emite una señal cuando se toca la zona metálica del sensor. Es posible ajustar la sensibilidad y así ajustar el valor especifico en el que la señal debe caer para que este sensor detecte contacto. Posee dos LEDs, uno para mostrar que está siendo alimentado y el otro se enciende cuando el sensor detecta contacto. Posee 2 pines para alimentar el sensor y otros dos para generar una señal analógica y digital cuando existe contacto.



Sensor ultrasónico (HC-SR04). Este sensor permite medir la distancia mediante ultrasonido desde los dos a los cuatrocientos cincuenta centímetros a través de la siguiente formula que permite calcular la distancia a partir del tiempo que tarda la onda ultrasónica en rebotar en el objeto, dentro del rango, y regresar al sensor. Posee dos pines para alimentar el sensor y otros dos, denominados *echo* y *trigger*, para poder determinar el momento de llegada y emisión del ultrasonido respectivamente para así poder calcular el tiempo que tarda en regresar el ultrasonido.

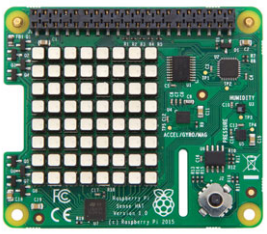
$$Distancia[cm] = 0,034[cm/s] * (tiempo/2)[s]$$

Buzzer pasivo. Este sensor permite emitir sonidos a una determinada frecuencia que depende de la frecuencia de la señal de entrada.

Diodo emisor de luz (LED). Nos permitirá emitir luz en el momento que apliquemos una determinada tensión entre sus dos pines. Añadiremos una resistencia para evitar que se queme.

Pantalla alfanumérica LCD 16x2. Este dispositivo permite visualizar todo tipo de caracteres en dos líneas cada carácter se codifica mediante una determinada combinación de los numerosos pines que posee.

Sense HAT add-on Board para Raspberry Pi. Se trata de una tarjeta que se conecta y controla con los cuarenta pines de la Raspberry Pi y que contiene un giroscopio, acelerómetro, magnetómetro, barómetro, termómetro, sensor de humedad así como un joystick y una matriz LED 8x8.



7.1.2. Controladores

Para emular las funcionalidades propuestas del controlador de un coche autónomo, salvando las distancias referidas al poder computacional de este, se va a usar un microcontrolador Arduino Uno R3. Este sencillo microcontrolador permite instalar un programa en el cual se basarán sus actuaciones. Estas actuaciones se limitan, principalmente, en la conexión y posterior comunicación con sensores a través de sus pines, mostrados en la Figura 7.1.

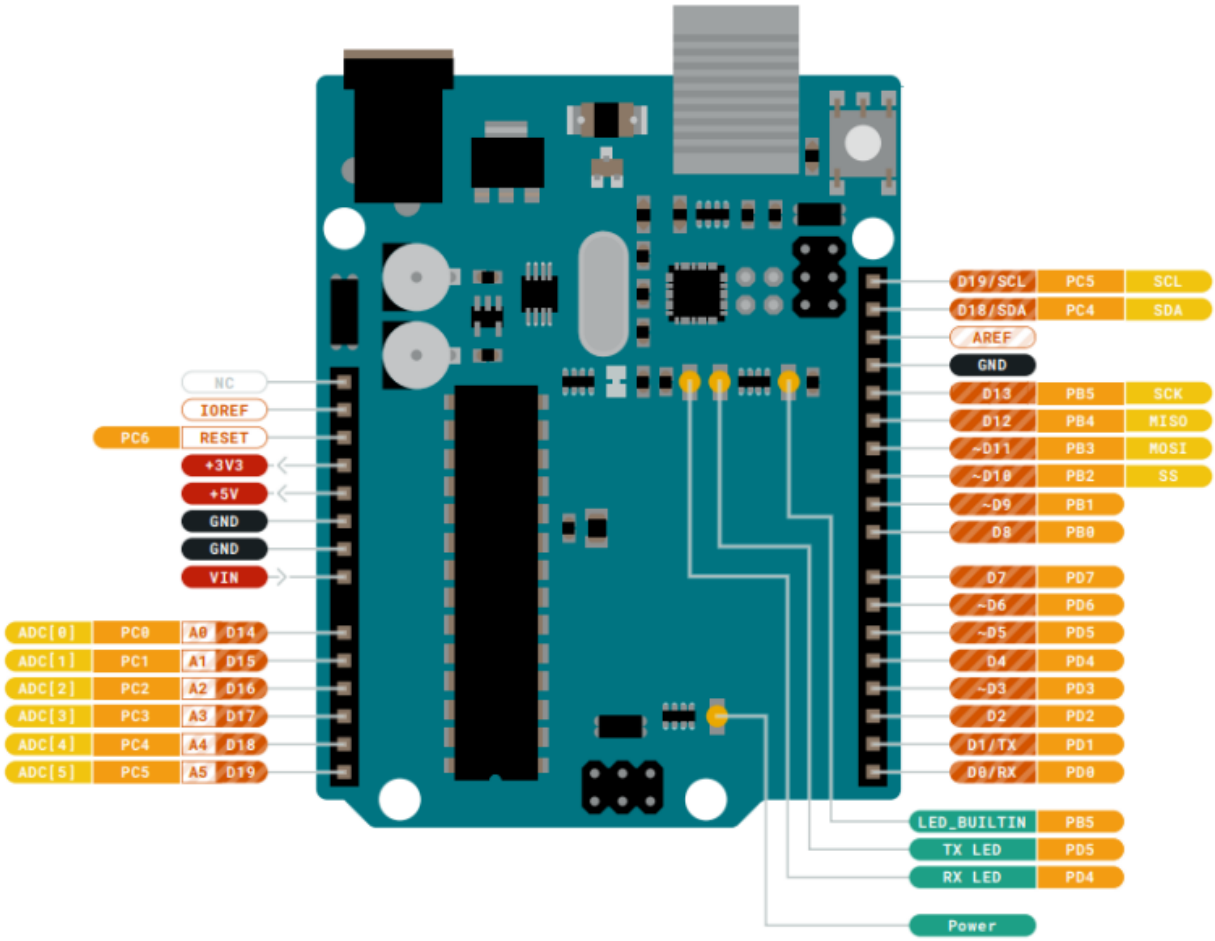


Figura 7.1: Pines de Arduino. Tomado de [Ard22]

7.1.3. Centro de datos en el borde

El dispositivo que va a actuar como servidor de borde y que, por tanto, se encargará de dar las correctas instrucciones al microcontrolador, de acuerdo con las funcionalidades que son objeto de desarrollo, será una Raspberry Pi 3B la cual puede verse en la Figura 7.2. También se encargará de comunicaciones con la nube en ciertos casos. Debe también permitir tener incorporados ciertos sensores ambientales.

Este dispositivo dispone de pines para incorporar ciertos sensores, se puede comunicar con el microcontrolador Arduino a través de una comunicación serie cableada y lo más importante, puede centralizar y distribuir la información, tanto a la nube, a través de Internet, como al microcontrolador Arduino, y procesarla para actuar debidamente. Su capacidad computacional es suficiente para llevar a cabo estas acciones.

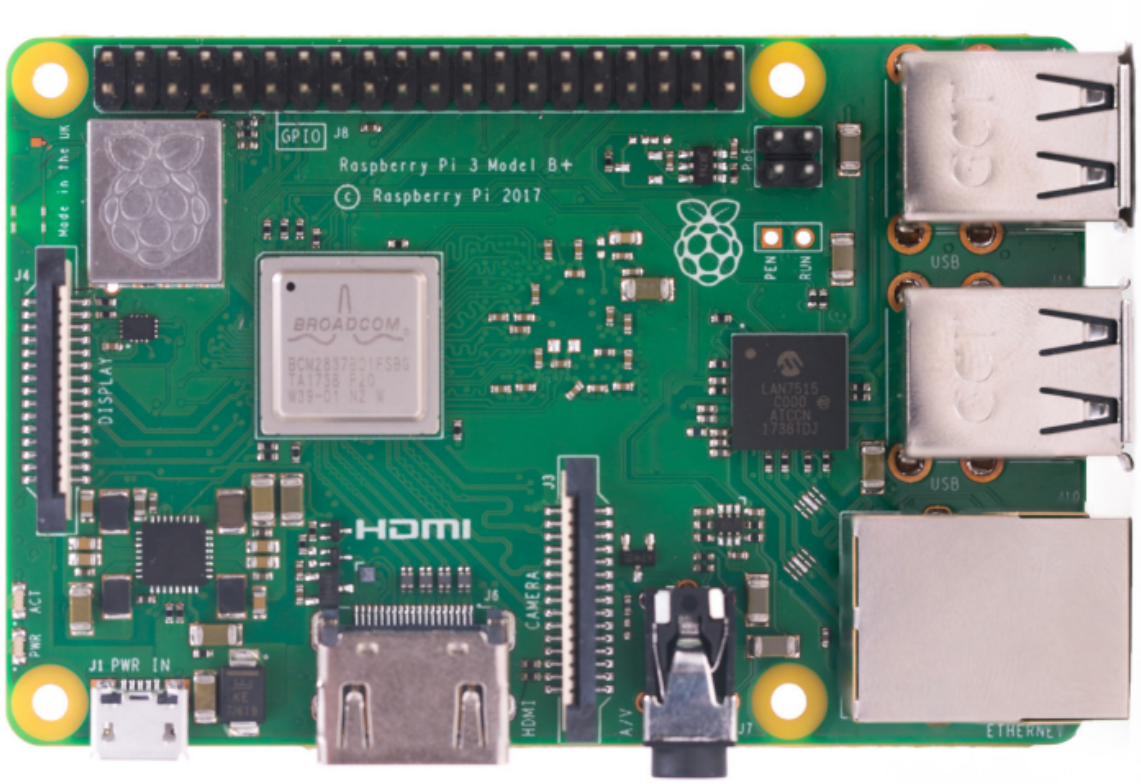


Figura 7.2: Raspberry 3B. Tomado de [Ras]

7.2. Componentes software

Para llevar a cabo nuestra solución IoT utilizaremos una plataforma que nos permita administrar y desplegar las aplicaciones que se ejecutan en el borde. El mercado de las plataformas en la nube está dominado por tres grandes plataformas:

- Amazon Web Services [Ama20]
- Microsoft Azure [Mic]
- Google Cloud [Goo]

Pero ... ¿No habíamos visto que la computación en la nube no era válida para una solución *Edge Computing*? Cierto es, pero estas plataformas además de proporcionar servicios de computación en la nube, debido a la necesidad de acercar los recursos de computo al borde, por el creciente impacto de los dispositivos IoT, nos proporcionan ciertos servicios que son útiles para conectar e interactuar con dispositivos IoT o incluso servicios específicos para hacer un despliegue de una solución *Edge Computing*. No obstante, esto no quiere decir que no podamos utilizar la computación en la nube para nuestra solución sino que nuestra solución debe lograr que:

- Los dispositivos que requieran una respuesta rápida, es decir, con requisitos de retardo, sean atendidos por el servidor de borde.
- El procesado básico pueda realizarse en el borde evitando así un consumo elevado del ancho de banda del núcleo de la red.
- Aquellos datos útiles para un procesado más complejo, y que no requieran de una respuesta

rápida, pueden enviarse a la nube.

De esta manera, podemos reiterar que la arquitectura *Edge Computing* no excluye al procesamiento en la nube sino que, al revés, se complementan para lograr dar un mejor servicio al usuario

Analizaremos tres servicios, que pueden servirnos de utilidad en la implementación de nuestra solución IoT. Todos ellos son proporcionados por las plataformas anteriormente mencionadas, los compararemos y de esta manera, tendremos el candidato idóneo para el desarrollo del prototipo.

- AWS IoT Greengrass
- Azure IoT Edge
- Google IoT Core

7.2.1. El protocolo MQ Telemetry Transport

Puesto que posteriormente vamos a utilizar el protocolo MQ Telemetry Transport (MQTT) para la comunicación con la nube, es preciso hacer aquí un inciso y describir, a grandes rasgos, su funcionamiento.

Definición

MQTT es un protocolo M2M (machine to machine) y permite comunicar a los dispositivos entre ellos de forma inalámbrica. Es considerado uno de los principales pilares de IoT por su sencillez y ligereza, requisitos indispensables para trabajar con dispositivos IoT.

Funcionamiento

Se trata de un servicio de mensajería de tipo publicador/suscriptor. Los mensajes se envían junto con una etiqueta que determina el *topic* (tema) al que pertenecen. De esta manera, un dispositivo puede enviar un mensaje según un determinado tema y otro tipo de dispositivos se pueden suscribir a un determinado tipo de temas y recibir, únicamente, los mensajes de esos determinados temas. Es necesario un intermediario, denominado broker, entre el publicador y el subscriptor que se encargue de determinar qué mensajes de un publicador deben llegar al suscriptor. Esto viene determinado, de nuevo, por el tema del mensaje y el tema del que quieres recibir mensajes.

No es necesario una conexión entre subscriptor y publicador, los dispositivos que quieren publicar o recibir mensajes de uno o varios temas se deben conectar al broker con un mensaje de tipo CONNECT y recibirán del broker un mensaje de confirmación CONNACK. Para suscribirse a un tema se debe enviar un mensaje SUSCRIBE con el nombre del tema y posteriormente se recibirá del broker la confirmación SUBACK. Para publicar datos se envían mensajes de tipo PUBLISH [mqt15].

Para mantener y asegurar una conexión activa, se envían periódicamente mensajes PINGREQ y para desconectarse se envía un mensaje DISCONNECT. Un esquema del funcionamiento puede verse en la Figura 7.3.

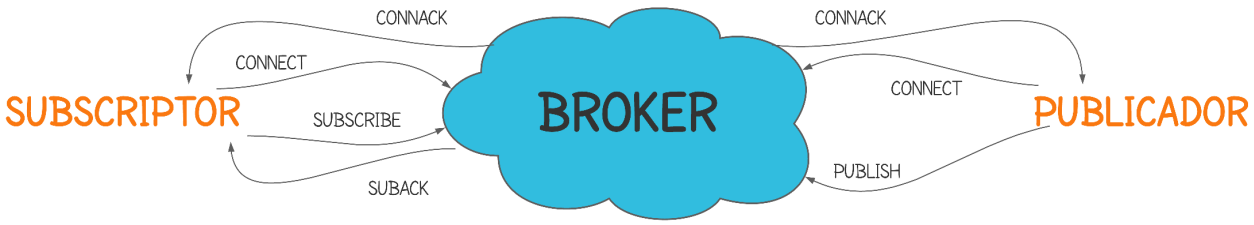


Figura 7.3: Funcionamiento MQTT

Para nuestro caso realizaremos una comunicación a través de este protocolo entre los sensores y el servicio de AWS IoT, el servidor de borde actuará como broker.

7.2.2. AWS IoT Greengrass

Funcionamiento básico

AWS IoT Greengrass permite comunicar a los dispositivos entre sí aunque no exista conexión a Internet. Los dispositivos que trabajan conjuntamente se agrupan en un "Greengrass group". Estos grupos son definidos y configurados en la nube. Todo Greengrass group necesita de un "Greengrass Core". Un Greengrass Core es un dispositivo físico que dispone de una CPU de 64 bits que ejecuta un sistema operativo Linux al que se le instala un "Greengrass Core software" que permitirá a este dispositivo conectarse de forma segura con AWS. Este Greengrass Core es el dispositivo central y por tanto, es el que actúa como servidor de borde. Podemos hacer despliegues desde la nube para modificar la configuración del sistema [Ama20].

En la Figura 7.4 se puede observar un pequeño esquema que refleja que el servicio AWS IoT Greengrass ha instalado su software "Greengrass Core" en un dispositivo lo que le permite a este realizar tareas de procesamiento local y mensajería, entre otras, sobre los dispositivos o sensores conectados a este sin necesidad de que haya conectividad a Internet. Este dispositivo está conectado al servicio en la nube AWS IoT Greengrass Cloud Service de forma segura. Cuando haya conexión a Internet, Greengrass Core podrá comunicarse con este servicio en la nube sin tener que preocuparse por la seguridad en la comunicación.

Por tanto, Greengrass Core permite ejecutar de manera local código. Interactúa directamente con la nube y trabaja localmente aun no habiendo conexión a Internet, requisito básico que debemos requerir en una solución IoT basada en *Edge Computing*. Además, la ejecución cercana del código va a permitir una comunicación con el servidor de borde con poco retardo y podremos enviar a la nube solo la información que sea pertinente, reduciendo así el ancho de banda utilizado en la conexión a Internet [Ama20].

La ejecución de código de forma local se hace a través de componentes. Los componentes son módulos software que se implementan y ejecutan en el dispositivo central, es decir, nuestro servidor de borde. Todo el software que pretendamos implementar en un servidor de borde a través de AWS debe hacerse a través de componentes.

Aws IoT Greengrass ofrece una variedad de componentes públicos como desarrollados por la comunidad. También existe la posibilidad de desarrollar componentes propios. Un componente, para poder ejecutarse necesita de [Ama20]:

- *Receta*. Es un fichero en formato JSON que describe el software que se va a ejecutar,

permite realizar ciertas configuraciones cuando el componente se encuentra en un determinado estado y proporciona parámetros al componente.

- *Artefacto*. Es el código fuente, binarios o scripts que definen el software que se ejecutará en el dispositivo.
- *Dependencias*. Para definir relaciones entre componentes para aplicar actualizaciones automáticas o reinicios de componentes dependientes.

Podemos retransmitir mensajes MQTT y otros datos entre dispositivos que tienen instalado el software "Greengrass Core" y el servicio en la nube AWS IoT Core. Para poder llevar a cabo esto deberemos instalar las librerías "awsiotsdk" en el código del componente deseado que proporcionan las funciones necesarias para una comunicación mediante este protocolo. Y como la conexión entre AWS IoT Core y el dispositivo con el software "Greengrass Core" es segura, no debemos de preocuparnos por velar por la confidencialidad e integridad de los datos intercambiados.

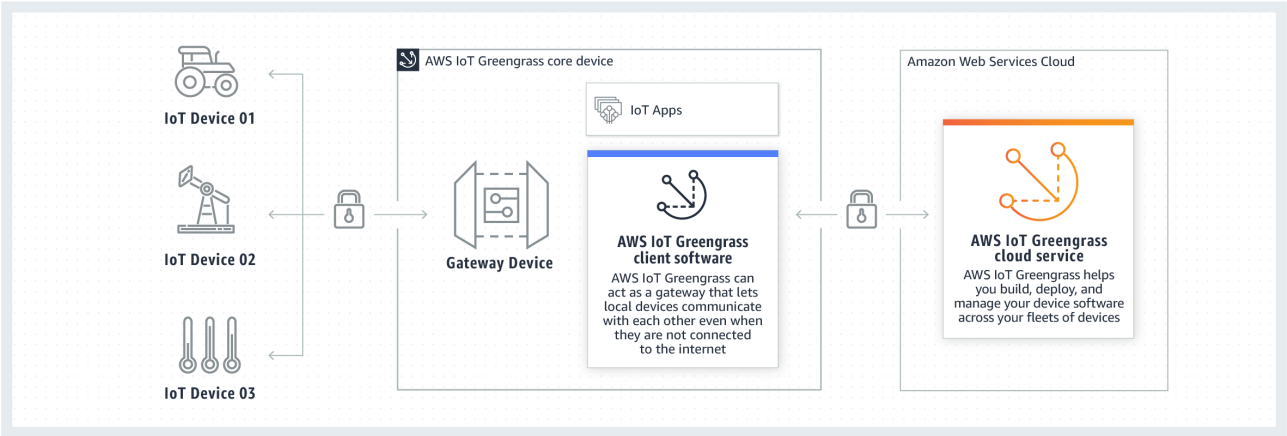


Figura 7.4: Ejemplo de una solución IoT mediante AWS IoT Greengrass, tomado de *[Ama20]*

Valoración

Podemos asegurar que AWS IoT Greengrass es apto para desarrollar y desplegar una solución IoT basada en *Edge Computing* por las siguientes razones:

- Se puede establecer un dispositivo central encargado del funcionamiento local de la solución o lo que es lo mismo, podemos hacer procesamientos en el borde a través del Greengrass Core. Esta arquitectura, con un servidor localizado cerca de los sensores, potencialmente puede garantizar una latencia máxima.
- El funcionamiento local de los diferentes dispositivos no depende de la conexión a Internet, algo que resulta esencial si pretendemos dar un servicio robusto ante caídas en la conexión a Internet. Por lo general, servicios que requieren de poca latencia necesitan ser servidos aún cuando no haya una conexión a Internet.

  Imaginémonos una fabrica cuyos sensores de las distintas máquinas son gestionados por el servidor de borde. Si el servicio no fuese robusto a caídas de Internet la fábrica podría sufrir grandes pérdidas por paradas en su producción.
- La mayor parte de las comunicaciones se hace de forma local no llegando a que la mayor parte de los datos lleguen a la nube. Logramos así reducir la carga en el núcleo de la red

y delegándola al borde.

- Cuando tengamos conexión a Internet podremos enviar datos a la nube para procesamientos más complejos sin requisitos de latencia.
- Además, de forma gratuita podemos desplegar hasta tres Greengrass Core, lo que nos resulta suficiente para las funcionalidades que pretendemos implantar.

7.2.3. Azure IoT Edge

Funcionamiento básico

Azure IoT Edge proporciona un servicio que se ejecuta en la nube y otro que se ejecuta en el entorno local, es decir en el borde. El servicio de ejecución local se encarga de iniciar y administrar los flujos de trabajo en el dispositivo. Estos flujos de trabajo constan de un conjunto de contenedores vinculados en un orden específico para crear un escenario a extremo a extremo. Desde la nube, a través de IoT Hub, podemos administrar y desarrollar las cargas de trabajo en el borde. Las cargas de trabajo se trata de módulos implementados mediante contenedores compatibles con Docker y contienen el código de ejecución necesario [Mic]. En la Figura 7.5 se muestra, a grandes rasgos, un ejemplo de arquitectura de una solución IoT basada en Azure IoT Edge.

En definitiva, Azure IoT Edge esta formado principalmente por tres componentes [Mic]:

- *Módulos*. Unidades de ejecución implementadas como contenedores. Es módulos ejecutan lógica de negocio en los dispositivos de borde. Estos módulos pueden ejecutar código diseñado por nosotros, código ajeno o proporcionado por Azure. Los módulos pueden comunicarse entre sí y pueden ejecutarse sin conexión a Internet. Estos módulos se despliegan en dispositivos IoT Edge que pueden actuar como puertas de enlace entre el borde y la nube. Por tanto pueden actuar como servidores de borde.
- *Entorno de ejecución*. Se ejecuta en ejecuta en la totalidad de los entornos de IoT Edge y administra el entorno de ejecución y la comunicación de los módulos implementados en cada dispositivo
- *Interfaz basada en la nube*. Permite supervisar y administrar los dispositivos de IoT Edge de forma remota.

Los módulos en Azure IoT Edge están formados por [Mic]:

- *Imagen*. Contienen las aplicaciones que aprovechan las características de administración, seguridad y comunicación de la instancia de IoT Edge en tiempo de ejecución. Se pueden desarrollar o exportar desde un servicio de Azure compatible. Las imágenes se encuentran en la nube y pueden actualizarse, cambiarse e implementarse en diferentes soluciones.
- *Instancia*. Es la unidad específica del cálculo que ejecuta la imagen de módulo en un dispositivo IoT Edge. La instancia de módulo se inicia mediante la instancia de IoT Edge en tiempo de ejecución.

  Cada vez que una imagen de módulo se implementa en un dispositivo y se inicia mediante la instancia de IoT Edge en tiempo de ejecución, se crea una instancia de ese módulo. Dos dispositivos en distintas partes del mundo podrían usar la misma imagen de módulo. Sin embargo, cada dispositivo tendría su propia instancia de módulo cuando este se inicie

en el dispositivo.

- *Identidad*. Fragmento de información almacenada en IoT Hub que está asociada a cada instancia de módulo.
- *Módulo gemelo*. Es un documento JSON almacenado en IoT Hub, que contiene información de los estados de una instancia de módulo, incluidos los metadatos, las configuraciones y las condiciones.

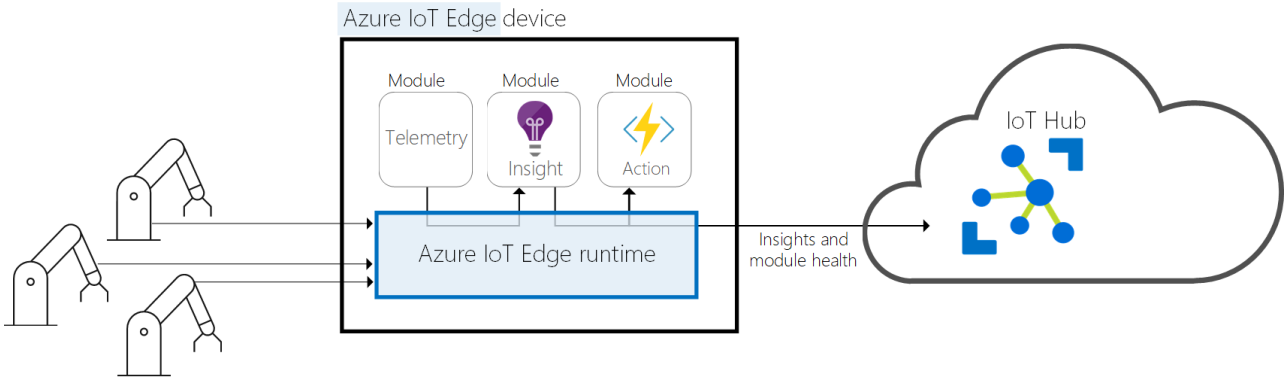


Figura 7.5: Ejemplo de una solución IoT mediante Azure IoT Edge, tomado de *[Mic]*

Valoración

Al igual que con AWS IoT Greengrass, podemos asegurar que Azure IoT Edge permite desarrollar e implementar una solución IoT basada en *Edge Computing* por razones similares:

- Se puede establecer un dispositivo central encargado del funcionamiento local de la solución o lo que es lo mismo, podemos hacer procesamientos en el borde a través de un dispositivo IoT Edge con lo que potencialmente puede garantizar una latencia máxima.
- El funcionamiento local de los diferentes dispositivos no depende de la conexión a Internet gracias a que los modulos se pueden ejecutar sin esa necesidad. Esta característica resulta esencial si pretendemos dar un servicio robusto ante caídas en la conexión a Internet. Como comentábamos con AWS IoT Greengrass, servicios que requieren de poca latencia necesitan ser servidos aún cuando no haya una conexión a Internet.
- La mayor parte de las comunicaciones se pueden hacer de forma local si los módulos que desplegamos no requieren de unas comunicación con Azure IoT Hub.
- Cuando tengamos conexión a Internet, e implementados módulos con comunicación a Azure IoT Hub, podremos enviar datos a la nube para procesamientos más complejos sin requisitos de latencia.
- Además, de forma gratuita podemos tener un número indeterminado de dispositivos IoT Edge. Sin embargo, gratuitamente solo podremos enviar 8000 mensajes al mes a la nube. Además, ciertos módulos que proporciona Azure IoT Edge son de pago.

7.2.4. Google IoT Core

Funcionamiento básico

Google IoT Core es un servicio que nos permite conectar, gestionar e ingerir datos de los dispositivos que conectemos. Las comunicaciones de estos dispositivos con Google Cloud Platform se realizan a través de los protocolos MQTT y HTTP. Los datos de un determinado tema activan ciertas funciones en la nube para su procesado. Por tanto, la arquitectura publicador/subscriptor de MQTT permite diferenciar diferentes tipos de mensajes para que se gestión y procesen de un manera determinada. En la Figura 7.6 se muestra, a grandes rasgos, un ejemplo de arquitectura de una solución IoT basada en Google IoT Core. Se puede decir que la arquitectura de Google IoT Core se centra en dos componentes [Goo]:

- *Administrador de dispositivos.* Para registrar los dispositivos en el servicio para administrarlos y configurarlos.
- *Dos puentes de protocolo.* HTTP y MQTT que son utilizados para conectarse a Google Cloud Platform.

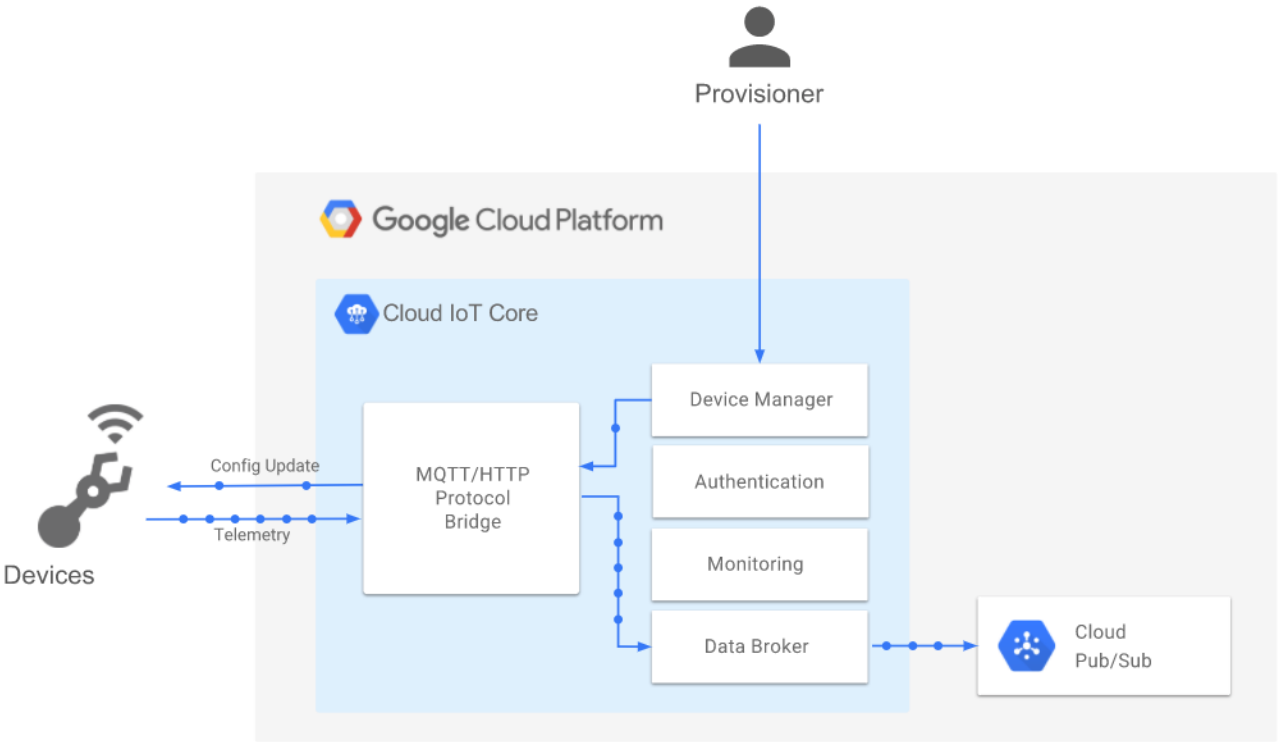


Figura 7.6: Ejemplo de una solución IoT mediante Google IoT Core, tomado de *[Goo]*

Valoración

A diferencia de los dos servicios anteriores, Google IoT Core no es un servicio apto para desarrollar e implementar una solución basada en *Edge Computing* por los siguientes motivos:

- No hay posibilidad de disponer de un dispositivo central en el entorno local al que se le pueda incluir un determinado código para atender peticiones de los dispositivos que se conectasen a él. Esto supone que los diversos dispositivos se conecten directamente a la

nube provocando elevadas tasas de transmisión hacia la nube y no podemos garantizar un retardo máximo.

- El funcionamiento de los dispositivos registrados va a depender en buena medida de la calidad de la conexión a Internet que tengamos.

7.2.5. Conclusiones

Nuestro debate sobre qué plataforma resulta más conveniente va a analizarse desde dos perspectivas distintas tomando en cuenta que nuestra solución no va a ser muy ambiciosa. El debate se va a centrar sobre si utilizar la plataforma de Azure o AWS ya que Google no proporciona un servicio adaptado a la tecnología *Edge Computing* y por tanto, no va a resultarnos útil para la elaboración de nuestro prototipo.

Complejidad

La implementación de código en los dispositivos que actúan como servidor de borde (Greengrass Core), a través de componentes, es muy simple. Solo requiere conocimientos de programación relativos a la configuración de ficheros JSON, crear scripts en un lenguaje de programación y saber utilizar las librerías adecuadas para establecer comunicaciones con el servicio IoT Greengrass.

Sin embargo, la implementación de código a través de Azure puede resultar más costoso de realizar por la necesidad de tener conocimientos relativos al uso de contenedores. Esos son conocimientos relativamente complejos que podemos evitar usando AWS IoT Greengrass a través de componentes. Probablemente, el uso de contenedores sea más conveniente para cierto tipo de soluciones IoT ambiciosas pero para nuestro proyecto creemos que nos basta con el desarrollo de scripts.

Coste del servicio

Nuestra solución solo requerirá del uso de un solo servidor de borde por lo que desde este punto de vista tanto AWS como Azure nos lo permiten hacer forma gratuita. Cierto tipo de servicios que queramos usar en la nube tambien están exentos de pago hasta un determinado límite. Por ejemplo, los primeros 5 GB para almacenamiento de datos seguro en la nube, a través del servicio AWS S3, son gratuitos. No existen componentes de AWS de pago.

Sin embargo, Azure impone otra serie de restricciones presupuestarias mayores ya que solo podremos enviar 8000 mensajes al mes a la nube y ciertos módulos que puede que tengamos que implementar son de pago. Por ejemplo, el módulo Azure SQL Database requiere de una licencia de pago y por tanto, nuestra solución, si quisiésemos almacenar datos en el servidor de borde no podría hacerse sin el pago de la licencia correspondiente. Servicios de almacenamiento que se proporcionan en la nube son de pago para cualquier nivel de utilización.

Documentación

La documentación ha sido otro factor que he tomado en cuenta. A diferencia de la documentación de Microsoft Azure IoT Edge, AWS IoT Greengrass posee un apartado en

su documentación que explica en que consiste el servicio y a la hora de iniciarse, permite comprender mejor este.

En conclusión, creemos que la plataforma que mejor se adapta a nuestras necesidades, presupuesto y conocimiento para llevar a cabo nuestra solución IoT basada en *Edge Computing* es AWS IoT Greengrass por lo que será AWS la plataforma con la que desarrollaremos nuestro proyecto.

7.3. Servicios

No vamos a hacer uso de ningún servicio proporcionado por alguna empresa. Al fin y al cabo, el trabajo, como uno de los objetivos principales, consiste en proporcionar un servicio completo sobre una solución IoT para llevar a cabo varias funcionalidades relacionadas con *Edge Computing*. Es por tanto, labor del autor de este trabajo elaborar un adecuado servicio tanto de diseño como de implementación y gestión de la solución IoT.

CAPÍTULO 8

Diseño de la solución IoT

En este capítulo dividiremos el diseño de la solución IoT en dos partes: un diseño hardware y otro software.

Para el diseño de la solución hardware, veremos cómo conectamos todos los componentes hardware, es decir, los sensores, el microcontrolador Arduino y la Raspberry Pi para que potencialmente puedan desarrollar las funcionalidades que se han pensado.

En el diseño software se pensará, mediante diagramas de flujo, cómo implementar los diversos códigos para que los distintos agentes que participan en la solución actúen de manera adecuada para obtener los resultados previstos.

8.1. Diseño hardware

Recordemos que el microcontrolador Arduino viene a emular el controlador de un coche y que, por tanto, deben situarse aquí los componentes que deberían colocarse en un coche. Los sensores de contacto y proximidad calcularán distancia y confirmación de choque o no para, posteriormente, enviarlo al servidor de borde. Los dispositivos de aviso, LED y buzzer, vienen a representar las indicaciones que el servidor manda al microcontrolador según distancia y posible contacto. La pantalla LCD proporcionará la temperatura y humedad que el servidor de borde recopila. La conexión de estos dispositivos se representa en la Figura 8.1.

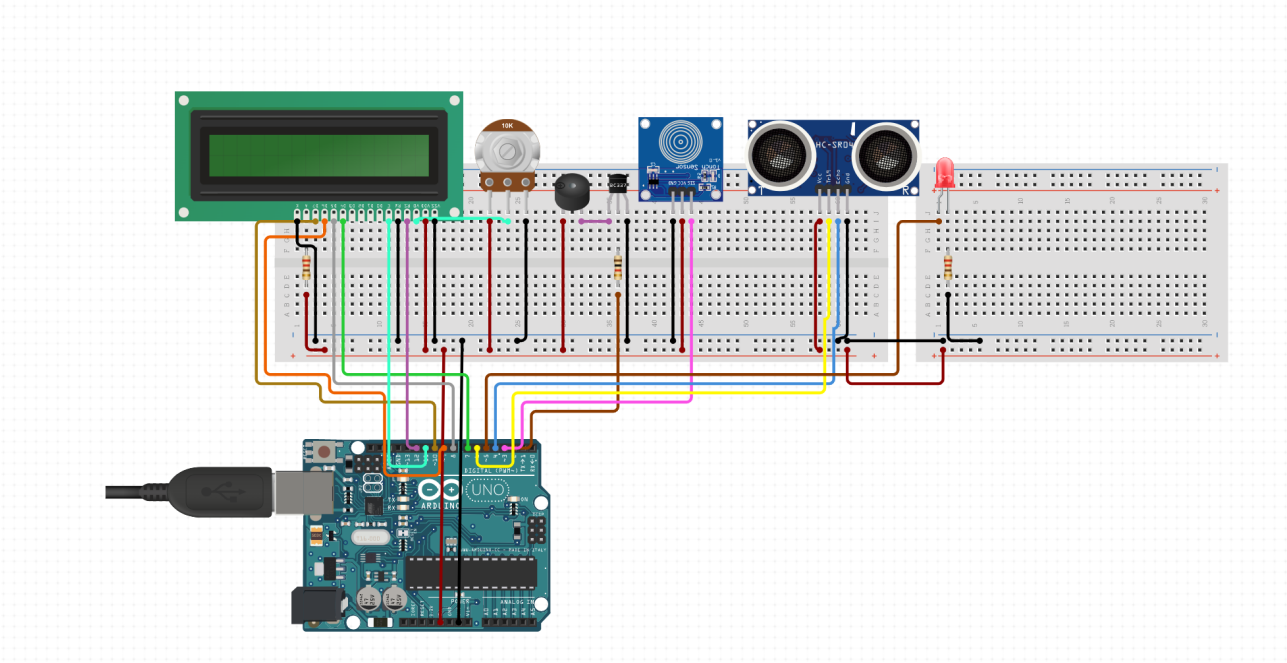


Figura 8.1: Componentes en el microcontrolador Arduino

Tan solo quedaría incorporar en la Raspberry Pi (servidor de borde) los sensores de proximidad y temperatura, a través de la placa Sense HAT tal y como podemos ver en la Figura 8.2, para poder medir dichos parámetros y enviarlos al microcontrolador. Finalmente, tanto la Raspberry Pi como el microcontrolador se conectarán de forma cableada para intercambiar información. Aunque habría sido más realista una comunicación inalámbrica entre el dispositivo embarcado en el automóvil (Arduino) y el servidor de borde, se ha implementado de forma cableada por simplicidad.

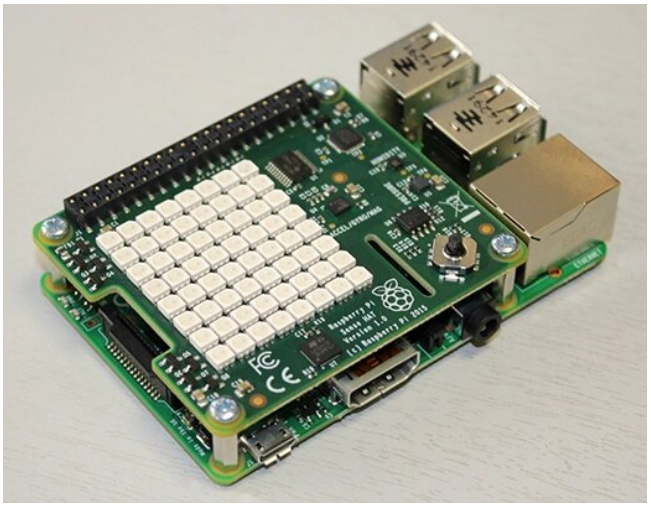


Figura 8.2: Componentes en el servidor de borde

En la Figura 8.3 podemos ver un esquema de cómo queda la arquitectura que pretendemos implementar. En el borde, la Raspberry Pi actúa como servidor instalando para ello el software IoT Core. Este servidor se conecta al microcontrolador y mediante la conexión cableada se realizará el intercambio de información. Si en algún momento ha habido un choque, además de responder el servidor al microcontrolador, se registrará el choque en la nube a través del servicio AWS S3. Para realizar la comunicación entre el servidor de borde y el servicio AWS

IoT Greengrass se utilizará el protocolo MQTT.

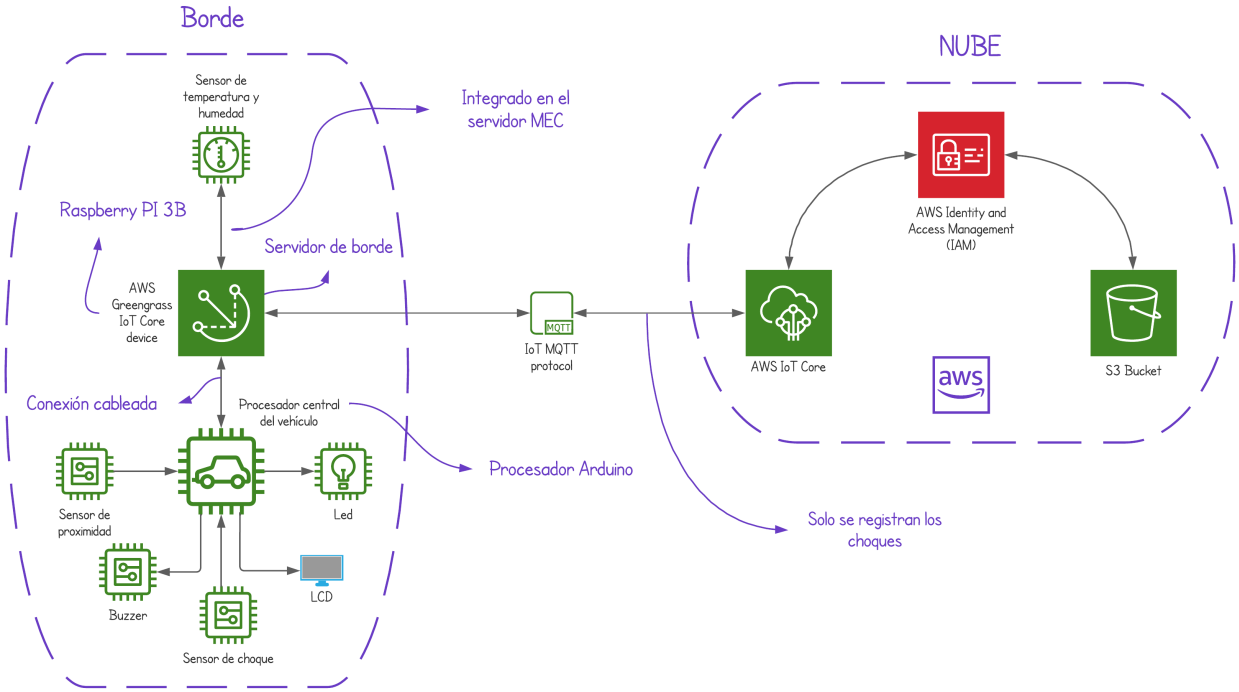


Figura 8.3: Arquitectura de la solución

8.2. Diseño software

En esta etapa vamos a realizar una primera aproximación de lo que posteriormente será la lógica del programa que desarrollaremos. Hay que tomar en cuenta que cada funcionalidad hay que verla desde 2 puntos de vista, que son la del servidor de borde y el microcontrolador.

8.2.1. Detección de proximidad y choque

Para la funcionalidad de la detección de proximidad y choque, cuyos diagramas de flujo se presentan en las Figuras 8.4 y 8.5, lo que primero se hace es medir la distancia a través del sensor de ultrasonidos. A su vez se evalúa en el sensor de contacto si lo ha habido o no. Se envía tanto la distancia como la existencia o no de contacto al servidor de borde. El servidor de borde por su parte va a evaluar de la siguiente manera:

- Si la distancia es mayor 50 cm, se enviarán *buzzerDelayValue* (cadencia de emisión del sonido del buzzer), *alarmDelayValue* (cadencia de emisión de luz del LED) y *frequency* (frecuencia del sonido que se emite por el buzzer) a 0 lo que se será interpretado por el microcontrolador como inexistencia de objetos cercanos.
- Si la distancia está comprendida entre los 30 y 50 cm, se enviará una frecuencia grave con una cadencia baja del buzzer (*buzzerDelayValue*). La alarma de contacto no se activa ($alarmDelayValue = 0$)
- Si la distancia está comprendida entre los 10 y 30 cm, no se activa la alarma, la cadencia del buzzer pasa a ser más elevada y su frecuencia algo más aguda.

- Si la distancia es menor a 10 cm pero no hay contacto, no se activa la alarma pero el buzzer pasará a emitir una frecuencia aguda y constante.
- Si la distancia es menor a 10 cm y hay contacto, se activa la alarma (*alarmDelayValue* distinto de 0) y el buzzer emitirá una frecuencia muy aguda y constante. Se registrará en el almacenamiento en la nube este suceso ya que se considerará que ha habido un choque.

Una vez recibidos estos valores (*frequency*, *alarmDelayValue* y *buzzerDelayValue*) son enviados por el servidor de borde al microcontrolador, este último activará, o no, la alarma de proximidad y choque con los parámetros recibidos.

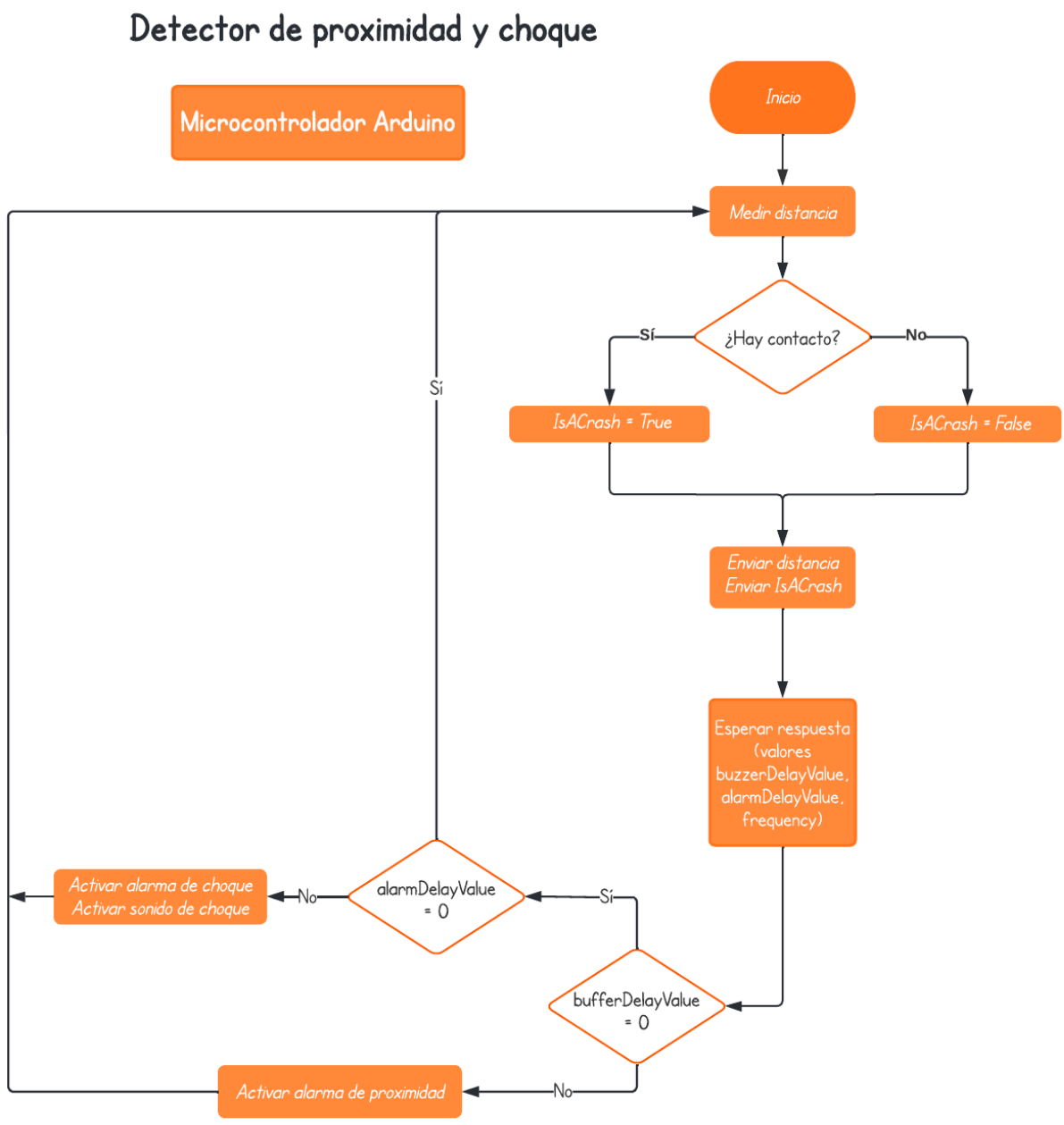


Figura 8.4: Diagrama de flujo funcionalidad proximidad y choque: punto de vista microcontrolador

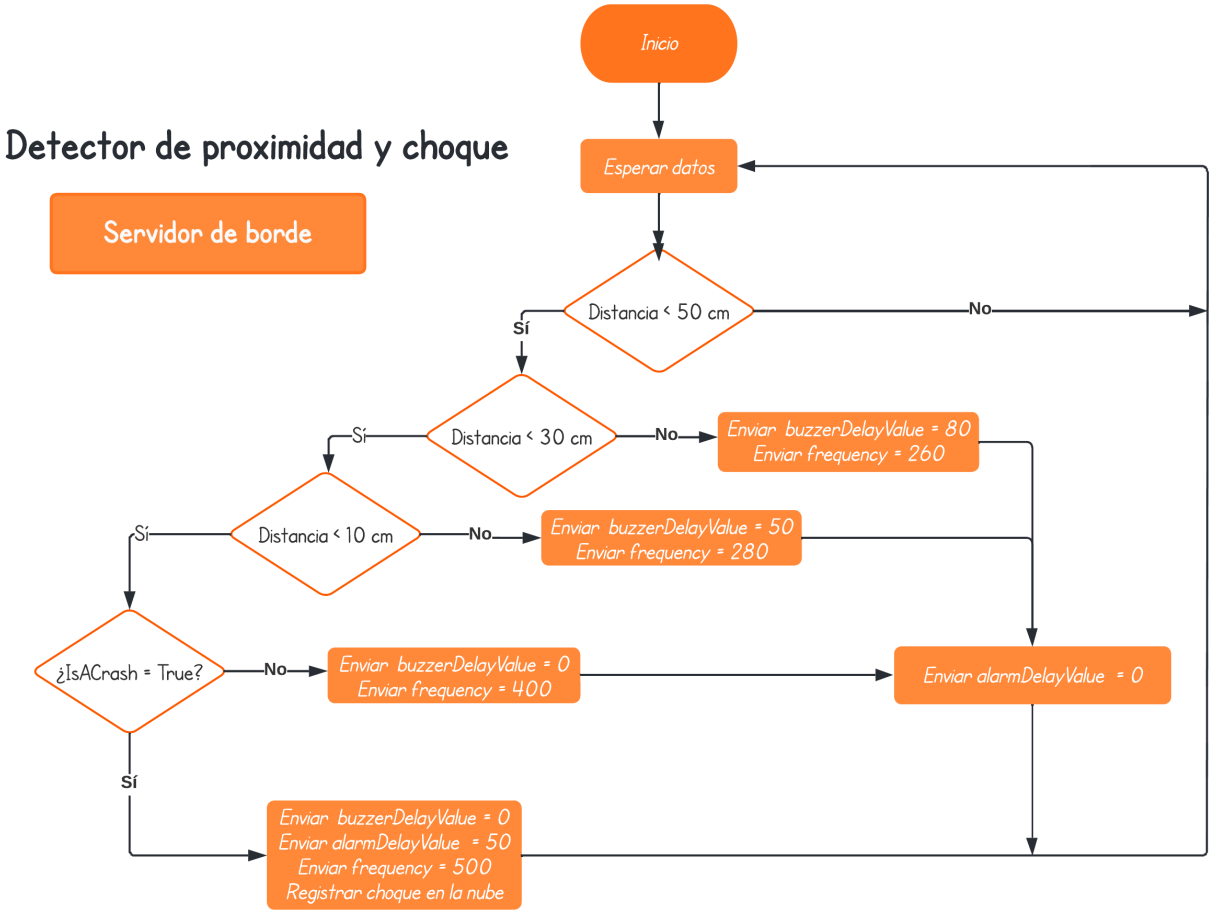


Figura 8.5: Diagrama de flujo funcionalidad proximidad y choque: punto de vista servidor de borde

8.2.2. Comunicación ambiental

Por otro lado, la funcionalidad de emisión de carácter ambiental, cuyos diagramas de flujo se presentan en las Figuras 8.6 y 8.7, pretende que el servidor de borde recopile información sobre temperatura y humedad en el momento que llegue una petición del microcontrolador. Esta petición del microcontrolador es la que recibe el servidor de borde con la distancia y estado de choque o no. En la respuesta que da el servidor de borde con los parámetros *frequency*, *alarmDelayValue* y *buzzerDelayValue*, se añadirán los parámetros de temperatura y humedad y el microcontrolador ya se encargará de asignar cada información donde corresponde. Esto se hace así porque el microcontrolador Arduino no permite que en él haya varios procesos activos, sino uno único, de forma que si tratásemos de hacer comunicaciones diferentes para cada funcionalidad, una quedaría inactiva mientras la otra se procesa. La solución de incluir en la comunicación ambas informaciones es la solución más factible ya que no bloquearía las funcionalidades y no afecta en los objetivos que pretendemos conseguir en este trabajo. Una vez que el microcontrolador recibe estos dos parámetros ambientales, se muestran en el LCD situado en nuestro "coche emulado".

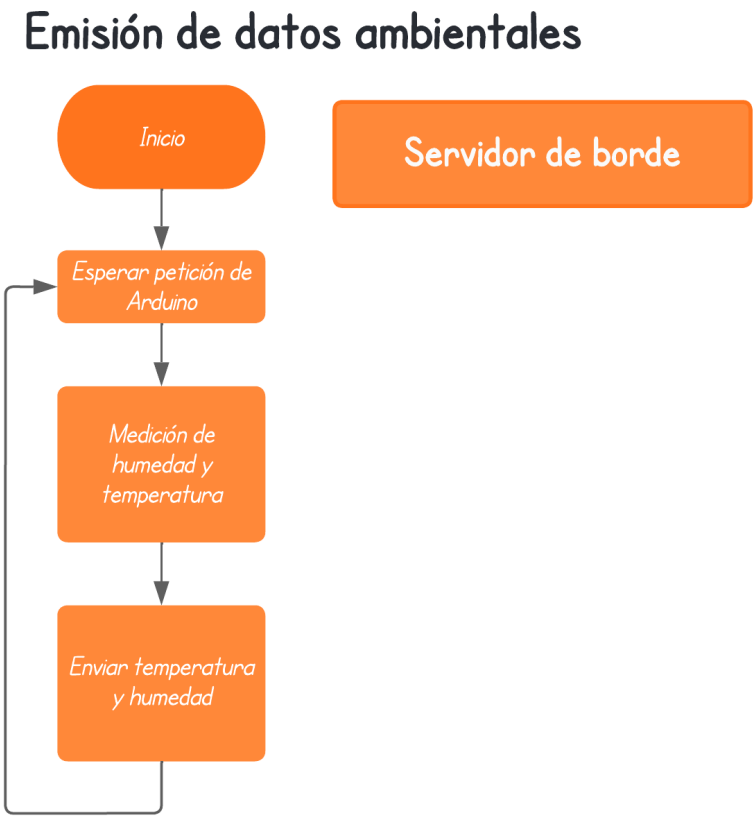


Figura 8.6: Diagrama de flujo funcionalidad proximidad y choque: punto de vista servidor de borde

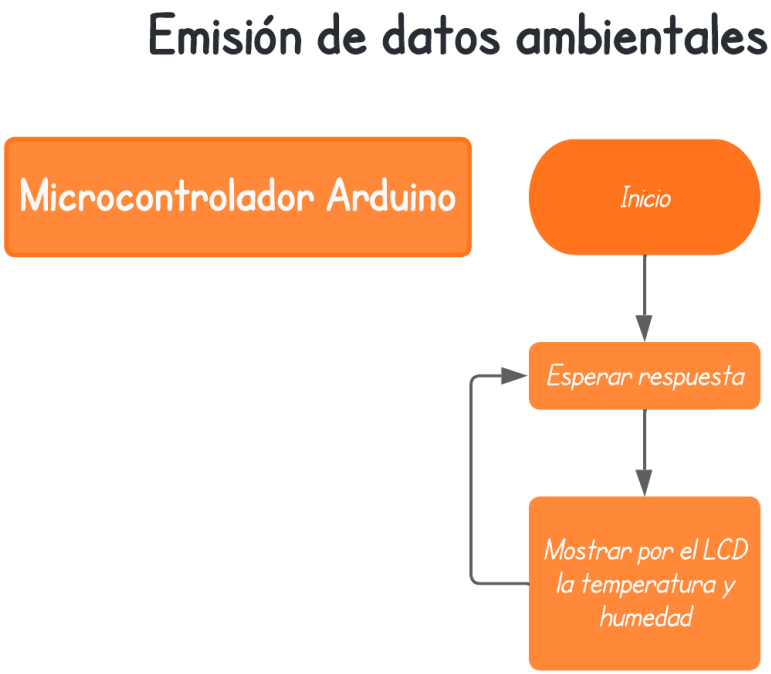


Figura 8.7: Diagrama de flujo funcionalidad proximidad y choque: punto de vista microcontrolador

CAPÍTULO 9

Implementación de la solución IoT

En este apartado vamos a poner en práctica los conocimientos que hemos ido adquiriendo a lo largo del trabajo para implementar el prototipo del caso de uso considerado. Primeramente, realizaremos el desarrollo físico para poder llevar a cabo la solución, esto es, la creación del circuito, el alta de usuarios en la nube y el registro e instalación del dispositivo de borde en la nube. Posteriormente, llevaremos a cabo el desarrollo software, tanto asociado a la placa Arduino como a la Raspberry Pi.

9.1. Creación del circuito

En el apartado 7.1 vimos los principales componentes hardware que iban a ser necesarios para implementar la solución IoT. Además de estos, hemos necesitado de dos resistencias, una de 220 ohmios y otra de 2000 ohmios, y de un potenciómetro para regular el contraste de la pantalla LCD así como cables para interconectar todo.

Como vemos en la Figura 9.1, he colocado cada funcionalidad en un protoboard distinto. En concreto, en el recuadro rojo se encuentra la funcionalidad de comunicación ambiental y en el recuadro azul se sitúa toda la circuitería referida a la funcionalidad de detección de proximidad y choque. En el recuadro verde se encuentra la Raspberry Pi que actúa como servidor de borde (y tendrá instalado el software Greengrass Core) y que tiene acoplado el conjunto de sensores Sense HAT que conforman la contraparte de la solución ambiental.

La comunicación con el servicio en la nube se hará a través de MQTT. Para ello, el mensaje pasará por nuestro router inalámbrico para posteriormente viajar por el núcleo de la red hasta llegar al servicio en la nube.

Por último, cabe mencionar la funcionalidad del microcontrolador Arduino, que viene a emular el procesador del coche, y que, a través de sus pines, recopila, procesa y envía la información que proviene del servidor de borde a través de la interfaz USB mostrada en la Figura 9.1. Los pines del microcontrolador Arduino son los encargados de recopilar o comunicar la información a los sensores. En concreto, para la funcionalidad de proximidad y choque:

- El pin número 7 recibe del sensor de proximidad el momento en el que se emite una onda de ultrasonidos.
- El pin número 6 recibe del sensor de proximidad el momento en el que esa onda previamente emitida ha regresado.

- El pin número 5 recibe del sensor de contacto una señal digital sobre si hay contacto o no.
- El pin número 4 envía información al buzzer sobre la frecuencia a la que emitir el sonido.
- El pin número 3 envía información al led para encenderse o apagarse.

Para la funcionalidad de comunicación ambiental, los pines que utilizamos son los numerados del 8 al 13 y todos ellos se encargan de codificar la información ambiental en la pantalla LCD, es decir, actúan como pines de salida. Además, utilizamos los pines de tierra y 5V de la placa para polarizar los sensores y la pantalla LCD.

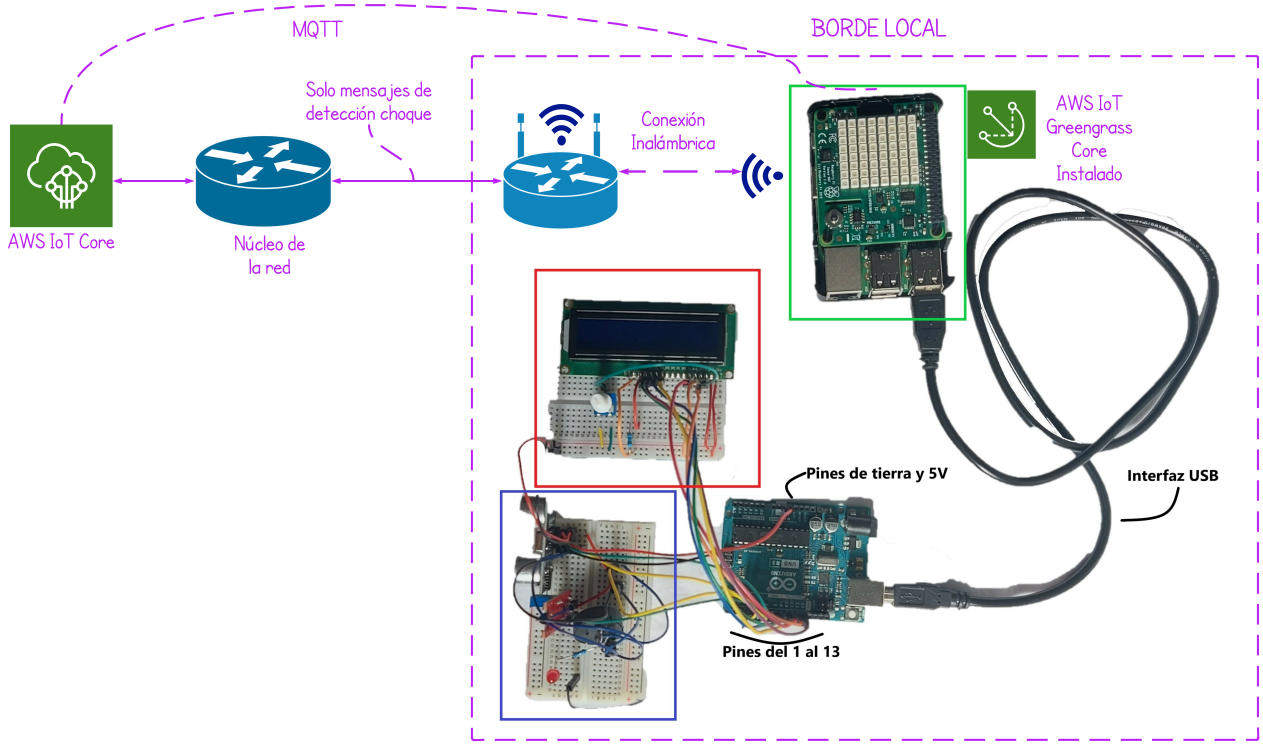


Figura 9.1: Montaje de la solución IoT

9.2. Configuración previa en la nube

En esta sección vamos a realizar las configuraciones necesarias para poder poner en funcionamiento en la nube el servicio AWS IoT Greengrass. Lo que haremos será dar de alta los usuarios necesarios para administrar el servicio AWS IoT Greengrass e instalar el software "Greengrass Core" en nuestro servidor de borde (Raspberry Pi).

9.2.1. Creación de usuarios

Lo primero que deberemos hacer es crear dos usuarios en nuestra cuenta de AWS [MB22]. Con nuestra cuenta accediendo al servicio IAM (*Identity and Access Management*) vamos a seguir los siguientes pasos:

1. Nos situamos en "Usuarios" y seleccionamos "Agregar usuarios"

2. Llamaremos Admin al primer usuario y habilitamos las dos opciones: "Clave de acceso: acceso mediante programación" y "Contraseña: acceso a la consola de administración de AWS". Esto hará que en la sección de confirmación para finalizar la creación del usuario se genere un fichero .csv con las credenciales AWS del usuario que utilizaremos para interactuar con la CLI de AWS.

3. Creamos un grupo al que he llamado "Administrators" y añado al usuario Admin a dicho grupo. Al grupo de administradores le vamos a asignar una política denominada AdministratorAccess para permitir a los usuarios de este grupo tengan permisos de nivel de administrador.

4. Instalamos la CLI de AWS en nuestro ordenador en la siguiente página: `https://aws.amazon.com/cli/`

5. Una vez instalada vamos a usar los credenciales que hemos descargado al crear el usuario Admin, esto es, el fichero .csv. Con el comando "aws configure" en la CLI de nuestro ordenador completamos los parámetros que se nos indican. Estos parámetros serán AWS Access Key ID y AWS Secret Key, que se hayan en el fichero .csv, la región donde realizaremos toda la implementación, en mi caso, us-west-2 y el formato de salida por defecto que en mi caso he seleccionado json. He seleccionado una región americana pero bien podría hacerse en una región europea salvo en eu-south-1 (Europa del sur, región que nos correspondería geográficamente y me habría decantado por ella), donde actualmente el servicio AWS IoT Greengrass no está disponible. Haber seleccionado esta región no va a suponer una limitación para nuestro caso de uso.

Una vez creado el usuario Admin, vamos a crear el otro usuario al que he llamado "idtgg" (abreviación de *identification Greengrass*). Los pasos que debemos seguir son similares:

1. Nos situamos en "Usuarios" y seleccionamos "Agregar usuarios"

2. Llamaremos "idtgg" al usuario y habilitamos "Clave de acceso: acceso mediante programación". Este usuario no necesitará de contraseña para administrar la consola de AWS. No nos olvidemos de descargar el fichero csv con sus credenciales al terminar de crearlo.

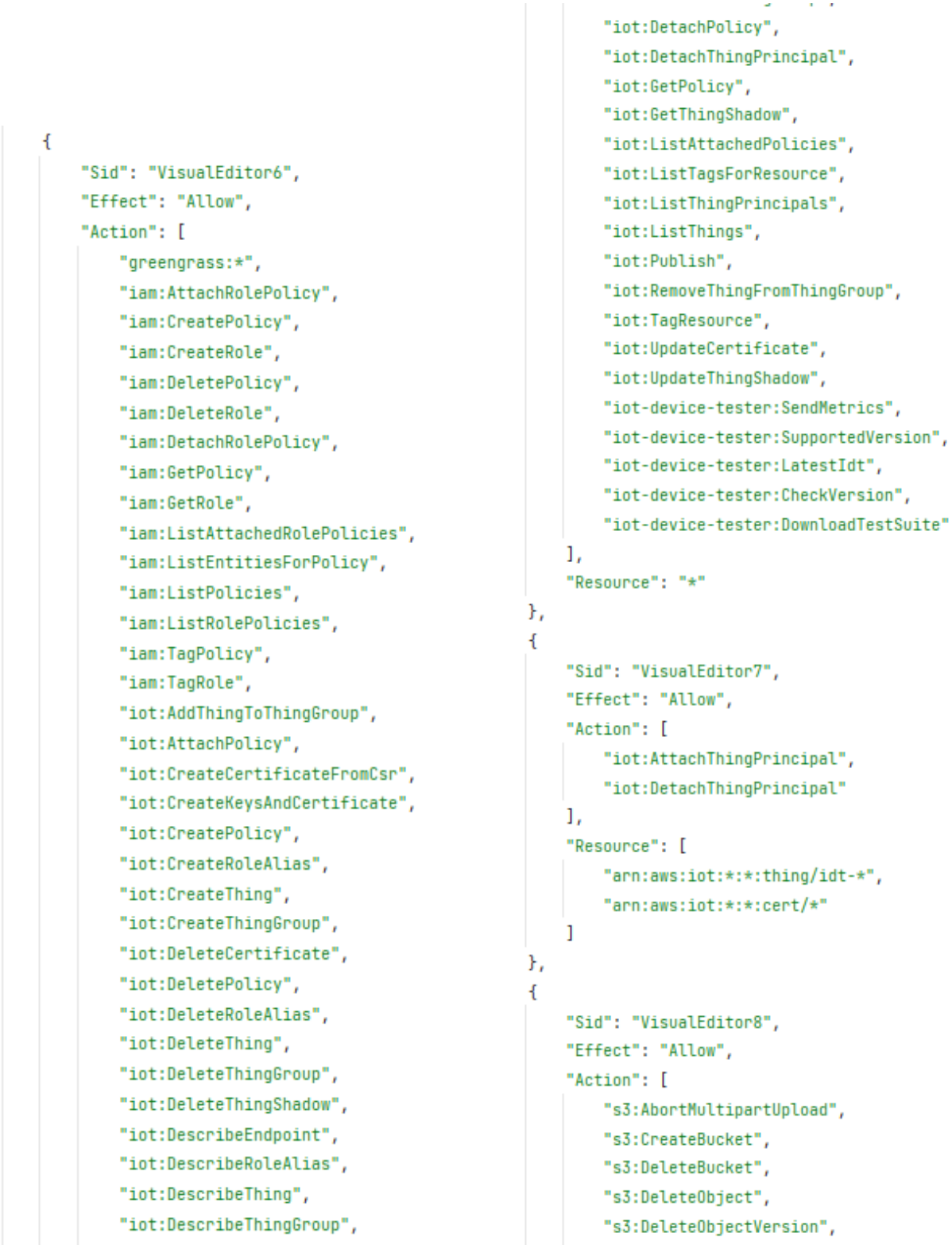
3. Vamos a crear una política de administración particular para este usuario. Para ello introducimos el formato de permisos que se muestra en las Figuras 9.2, 9.3 y 9.4 en la pestaña JSON. La política la he llamado "idt-gg-permisions".

4. Creamos un grupo al que he llamado "Provision-IDT-Greengrass" y añado al usuario idtgg a dicho grupo. A este grupo le vamos a asignar la política creada anteriormente.

5. Una vez instalada vamos a usar las credenciales que hemos descargado al crear el usuario idtgg, esto es, el fichero .csv y, de nuevo, con el comando "aws configure –profile idtgg" en la CLI de nuestro ordenador completamos los parámetros que se nos indican. Estos parámetros serán AWS Access Key ID y AWS Secret Key, que se hallan en el fichero .csv, la región donde realizaremos toda la implementación, en mi caso, us-west-2 y el formato de salida por defecto, que en mi caso he seleccionado json.

(a) Sección 1 (b) Sección 2

Figura 9.2: Política para el usuario idtgg (parte 1), tomado de *[MB22]*



(a) Sección 3 (b) Sección 4

Figura 9.3: Política para el usuario idtgg (parte 2), tomado de *[MB22]*

```
            "s3:GetObject",
            "s3:ListBucket",
            "s3:ListBucketVersions",
            "s3:ListMultipartUploadParts",
            "s3:PutBucketTagging",
            "s3:PutObject",
            "s3:PutObjectTagging"
        ],
        "Resource": "arn:aws:s3:::idt*"
    },
    {
        "Sid": "VisualEditor9",
        "Effect": "Allow",
        "Action": [
            "iot:AddThingToThingGroup",
            "iot:CreateThingGroup",
            "iot:DescribeThingGroup"
        ],
        "Resource": "arn:aws:iot:*:*:thinggroup/*"
    },
    {
        "Sid": "VisualEditor10",
        "Effect": "Allow",
        "Action": [
            "iot:AddThingToThingGroup"
        ],
        "Resource": "arn:aws:iot:*:*:thing/idt-*"
    },
    {
        "Sid": "VisualEditor11",
        "Effect": "Allow",
        "Action": [
            "iot:DescribeRoleAlias"
        ],
        "Resource": "arn:aws:iot:*:*:rolealias/*"
    }
]
}
```


Figura 9.4: Política para el usuario idtgg (parte 3), tomado de *[MB22]*

9.2.2. Instalación del software "Greengrass Core"

Una vez hemos creado los usuarios, vamos a instalar el software de IoT Greengrass en nuestro servidor de borde, es decir, vamos a instalar el software "Greengrass Core" en nuestra Raspberry Pi. En la consola de la Raspberry Pi hacemos lo siguiente:

1. Nos situamos en el directorio principal: cd ∼/
2. Descargamos el software: curl -s `https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-latest.zip` > greengrass-nucleus-latest.zip (comando que nos recomienda AWS en el apartado del servicio en la nube AWS IoT/Dispositivos de Greengrass/Dispositivos Principales/Configurar un dispositivo de núcleo)

3. Descomprimimos el fichero descargado: unzip greengrass-nucleus-latest.zip -d greengrass && rm greengrass-nucleus-latest.zip
4. Debemos crear dos variables de entorno con las credenciales del fichero .csv del usuario idtgg:
   - export AWS_ACCESS_KEY_ID=CredencialPublica_idtgg
   - export AWS_SECRET_ACCESSACCESS_KEY_ID=CredencialPrivada_idtgg
5. Por último, ejecutamos el siguiente comando que instalará el software:

```
sudo -E java -Droot="/greengrass/v2" -Dlog.store=FILE
-jar ./GreengrassCore/lib/Greengrass.jar
--aws-region us-west-2
--thing-name EdgeServer001
--thing-group-name AutoGroup
--component-default-user ggc_user:ggc_group
--provision true
--setup-system-service true
--deploy-dev-tools true
```

   Con este comando se instalará, junto con el software, la CLI de Greengrass que nos permitirá interactuar con el núcleo de Greengrass. También hemos llamado a nuestro núcleo como "EdgeServer001" que se encuentra en el grupo de dispositivos de núcleo "AutoGroup".

9.3. Software instalado en el microcontrolador Arduino

La instalación del programa en la placa del microcontrolador se hace a través de un entorno de programación (IDE) particular para este dispositivo llamado Arduino IDE. Este entorno de desarrollo consiste en un editor de código, un compilador, un depurador y una interfaz gráfica (GUI). El programa que diseñamos, una vez compilado, se instalará en la memoria flash de Arduino a través del puerto serie. El lenguaje de programación también es particular pero está basado en el lenguaje C. Los programas que creamos a través del editor, denominados sketchs, se almacenan en un fichero con extensión .ino.

Con este entorno de desarrollo, he generado un fichero llamado "controladorVehiculo.ino" cuyo código se va a poner en funcionamiento en el microcontrolador para poder comunicarse con los sensores y el servidor para así llevar a cabo correctamente las dos funcionalidades propuestas.

Un sketch de Arduino esta formado por dos funciones principales:

- *void setup()*. Las instrucciones que incluyamos en esta función se ejecutarán una vez en el microcontrolador. Por tanto, esta función sirve para inicializar el sistema.
- *void loop()*. Es una función que se ejecutará indefinidamente mientras la placa esté habilitada. Introduciremos aquí los comandos que se ejecutarán en la placa Arduino.

9.3.1. Definición de variables

Antes de comentar el contenido de las funciones setup() y loop(), conviene mencionar que hemos hecho una serie de definiciones de ciertas variables como se puede ver en la Figura 9.5, que, al estar fuera de ambas funciones se comportan como variables globales. Serán útiles para el desarrollo posterior (contenido de loop() y setup()).

Lo que hacemos es incluir el fichero de cabecera "LiquidCrystal.h", que nos permite usar las funciones necesarias para trabajar con la pantalla LCD. Los pines en vez de nombrarles por su número correspondiente, los damos un nombre para que sea más comprensible el código y no lleve a dudas durante su programación. Posteriormente, se definen los pines que se utilizan en el LCD, que, como vimos, eran aquellos que iban del 8 al 13.

Definimos dos variables tipo *long*, `duration` y `distance`, que servirán para calcular la distancia a la que se encuentra el objeto del sensor de proximidad.

```
#include <LiquidCrystal.h>

#define trigPin 7
#define echoPin 6
#define isAcrash 5
#define buzzer 4
#define redAlarm 3

LiquidCrystal lcd(12, 11, 10, 8, 13, 9);
long duration, distance;
```

Figura 9.5: Definición de variables globales

9.3.2. void setup()

En la Figura 9.6 se muestran los comandos necesarios para inicializar correctamente nuestro programa. Lo primero que hacemos es establecer, en bits por segundo, la tasa de transmisión en el puerto serie de Arduino, que es por donde enviaremos y recibiremos los datos. Posteriormente, inicializamos la interfaz de la pantalla LCD proporcionando sus dimensiones (16X2). Finalmente, los pines que utilizamos para conectarlos a los sensores debemos definirlos como pines de salida o de entrada de datos.

```
void setup()
{
  // Establecemos la tasa de transmision de datos en serie
  Serial.begin(115200);
  // Se inicializa la interfaz de la pantalla LCD
  lcd.begin(16, 2);
  // Configuracion del modo de los pines que utilizamos
  pinMode(trigPin, OUTPUT);
  pinMode(buzzer, OUTPUT);
  pinMode(redAlarm, OUTPUT);
  pinMode(echoPin, INPUT);
  pinMode(isAcrash, INPUT);
}
```

Figura 9.6: Función setup()

9.3.3. void loop()

La función loop(), que se puede ver en la Figura 9.7, contiene los comandos que se ejecutarán periódicamente en el microcontrolador. Los primeros cinco comandos sirven para enviar un pulso de 10 microsegundos a través del sensor de proximidad y a partir de dicho momento llamamos a la función pulseIn() que calculará el tiempo que tarda en regresar dicho pulso. Como tenemos la duración y velocidad a la que se propaga el pulso, podemos calcular la distancia a la que se encuentra el objeto, variable que hemos llamado distance.

A través de la función sendData() enviamos la información sobre la distancia del objeto, junto con la existencia o no de contacto en el sensor de choque, al servidor de borde. Utilizamos la función readAndProcessData() para esperar a la llegada de la respuesta y actuar en los sensores en función de dicha respuesta.

```
void loop()
{
  // Enviamos un pulso
  digitalWrite(trigPin, LOW);
  delayMicroseconds(4);
  digitalWrite(trigPin, HIGH);
  delayMicroseconds(10);
  digitalWrite(trigPin, LOW);
  // Recibimos el ultrasonido de vuelta
  duration = pulseIn(echoPin, HIGH);
  // Calculamos la distancia en cm
  distance = (duration/2) / 29.1;
  // Los datos recibidos los enviamos al
  // servidor de Borde mediante una comunicación en serie
  sendData(distance, digitalRead(isAcrash));
  // Esperamos a recibir la respuesta
  readAndProcessData();
}
```

Figura 9.7: Función loop()

Nos falta ver cómo se han programado las dos funciones, sendData() y readAndProcessData(), que hemos creado para llevar a cabo las funcionalidades.

Función sendData()

Como podemos ver en la Figura 9.8, los parámetros se envían al servidor de borde mediante la función Serial.print(). Los dos parámetros, la distancia y la existencia o no de choque, se envían separados por una coma.

```
// Funcion que envia los datos recopilados por los sensores al servidor de borde
void sendData(long distance, bool crash)
{
    // Escribimos la distancia y si hay choque en el puerto serie
    Serial.print((String) distance);
    Serial.print(",");
    Serial.print((bool) crash);
}
```

Figura 9.8: Función sendData()

Función readAndProcessData()

La función, que puede observarse en la Figura 9.9, inicialmente espera a que llegue la respuesta a través de la condición Serial.available() == 0 y establecemos también un tiempo máximo para esperar a la respuesta de diez segundos. El mensaje que recibimos posee cinco parámetros, que, al separarse, ya que vienen delimitados por comas, se almacena en un vector de Strings llamado message y contendrá:

- message[0]. La frecuencia a la que debe emitir el buzzer.
- message[1]. La cadencia a la que tiene que emitir el tono el buzzer.
- message[2]. La cadencia a la que debe encenderse y apagarse el LED.
- message[3]. La temperatura que nos comunica el servidor de borde.
- message[4]. La humedad que nos comunica el servidor de borde.

Los dos últimos parámetros son enviados al LCD para mostrarse a través de la función showTemperatureAndHumidity(). Los tres primeros parámetros son evaluados de la siguiente manera:

- Si la cadencia a la que debe encenderse y apagarse el LED no es nula, quiere decir que sí o sí estamos en un caso de choque, ya que el LED solo debe encenderse en caso de choque. Por ello, se llama a la función thereIsACrash() que realiza lo pertinente en los sensores.
- Si la cadencia del LED es nula y además la frecuencia en la que debe emitir el buzzer es nula, estamos ante un caso de no existencia de objeto en las cercanías del sensor y, por tanto, no debe hacerse nada.
- En cualquier otro caso estamos ante un caso de cercanía de un objeto sin llegar al choque. Para ello, se invoca a la función thereIsNoCrash() para que haga lo que corresponda en los sensores.

```
// Funcion que recibe los datos provenientes del servidor de borde
void readAndProcessData()
{
    String message[5] = {"000", "000", "000", "00.0", "00.0"};
    String msg = "";
    int j = 0;
    int k = 0;
    // En caso de haber datos disponibles y los procesamos, enviamos true
    while(Serial.available() == 0){}
    // Tiempo máximo de espera para leer del puerto seria
    Serial.setTimeout(10);
    // Leemos del puerto serie
    msg = Serial.readStringUntil('\n');
    // Parseamos el mensaje ya que los parametros vienen separados
    // por comas en un mismo mensaje
    for(int i = 0; i < msg.length(); i++)
    {
       if(msg[i] == ',')
       {
          j++;
          k = 0;
       }
       else
       {
          message[j][k] = msg[i];
          k++;
       }
     }
     // Mostramos la temperatura y humedad recibida por LCD
     showTemperatureAndHumidity(message[3], message[4]);
     // Según el valor de la respuesta se da una situación de choque o distinta
     if (message[2].toInt() != 0)
     {
        thereIsACrash(message[0].toInt(), message[2].toInt());
     }
     else if (message[0].toInt() == 0)
     {
        noTone(buzzer);
     }
     else
     {
        thereIsNoCrash(message[0].toInt(), message[1].toInt());
     }
}
```

Figura 9.9: Función readAndProcessData()

De esta manera, podemos llevar a cabo el procesamiento en Arduino necesario para recibir los datos del servidor de borde y procesarlos adecuadamente. Sin embargo, debemos todavía concretar más ciertas funciones que tienen un menor nivel de abstracción y que explican aún más todo este proceso. Estas funciones son showTemperatureAndHumidity(),

showTemperatureAndHumidity(), thereIsACrash() y thereIsNoCrash().

Función showTemperatureAndHumidity()

Lo que hace esta función se muestra en la Figura 9.10. Primero se borra todo el texto previo que hubiese en el LCD para posteriormente escribir en la primera línea la nueva temperatura y en la segunda línea la nueva humedad.

```
// Muestra en el LCD la temperatura y la humedad recibida del servidor de borde
void showTemperatureAndHumidity(String temperature, String humidity)
{
   lcd.clear();
   lcd.setCursor(0, 0);
   lcd.print("Temperatura:" + temperature);
   lcd.setCursor(0, 1);
   lcd.print("Humedad:" + humidity);
}
```

Figura 9.10: Función showTemperatureAndHumidity()

Función thereIsACrash()

En esta función se aplican cambios en los sensores cuando hay choque. Tal y como muestra la Figura 9.11, lo que se hace es activar el buzzer a la frecuencia correspondiente y posteriormente, encender y apagar el LED rojo a una cadencia dada.

```
// Funcion que activa la alarma con una determinada frecuencia y cadencia dados
void thereIsACrash(int buzzerTone, int delayAlarmValue)
{
    // Tono del buzzer
    tone(buzzer, buzzerTone);
    // Activo la alarma
    digitalWrite(redAlarm, HIGH);
    // Cadencia de la alarma
    delay(delayAlarmValue);
    digitalWrite(redAlarm, LOW);
    delay(delayAlarmValue);
}
```

Figura 9.11: Función thereIsACrash()

Función thereIsNoCrash()

De forma similar a la función thereIsACrash(), esta función sería la que pone en funcionamiento los sensores cuando no existe choque. Como se muestra en la Figura 9.12, lo que se hace es encender el buzzer a la frecuencia dada y generaremos cadencia si estamos en los

casos en los que el objeto esta alejado y que, en consecuencia, recibiríamos un delayBuzzerValue diferente de cero.

```
// Funcion que activa el buzzer, con una determinada cadencia
// Situación de no choque
void thereIsNoCrash(int buzzerTone, int delayBuzzerValue)
{
    // No enciendo alarma
    digitalWrite(redAlarm, LOW);
    // Tono del buzzer
    tone(buzzer, buzzerTone);
    // Cadencia del buzzer
    delay(delayBuzzerValue);
    if(delayBuzzerValue != 0)
    {
      noTone(buzzer);
    }
}
```

Figura 9.12: Función thereIsNoCrash()

Con esto quedarían definido el software que instalaremos en el microcontrolador de Arduino para que pueda manejar las funcionalidades de proximidad y choque y de comunicación ambiental. Ahora veremos el software que instalaremos en el servidor de borde para manejar desde su parte dichas funcionalidades.

9.4. Creación del componente

Recordemos que, para incluir software en un dispositivo con el software Greengrass Core instalado, es necesario crear un componente y para ello es necesario un artefacto y una receta. De esta manera, el dispositivo Greengrass Core, al que hemos llamado EdgeServer001, puede ejecutar el bloque de código.

9.4.1. Creación del artefacto

El artefacto es el bloque de código que se va a estar ejecutando en el dispositivo de núcleo, es decir, en el servidor de borde, en nuestra Raspberry Pi. En IoT Greengrass podemos crear el artefacto a través de diversos lenguajes de programación pero hemos seleccionado Python por la versatilidad y legibilidad que otorga al código. A continuación, veremos el código que hemos creado para permitir una comunicación con Arduino y con el servicio en la AWS IoT Core a través de MQTT que permita llevar a cabo las funcionalidades previamente diseñadas.

Nuestra función main() se muestra en la siguiente Figura:

```
if __name__ == '__main__':
    comunicationArduinoAndCloud()
```

Figura 9.13: Función main()

Como podemos observar es una función muy simple que llama a comunicationArduinoAndCloud()

Función comunicationArduinoAndCloud()

La siguiente función, que puede observarse en la Figura 9.14, se encarga de realizar todas las funcionalidades que esperamos cumplir con este trabajo. Hace uso, a su vez, de otras funciones para lograr la totalidad de la funcionalidad.

Lo que se hace primeramente es conectarse al sensor SenseHat que mide la temperatura y humedad que vamos a recopilar. Posteriormente, abrimos un puerto serie que se corresponde con el que vamos a usar para intercambiar información con Arduino. Una vez hemos abierto el puerto realizamos repetidamente lo siguiente:

- Esperamos la llegada de los dos parámetros que nos envía Arduino, que son la distancia y la existencia o no de choque.
- Estos parámetros los dividimos ya que vienen separados por una coma y se obtiene, mediante la función processingEdgeServer(), el mensaje de cinco parámetros que debe enviarse a Arduino.
- Con la función write() enviamos estos cinco parámetros a Arduino.

```
def comunicationArduinoAndCloud():
    # Con la siguiente instrucccion nos conectamos al sensor SenseHat
    sense = SenseHat()
    # Abrimos el puerto serie
    with serial.Serial("/dev/ttyACM0", 115200, timeout = 0.01) as arduino:
        arduino.flush()
        response = "000,000,000"
        time.sleep(0.1) # Esperamos a abrir el puerto serie
        if arduino.isOpen():
            try:
                while True:
                    if arduino.inWaiting() > 0:
                        # Recopilamos temperatura y humedad
                        temperature = str(round(sense.get_temperature(), 1))
                        humidity = str(round(sense.get_humidity(), 1))
                        # Hay datos que leer
                        msg = arduino.readline().decode("utf-8")
                        # El mensaje viene en una string, hay que parsearlo
                        message = str(msg).split(',')
                        # Procesado del servidor de borde junto con la información ambiental
                        try:
                            response = processingEdgeServer(float(message[0]), int(message[1]), temperature, humidity)
                        except ValueError:
                            response = "000,000,000" + temperature + ',' + humidity
                        except IndexError:
                            response = "000,000,000" + temperature + ',' + humidity
                        # Despues de leer eliminamos los datos
                        arduino.flushInput()
                        # Enviamos el resultado del procesado
                        arduino.write(response.encode("utf-8"))
            except KeyboardInterrupt:
                print("KeyboardInterrupt has been caught.")
```

Figura 9.14: Función comunicationArduinoAndCloud()

Vamos a ver, de forma más precisa, de que manera el servidor de borde realiza el procesamiento para obtener los parámetros que luego envía a Arduino.

Función proccessingEdgeServer()

Esta función, dada en la Figura 9.15, se encarga de generar el mensaje que se debe enviar a Arduino en función de la distancia y la existencia o no de choque. Cuanto más corta sea la distancia, un tono más agudo enviaremos (primer parámetro), la cadencia del buzzer se verá también reducida (segundo parámetro) y si hay choque existirá cadencia en el LED (tercer parámetro). El parámetro cuarto y quinto es la temperatura y humedad respectivamente.

Además, si hay choque, ejecutamos otras dos funciones, buildmessage() y publishMessageToIoTCore() que se encargan de comunicar el choque en la nube. buildmessage() se encargará de crear el mensaje que debemos enviar y publishMessageToIoTCore() envía el mensaje a la nube a través de MQTT.

```
# Evalua la distancia y actua en consecuencia
def processingEdgeServer(distance, crash, t, h):
    if (distance <= 10 and crash == 1):
        # Como hay un choque, debemos comunicarlo y enviarlo al servicio IoTCore
        # Creo el mensaje
        message = buildMessage(t, h)
        # Publico el mensaje
        publishMessageToIoTCore(message)
        return "500,000,050," + t + ',' + h
    elif (distance <= 10): # Sonido continuo
        return "400,000,000," + t + ',' + h
    elif (distance < 30): # Sonido intermitente
        return "280,050,000," + t + ',' + h
    elif (distance < 50): # Sonido intermitente
        return "260,080,000," + t + ',' + h
    else:
        return "000,000,000," + t + ',' + h
```

Figura 9.15: Función proccessingEdgeServer()

Función buildMessage()

Esta función, que puede observarse en la Figura 9.16, tiene la responsabilidad de generar el mensaje que debemos enviar a la nube. Se construyen tres campos en el mensaje:

- *Timestamp*. Momento en el que se produce el choque.
- *devideId*. Id del dispositivo de borde
- *crashInfo*. Información sobre el choque.

```python
# Construye el mensaje que debemos enviar a AWS IoT Core
def buildMessage(temp, hum):
    message = {}
    # Momento del choque
    now = datetime.now()
    dateTime = now.strftime("%d/%m/%Y, %H:%M:%S")
    message['timestamp'] = 'Current date/time: ' + dateTime
    # id del dispositivo de borde
    message['deviceId'] = 'EdgeServer001'
    # Informacion que debemos comunicar
    message['crashInfo'] = 'Ha detectado un golpe. La temperatura es de ' + temp + ' grados y la humedad de ' + hum

    return message
```

Figura 9.16: Función buildMessage()

Función publishMessageToIoTCore()

En esta función, además de registrar información útil en un log que he creado (Figura 9.18), establezco el tema por el que se envían los mensajes de detección de choque "dt/local/crash". Posteriormente, publico el mensaje en la nube a través de la función publishMessage(), que hace uso de la conexión que establecemos en la Figura 9.18 y que a continuación analizaremos. Lo que se ha hecho es establecer una conexión con el servicio GreengrassCoreIPC y crea un cliente para poder comunicarnos con este.

```python
# Se encarga de preparar el mensaje a través de mqtt y posteriormente envia el mensaje
def publishMessageToIoTCore(message):
    # Topico del mensaje
    topic = 'dt/local/crash' if len(sys.argv) < 2 else sys.argv[1]
    testMode = False if len(sys.argv) < 3 else sys.argv[2]
    if testMode:
        logger.info('[testMode] truthy argument passed in sys.argv[2], testMode enabled')
        logger.info('[testMode] topic is: %s', topic)
        logger.info('[testMode] payload is: %s', message)
    else:
        publishMessage(ipcClient, topic, message)
```

Figura 9.17: Función publishMessageToIoTCore()

```python
# Creo un objeto logger para monitorizar los diferentes eventos
logger = logging.getLogger()
handler = logging.StreamHandler(sys.stdout)
logger.setLevel(logging.INFO)
logger.addHandler(handler)

ipcClient = awsiot.greengrasscoreipc.connect()

TIMEOUT = 10
DELAY = 10
qos = QOS.AT_LEAST_ONCE
```

Figura 9.18: Creación de un log

Función publishMessage()

La Figura 9.19 muestra cómo a partir del tema y el cliente, que se obtiene a través de la conexión establecida con el servicio en la nube AWS IoT Core, envía el mensaje a este servicio en formato json. Primero creamos una solicitud a la que necesitamos agregar el tema (topic), el contenido del mensaje (payload) y la calidad del servicio que debe proporcionarse que, para nuestro caso, creemos que lo más conveniente es que el mensaje llegue, al menos, una vez al receptor ($qos = QOS.AT\_LEAST\_ONCE$). Posteriormente, creamos una operación de tipo publicar en el servicio IoT Core e iniciamos la operación, a través de la función activate(), con la solicitud (`request`) creada. Creamos una variable llamada `future` que, si no recibe respuesta a los diez segundos, cancelará la operación.

```
# Función que envia el mensaje
def publishMessage(ipcClient, topic, message):
    messageString = json.dumps(message)
    request = PublishToIoTCoreRequest()
    request.topic_name = topic
    request.payload = bytes(messageString, "utf-8")
    request.qos = qos
    operation = ipcClient.new_publish_to_iot_core()
    operation.activate(request)
    future = operation.get_response()
    future.result(TIMEOUT)
```

Figura 9.19: Función publishMessage()

9.4.2. Creación de la receta

La receta de un componente es un fichero que define los detalles, dependencias, artefactos y ciclos de vida de un componente. La receta que hemos creado se puede ver en la Figura 9.20. Los ciclos de vida que hemos definido han sido el de instalación "install", que descarga la librería awsiotsdk, que es indispensable para descargar o establecer aquellos recursos que son necesarios para el funcionamiento del artefacto. El ciclo de vida "Run" ejecutará el artefacto que hemos creado. No existe, en este caso, ninguna dependencia con ningún componente. El apartado "Manifests" especifica los requisitos de plataforma que, para nuestro caso, es únicamente que el dispositivo donde se ejecuta el componente posea un sistema operativo Linux.

Comentar que en "AccessControl" habilitamos a nuestro componente a poder llevar a cabo la operación de publicación tan solo con el tema dt/local/crash hacia el servicio en la nube AWS IoT Core. Esta operación se denomina "aws.greengrass#PublishToIoTCore". El resto de parámetros ofrecen detalles sobre el componente pero que no afecta a la ejecución de este.

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.publisher",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "AWS IoT Greengrass component for Edge Computing",
  "ComponentPublisher": "Me",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "Message": "Hello",
      "accessControl": {
        "aws.greengrass.ipc.mqttproxy": {
          "com.publisher:mqttproxy:1": {
            "policyDescription": "Allows Access to publish to dt/local/crash",
            "operations": [
              "aws.greengrass#PublishToIoTCore"
            ],
            "resources": [
              "dt/local/crash"
            ]
          }
        }
      }
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux"
      },
      "Lifecycle": {
        "Install": {
    "RequiresPrivilege": true,
          "script": "python3 -m pip install --user awsiotsdk"
        },
        "Run": {
          "RequiresPrivilege": true,
          "script": "python3 {artifacts:path}/hbs_publisher.py"
        }
      }
    }
  ]
}
```

Figura 9.20: Receta para nuestro componente

9.4.3. Incorporar el componente al dispositivo de borde

Poseemos ya todo el software necesario para crear un componente e implementarlo. Sabiendo que las rutas donde se encuentran nuestra receta y artefacto son:

- *Receta* en ∼/Desktop/Components/recipes/hbs.publisher-1.0.0.json
- *Artefacto* en ∼/Desktop/Components/artifacts/hbs.publisher/1.0.0/hbs_publisher.py

Utilizamos el siguiente comando situándonos en el directorio ∼/:

sudo /greengrass/v2/bin/greengrass-cli deployment create –recipeDir Desktop/Components/recipes/ –artifactDir Desktop/Components/artifacts/ –merge "com.publisher=1.0.0"

Como podemos ver, hacemos uso de /greengrass/v2/bin/greengrass-cli que es la CLI utilizada para interactuar con el software de Greengrass y que instalamos junto con el software Greengrass Core en la Raspberry Pi.

Instalado el componente en el software de Greengrass y el sketch en el microcontrolador, podemos observar que ambas funcionalidades han sido implementadas con éxito y el prototipo funciona según lo esperado.

CAPÍTULO 10

Análisis, limitaciones y conclusiones del trabajo

En este capítulo, realizaremos un análisis sobre el funcionamiento del prototipo que hemos diseñado e implementado. También, expondremos las principales limitaciones a las que se ha expuesto este trabajo y que han causado un rendimiento menor al ideal. Por último, comentaremos una serie de conclusiones que pueden extraerse de este trabajo.

10.1. Análisis

Aspectos técnicos

Las funcionalidades que hemos implementado no requieren de recursos desmedidos, es más, podemos observar que con la Raspberry Pi es suficiente para poder llevar a cabo estas soluciones. Por una parte, el principal almacenamiento para poder llevar a cabo esta solución ha sido de:

- 101,7 MB para instalar el software Greengrass Core.
- 139,4 MB para instalar Python3 junto con las librerías necesarias.
- 4 KB para la receta del componente que hemos desarrollado.
- 8 KB para el artefacto del componente desarrollado.

Al disponer de 32 GB de almacenamiento, podemos ver que no hay ningún problema para mantener instalada la solución IoT. Por otra parte, la ejecución del componente en términos de recursos utilizados en la memoria RAM puede verse en la Figura 10.1. A través del comando:

sudo systemctl status greengrass.service

Podemos ver los procesos que tiene en funcionamiento y que, por tanto, son los que consumen recursos de RAM y procesamiento. Estos son:

- Dos procesos sh para ejecutar el software de Greengrass y el artefacto del componente creado (PID 2301 y 2408).
- El software de Greengrass se ejecuta a través de la Máquina Virtual de Java ya que este software está escrito en Java (PID 2305)
- El artefacto del componente que hemos creado, hbs.publisher, también debe ejecutarse.

A diferencia del software de Greengrass, este se ejecuta a través de Python. (PID 2411)

En consecuencia, el proceso que ejecuta el componente ocupa de manera ininterrumpida entre un 18 y un 20 por ciento de la CPU. Los cuatro procesos ocupan también 268,4 MB de la memoria RAM, lo que supone un 29 % de esta. Por tanto, nuestro servidor de borde es capaz de mantener, con cierta holgura, el servicio que pretendemos ofrecer.

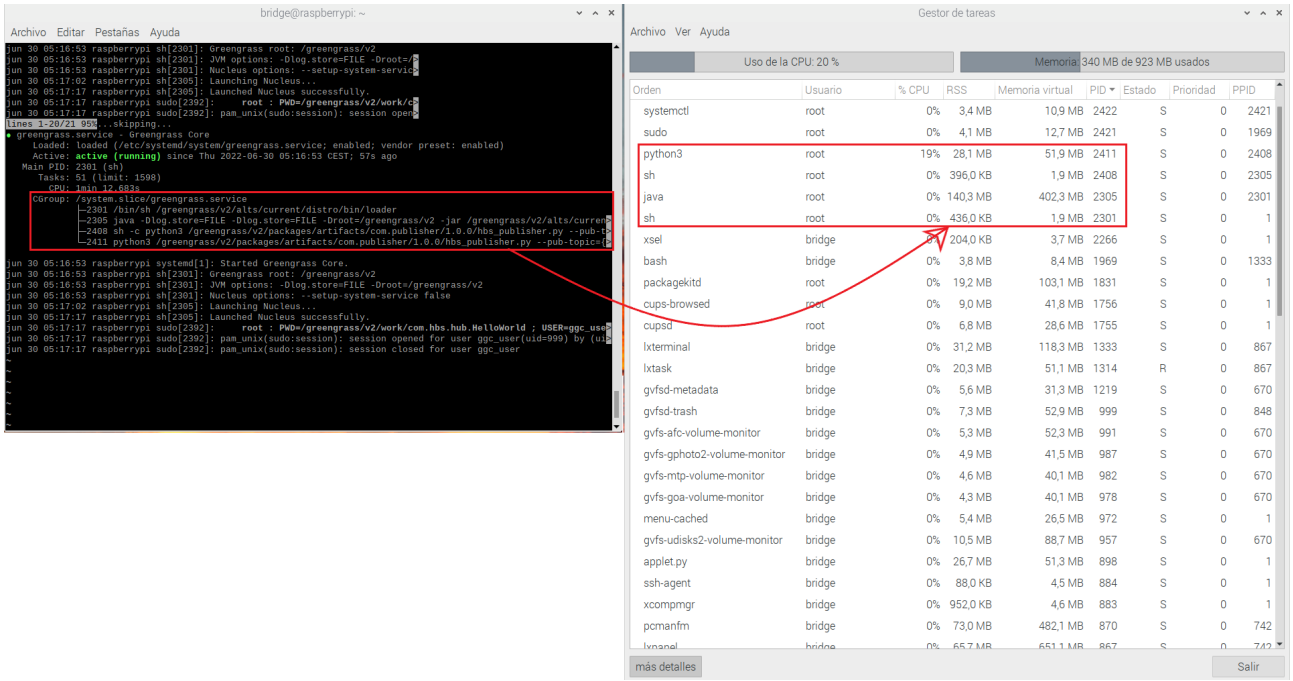


Figura 10.1: Uso de memoria RAM por parte de Greengrass

Podemos concluir que, la solución que hemos desarrollado se adecua a los recursos que tenemos. Sin embargo, una solución realista requeriría de servidores dotados con un mayor procesamiento computacional y almacenamiento para poder llevar a cabo las operaciones complejas y numerosas que componen una solución IoT para la conducción autónoma.

Irrelevancia de una conexión a Internet

Como vimos al principio de este trabajo, una solución *Edge Computing* se caracteriza, entre otras cosas, en realizar la mayor parte de su procesamiento en el servidor de borde y solo transmitir a la nube aquellas operaciones que requiriesen de un tratamiento computacional más complejo sin requisitos de retardo. Por lo tanto, tendría sentido que nuestra solución, a pesar de no tener conexión a Internet, pudiese seguir funcionando con normalidad, esto es, ofreciendo los resultados que se obtienen a través del procesamiento en el borde.

En nuestro caso, si desconectamos el servidor de borde de Internet a través del comando:

sudo iwconfig wlan0 txpower off

Podemos observar que, aun no habiendo conexión, las funcionalidades siguen su curso habitual salvo para la notificación en la nube del choque. Greengrass ejecuta los componentes de forma local lo que evita que el procesamiento se interrumpa por un fallo en la conexión a Internet. Los cambios que apliquemos en nuestra solución desde la nube, se harán efectivos cuando vuelva la conexión.

¿Y qué podemos decir de las notificaciones de choque con una interrupción en la conexión a Internet?

Al estar usando MQTT para enviar los mensajes a la nube, mientras se mantenga la sesión establecida, los mensajes se almacenarán y se reenviarán cuando se restablezca la sesión. Sin embargo, si la sesión expira, esto curre cuando pasan unos 30 minutos [Ama], los mensajes serán descartados y se establecerá una nueva sesión.

Por lo tanto, lograr una reconexión antes de que pasen los 30 minutos, algo que suele ser habitual, mantendrá los mensajes de notificación de choque para ser enviados lo antes posible a la nube. Si deseásemos que los mensajes no enviados se mantengan durante más tiempo, se tendría que hacer uso de otro tipo de protocolos de comunicación con la nube. En la Figura 10.2 puede observarse que ha llegado una notificación de choque producido a las 00:03:52 siendo, cuando se ha recibido en la nube, las 00:04:39. Lo que ha ocurrido es que temporalmente hemos apagado la conexión a Internet y en ese tiempo ha habido un choque cuya notificación no ha podido llegar a la nube hasta que fue restablecida la conexión.

Esta característica resulta muy útil ya que evitamos que la funcionalidad principal deje de funcionar ante cortes de conexión a Internet y además, notificaciones tan importantes, como las de avisar que ha habido un choque, se enviarán lo antes posible a la nube, es decir, se envían a la nube cuando hay conexión y cuando no la hay, no se elimina la notificación sino que se almacena y se envía en cuanto se obtenga de nuevo acceso a Internet.

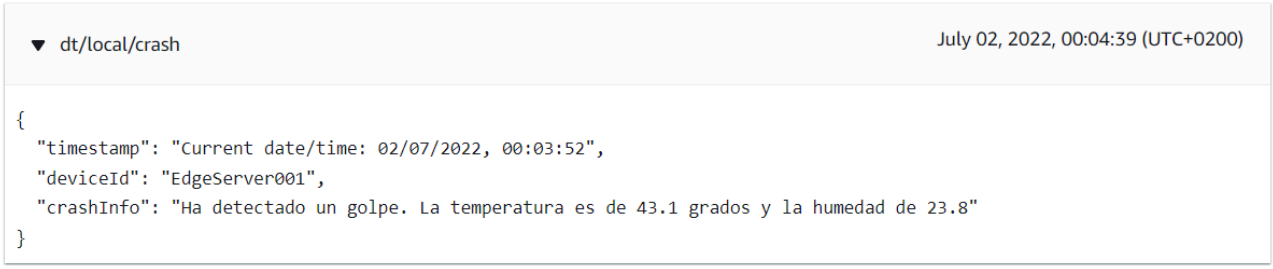


Figura 10.2: Ejemplo notificación de choque tras reconexión

En conclusión, la solución que hemos implementado también cumple con una de las características más básicas de *Edge Computing*, el mantenimiento de la funcionalidad aún no teniendo conexión. Y para interrupciones en la conexión puntuales con una duración menor a 30 minutos, podremos comunicar a la nube las ocurrencias de los choques.

El retardo

Si analizamos el retardo total de los mensajes, intercambiados entre el prototipo y el servidor de borde, como suma de 4 retardos diferentes [KR17]:

- Retardo de procesamiento
- Retardo de cola
- Retardo de transmisión
- Retardo de propagación

Al ejecutarse el componente en el servidor de borde, el retardo de propagación que tenemos en nuestro prototipo es, prácticamente, nulo. Tampoco habría retardo de cola en el servidor

de borde ya que solo llegan mensajes desde un solo dispositivo y además, la comunicación es síncrona lo que significa que no se enviará un nuevo mensaje hasta que no se reciba la respuesta al mensaje previo. El retardo de transmisión también se puede considerar prácticamente nulo ya que los mensajes que se envían no sobrepasan los 19 caracteres (19 bytes) y la capacidad del enlace, únicamente utilizada por el prototipo, es de 115200 bits por segundo por lo que este tipo de retardo también se puede considerar casi nulo.

El retardo de procesamiento sea, tal vez, el que más influencia tiene, y dependerá de la rapidez del servidor de borde de evaluar la solicitud del prototipo (microcontrolador) como la rapidez de este último para aplicar la respuesta en sus sensores y enviar otra solicitud. A pesar de que este tipo de retardo sea el dominante, no alcanza valores mayores los 15ms. Si añadimos el código marcado en rojo de la Figura 10.3 en el prototipo, podremos medir el retado total, es decir, el tiempo que hay entre el envío de una respuesta y la siguiente.

```
time.sleep(0.1) # Esperamos a abrir el puerto serie
if arduino.isOpen():
    try:
        while True:
            inicio = time.time()
            if arduino.inWaiting() > 0:
                # Recopilamos temperatura y humedad
                temperature = str(round(sense.get_temperature(), 1))
                humidity = str(round(sense.get_humidity(), 1))
                # Hay datos que leer
                msg = arduino.readline().decode("utf-8")
                # El mensaje viene en una string, hay que parsearlo
                message = str(msg).split(',')
                # Procesado del servidor de borde junto con la información ambiental
                try:
                    response = processingEdgeServer(float(message[0]), int(message[1]), temperature, humidity)
                except ValueError:
                    response = "000,000,000" + temperature + ',' + humidity
                except IndexError:
                    response = "000,000,000" + temperature + ',' + humidity
                # Despues de leer eliminamos los datos
                arduino.flushInput()
                # Enviamos el resultado del procesado
                arduino.write(response.encode("utf-8"))
                fin = time.time()
                print(fin - inicio)
    except KeyboardInterrupt:
        print("KeyboardInterrupt has been caught.")
```

Figura 10.3: Código para medir el retardo de ida y vuelta

Como podemos ver en la Figura 10.4, los retardos no sobrepasan los 15 ms por lo que nuestra aplicación puede considerarse que actúa en tiempo real y es así como se percibe cuando hacemos uso del prototipo. Ahondando más, hemos logrado una aplicación en tiempo real a través de AWS IoT Greengrass con dos dispositivos que se alejan de ser los más rápidos en cuanto a procesamiento. Esto quiere decir que, para despliegues reales con servidores realmente potentes y con controladores en los coches de mayor poder computacional podríamos reducir aún más este retardo de procesamiento.

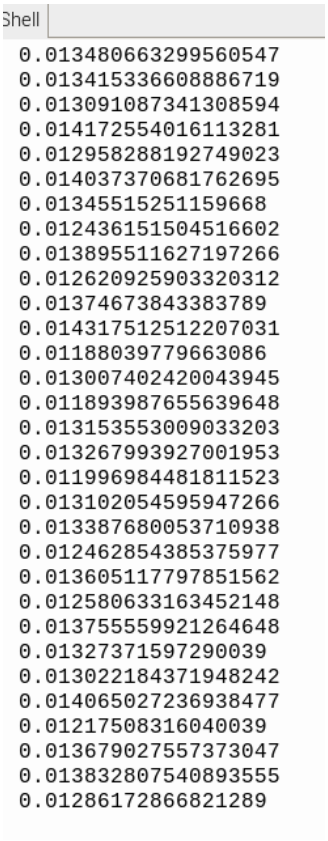


Figura 10.4: Retardos en la comunicación

Utilización del ancho de banda

Si bien se envían una multitud de mensajes al servidor de borde, no se envían hacia la nube sino que AWS IoT Greengrass nos dota de un software (Greengrass Core) que permite realizar procesados den el servidor de borde. Tan solo las notificaciones de choque se enviarían a la nube, que siendo realistas, esto no ocurriría tan frecuentemente. De esta manera, logramos que la sobrecarga de los dispositivos que procesan los mensajes que viajan a través de Internet, a lo que denominamos el núcleo de la red, se vea menos sobrecargado. Si esto lo viésemos desde una perspectiva más amplia, a través de un servicio aplicable a nivel mundial como es la conducción autónoma, donde los coche envían una ingente cantidad de datos a la red, con *Edge Computing* lograríamos reducir el ancho de banda que se utilizaría en el núcleo de la red para evitar congestiones de red que podrían tener consecuencias globales.

Colaboración mutua

Es importante destacar que, no solo los dispositivos que se conectan al servidor pueden generar datos, sino que también los propios servidores pueden generar información, útil y rápida, para los dispositivos conectados. En nuestro caso, hemos visto como la función de comunicación ambiental recoge datos ambientales en el servidor y los envía al microcontrolador. Esto, abre la posibilidad de establecer sensores en el servidor de borde, en lugar de situarlos en cada dispositivo, que son comunes a todos los dispositivos y que medirían la misma información, o muy similar, en la zona de cobertura del servidor, sin importar el dispositivo en el que estuviesen colocados reduciendo así el coste y complejidad de los dispositivos.

10.2. Limitaciones

Potencia computacional

Soluciones relativamente sencillas son capaces de llevarlas a cabo dispositivos como Arduino y Raspberry Pi tal y como hemos visto en el apartado anterior. Sin embargo, aplicaciones ambiciosas como un sistema completo de conducción autónoma sería impensable realizarlo a partir de estos dispositivos. Por tanto, la capacidad que nos proporcionan estos dispositivos nos ha limitado nuestras aspiraciones a crear sistemas más complejos.

Recursos

Hilando con la limitación anterior, la capacidad computacional no es muy elevada. Ahora bien, este trabajo no se centra en elaborar soluciones ambiciosas sino un primer acercamiento a esta tecnología para comprender sus principales características alejándonos de un prototipo realista. No olvidemos también que AWS IoT Greengrass nos cobraría por su servicio si este supera un determinado número de dispositivos de borde utilizados con su software lo que incrementaría el presupuesto.

Tiempo

El tiempo ha jugado también en nuestra contra, la no realización de proyectos más ambiciosos se debe en buena parte a esta limitación. Tal vez, si el tiempo de este proyecto fuese más amplio el proyecto habría alcanzado otra serie de objetivos.

Realismo

A fin y al cabo, solo hemos creado una arquitectura con un solo servidor de borde el cual solo debe atender las solicitudes de un solo dispositivo. De esta manera, nos resulta imposible analizar esta tecnología en un escenario donde un mayor de dispositivos interactúan con diversos servidores de borde en el contexto de una solución IoT para la conducción autónoma más ambiciosa. Por tanto, de este trabajo solo podemos extraer características básicas y no concretas sobre el funcionamiento en un contexto realista.

10.3. Conclusiones

De este trabajo podemos extraer las siguientes conclusiones sobre la tecnología *Edge Computing* y el prototipo desarrollado:

- El número de dispositivos conectados a Internet es cada vez mayor en cantidad, en diversidad de servicios y en generación de datos. En consecuencia, el ancho de banda utilizado en la red va a ser cada vez mayor. *Edge Computing* surge como una posible solución para reducir el ancho de banda que utilizan estos dispositivos en la red.
- *Edge Computing* es un paradigma computacional que aboga por un sistema descentralizado y jerarquizado. Es un sistema descentralizado porque delegamos el

procesamiento básico a servidores situados en el perímetro de la red mientras que la autoridad central, la nube, se encargaría de procesamientos más complejos. Podemos ver que también existe una jerarquía entre ambas partes, la nube, que no solo realiza procesamientos de mayor complejidad y delega los procesamientos más básicos al perímetro sino que controla el funcionamiento de esos servidores perimetrales.

- El borde o perímetro, zona que se encuentra cerca del origen y consumo de los datos, puede estar situado en el perímetro local, en el perímetro de la red de las telecomunicaciones o en el borde coubicado. La ubicación que se seleccione dependerá del servicio que se pretende dar.
- *Edge Computing*, además de reducir el ancho de banda en el núcleo de la red, permite reducir considerablemente el retardo de comunicación entre dispositivos al realizar el procesamiento en el perímetro. También permite ahorrar energía al no hacer uso de la infraestructura de la red ni de la nube. Tratar los datos de forma local evita problemas geográficos que puedan surgir.
- Si bien *Edge Computing* proporciona una serie de características novedosas, a cambio perdemos capacidad de procesamiento y almacenamiento. Si nos centramos únicamente en aspectos de procesamiento y almacenamiento, los servicios de computación en la nube son los únicos que llevarían mejor a cabo estas tareas. También es cierto que, *Edge Computing* es compatible con el uso de la nube para procesar y almacenar determinados datos que en el borde no se podrían llevar a cabo.
- Según que características que busquemos para nuestra solución IoT, *Edge Computing* podría ser una buena solución o no. Por ejemplo, servicios relacionados con ciudades inteligentes, conducción autónoma o automatización industrial se adecuarían muy bien con una tecnología como la de *Edge Computing*.
- Para demostrar los beneficios de esta tecnología, hemos optado por diseñar un prototipo que pretende dar cubrir funcionalidades, en su forma más básica, relacionadas con la conducción autónoma.
- Para poder desarrollar estas funcionalidades, es necesario de una serie de componentes hardware, como el servidor de borde, un microcontrolador Arduino y una serie de sensores, componentes software, en el que utilizaremos el servicio en la nube AWS IoT Greengrass el cual permite implementar una solución basada en *Edge Computing* y un servicio para diseñar, implementar y gestionar esta solución, llevada a cabo por nosotros mismos.
- AWS IoT Greengrass instalará un software denominado "Greengrass Core" en el servidor de borde y permitirá que a este se le agreguen módulos de software, denominados componentes y que proporcionan la lógica al servidor para actuar.
- Un diseño previo sobre la solución que se pretende dar, tanto a nivel de software como hardware, facilitará posteriormente nuestra implementación de la lógica en un componente y el programa para el microcontrolador así como facilitar la construcción del circuito.
- Una vez puesta en marcha la solución IoT, podemos observar como nos podemos comunicar con la nube, proporcionar un servicio en tiempo real y reducir el ancho de banda en el núcleo de la red sin necesidad de una conexión a Internet. Por tanto, AWS IoT Greengrass posibilita la creación de soluciones basadas en *Edge Computing*.
- Debido a limitaciones temporales, de capacidad computacional y presupuestarias, entre otras, nuestra solución IoT únicamente logra demostrar los aspectos más básicos de *Edge*

Computing.

10.4. Líneas futuras

Como hemos estado comentando, este trabajo ha consistido en entender y aplicar los conceptos básicos de *Edge Computing*. Por tanto, este documento puede servir como base para posteriores investigaciones. Un ejemplo de líneas que podrían tratarse son:

- Como AWS IoT Greengrass permite realizar soluciones IoT basadas en *Edge Computing* y sabemos cómo implementarlas, un camino que podría seguirse es la elaboración de soluciones más complejas, es decir, con un mayor número de componentes, dispositivos embarcados en el automóvil y dispositivos de núcleo. De esta manera, podemos analizar una comunicación más variada entre:
  - Dispositivos embarcados en el automóvil conectados tanto en el mismo dispositivo de núcleo como en distintos dispositivos de núcleo.
  - Dispositivos de núcleo.
  - De un dispositivo embarcado en el automóvil que cambia de dispositivo de núcleo que le atiende debido a que es móvil.
- Análisis comparativo, en términos prácticos, de otros servicios como Microsoft Azure IoT Edge y compararlos con AWS IoT Greengrass. Con ello, conseguiríamos estadísticas acerca del servicio que tiene un mejor rendimiento o por el contrario, dependerían dependería del caso de uso en cuestión.
- Explotación de los servicios compatibles con AWS Greengrass como los servicios de control como AWS IoT Core, IoT Device Defender o IoT RoboRunner o servicios de análisis como IoT Analitics, IoT Events o IoT SiteWise.
- AWS IoT Greengrass permite incorporar el aprendizaje automático a los dispositivos de borde. Se pueden utilizar modelos de aprendizaje automático creados, entrenados y optimizados en la nube para ejecutar inferencias localmente en los dispositivos. De esta manera, podría verse el rendimiento que tienen estos modelos a través de AWS IoT Greengrass.

Bibliografía

[aec18] Automotive Edge Computing Consortium, 2018. [Online] `https://aecc.org/`, accedido el 24 de Mayo de 2022.

[aia20] Cisco and/or its affiliates. Cisco annual internet report (2018–2023) white paper. Technical report, Cisco Systems, 2020.

[Ama] Amazon Web Services. *MQTT*, 1st edition.

[Ama20] Amazon Web Services, The address of the publisher. *AWS IoT Greengrass Developer Guide, Version 2*, 1st edition, Diciembre 2020.

[Ard22] Arduino Documentation. *Arduino UNO R3*, mayo 2022.

[Bar62] Paul Baran. *On Distributed Communications Networks*. Rand Corporation, California, 1st edition, 1962.

[CH20] V. K. CodyBumgardner and Caylin Hickey. *Making Sense of Edge Computing*. Hanning, Shelter Island, meap edition, 2020.

[Ele20] Elegoo Inc., Shenzhen, China. *37 Sensor Kit Tutorial: For Uno and Mega*, 2020.

[GLZ21] Jie Gao, Mushu Li, and Weihua Zhuang. *Connectivity and Edge Computing in IoT: Customized Designs and AI-based Solutions*. Springer, 1st edition, 2021.

[Goo] Google Cloud. *Documentación de Google Cloud IoT Core*. Disponible en `https://cloud.google.com/iot/docs`, accedido el 20 de Mayo de 2022.

[JCS18] QuanZhang Jie Chao and Weisong Shi. *Edge Computing: A Primer*. Springer, Detroit, 1st edition, 2018.

[JWJZ18] F. Jameel, S. Wyne, MA Javed, and S. Zeadally. Interference-aided vehicular networks: Future research opportunities and challenges. *IEEE Communications Magazine*, 56(10), October 2018.

[KR17] James F. Kurose and Keith W. Ross. *Computer Networking: A Top-down Approach*. Pearson, Shelter Island, 7th edition, 2017.

[KSDN22] K. Anitha Kumari, G. Sudha Sadasivam, D. Dharani, and M. Niranjanamurthy. *Edge Computing: Fundamentals, Advances and Applications*. CRC Press, 1st edition, 2022.

[KYT+20] Latif U. Khan, Ibrar Yaqoob, Nguyen H. Tran, S. M. Ahsan Kazmi, Tri Nguyen Dang, and Choong Seon Hong. Edge-computing-enabled smart cities: A

comprehensive survey. *IEEE Internet of Things Journal*, 7, April 2020.

[Lit17] T. Litman. Autonomous vehicle implementation predictions. *Victoria Transport Policy Institute*, October 2017.

[LLT+19] Shaoshan Liu, Liangkai Liu, Jie Tang, Bo Yu, Yifan Wang, and Weisong Shi. Edge computing for autonomous driving: Opportunities and challenges. *IEEE*, 107(8), August 2019.

[LNPW14] Q. C. Li, H. Niu, A. T. Papathanassiou, and G. Wu. 5g network capacity: Key elements and technologies. *IEEE Vehicular Technology Magazine*, 9(1), 2014.

[MB22] Indraneel Mitra and Ryan Burke. *Intelligent Workloads at the Edge*. Packt, 1st edition, 2022.

[McK22] Nick McKeown. Defining and leading the edge. *Intel newsroom*, February 2022.

[Mic] Microsoft Azure. *Azure IoT Edge*. Disponible en `https://docs.microsoft.com/es-es/azure/iot-edge/?view=iotedge-2020-11`, accedido el 20 de Mayo de 2022.

[mqt15] MQTT Essentials: The Ultimate Guide to MQTT for Beginners and Experts, 2015. [Online] `https://www.hivemq.com/mqtt-essentials/`, accedido el 26 de Mayo de 2022.

[Ras] Raspberry Pi. *Raspberry Pi 3 Model B+*.

[RGR17] David Reinsel, John Gantz, and John Rydning. Data age 2025: The evolution of data to life-critical, April 2017.

[SPX19] Weisong Shi, George Pallis, and Zhiwei Xu. Edge computing. *Proceedings of the IEEE*, 107(8), August 2019.

[TS08] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms*. Pearson, 2nd edition, 2008.

[Vá20] Leire Gutiérrez Vázquez. La industria del videojuego en España: Anuario 2020. Technical report, Asociación Española de Videojuegos, 2020.

[WABN20] Matthew Wopata, Sharmila Annaswamy, Fernando Brügge, and Kuan Ngiam. Industrial edge computing market report 2020-2025. Technical report, IoT Analytics, october 2020.

[YLH+18] Wei Yu, Fan Liang, Xiaofei He, William Grant Hatcher, Chao Lu, Jie Lin, and Xinyo Yang. A survey on the edge computing for the internet of things. *IEEE Access*, March 2018.