TRABAJO FIN DE MÁSTER

MÁSTER UNIVERSITARIO EN DESARROLLO DE SOFTWARE

Sistema IoT para el seguimiento médico y control de la salud de las personas adultas

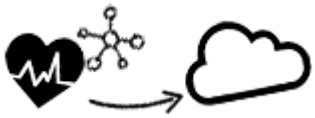
Autor
Jaime Frías Funes

Directores
Carlos Rodríguez Domínguez
Miguel J. Hornos Barranco

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Granada, septiembre de 2020



Sistema IoT para el seguimiento médico y control de la salud de las personas adultas

Autor
Jaime Frías Funes

Directores
Carlos Rodríguez Domínguez
Miguel J. Hornos Barranco

Sistema IoT para el seguimiento médico y control de la salud de las personas adultas

Jaime Frías Funes

Palabras clave: Internet de las Cosas, microservicios, composicion de microservicios, Kubernetes, salud

Resumen

El crecimiento inpredecible y exponencial del Internet de las Cosas ha hecho que esta tecnología se utilice en cada vez más aspectos de nuestra sociedad actual. Uno de los campos en los que se está aplicando es en el campo de la la salud, donde puede suponer una gran avance para complementar la labor de los sanitarios. Este trabajo propone un sistema dirigido a esta tecnología aprovechando las ventajas de los microservicios, con el objetivo de capturar los parámetros de un adulto y procesarlos para ser utilizados por sanitarios sin que se requiera la presencia física de ellos. Este sistema utilizará Kubernetes para la composición de un conjunto de microservicios que procesarán esta información y también un servicio web para visualizar correctamente estos datos en tiempo real.

IoT system for medical monitoring and health control of adults

Jaime Frías Funes

Keywords: Internet of Things, microservices, microservice composition, Kubernetes, healthcare

Abstract

The unpredictable and exponential growth of the Internet of Things has made this technology is used in more and more aspects of our society today. One of the fields in which it can apply is in the field of health, where it can be a great advance to complement the work of the doctors. This work proposes an IoT system taking the advantages of microservices, capturing the parameters of an adult and processing them to be used by doctors without requiring their presence. This system will use Kubernetes for the composition of a set of microservices that will process this data and also a web service to correctly visualize this information in real time.

Yo, **Jaime Frías Funes**, alumno del **Máster en Desarrollo de Software** de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI 50625384S, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Máster en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: Jaime Frías Funes

Granada a 18 de septiembre de 2020.

D. **Carlos Rodríguez Domínguez**, Profesor del Departamento de Lenguajes y Sistemas Informáticos de la Universidad de Granada.

D. **Miguel J. Hornos Barranco**, Profesor del Departamento de Lenguajes y Sistemas Informáticos de la Universidad de Granada.

Informan:

Que el presente trabajo, titulado ***Sistema IoT para el seguimiento médico y control de la salud de las personas adultas***, ha sido realizado bajo su supervisión por **Jaime Frías Funes**, y autorizamos la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a 18 de septiembre de 2020.

Los directores:

Carlos Rodríguez Domínguez **Miguel J. Hornos Barranco**

Yo, **Jaime Frías Funes**, alumno del **Máster en Desarrollo de Software** de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI 50625384S, declaro explícitamente que el trabajo presentado es original, entendido en el sentido que no he utilizado ninguna fuente sin citarla debidamente.

Fdo: Jaime Frías Funes

Granada a 18 de septiembre de 2020.

Agradecimientos

Agradezco en primer lugar a mis dos tutores por guiarme y tener la paciencia necesaria con en este proyecto en el que prácticamente me iniciaba con este tipo de tecnologías con las que no había trabajado anteriormente. Además, agradecer a mis compañeros de máster por ayudarme tanto cuando lo he necesitado en este trabajo como en el resto de asignaturas de este curso.

Índice general

Índice de figuras

Capítulo 1

Introducción

El crecimiento inpredecible y exponencial del Internet de las Cosas ha cambiado el mundo y también ha permitido el abaratamiento rápido en los costes de estos componentes, permitiendo su uso al publico general, lo que ha ocasionado que no dejen de existir nuevos avances e innovaciones en este campo, sobre todo en el área de los dispositivos ubicados en viviendas [22]. Esto también ha permitido diversos avances, y en este caso, el de la salud. Gracias a estas nuevas tecnologías, a través de determinados sensores conectados a una red IoT, es posible tratar estos datos sin necesitar, por ejemplo, un paramédico que extraiga esta información.

Este proyecto trata de obtener datos obtenidos de un paciente y de su entorno correspondiente a través de sensores y diversos servicios, procesarlos y finalmente hacerlos llegar a una plataforma objetivo, a través de diversos protocolos de comunicación. Para ello, es necesario valorar independientemente cada uno de los puntos por los que esta información será transladada, y cómo y qué tecnologías se utilizarán en cada punto de este proceso.

1.1. Motivación

La tecnología del Internet de las Cosas, al ser cada vez más popular y utilizada en cada vez más entornos, es una tecnología cada vez más barata. El campo de la salud es un campo cada vez más afectado por los recortes presupuestarios, siendo cada vez más difícil disponer de una atención continua por parte de los sanitarios. Tener un control de la salud de estas personas con algún tipo de riesgo es esencial, pero no siempre es posible disponer de una persona que esté presencialmente tomando estos datos. Por ello, propuestas de monitorización de pacientes con esta tecnología son muy necesarias para seguir avanzando en este campo.

1.2. Objetivos

El objetivo principal de este trabajo es el de desarrollar un sistema capaz de procesar de forma rápida y con pocos recursos de computación los parámetros sobre la salud de un adulto.

Como objetivos específicos de este proyecto, se podrían distinguir:

- Llevar a cabo una revisión bibliográfica sobre los conceptos clave sobre los que se trabajará en este trabajo: composición de servicios web, microservicios y Cloud Computing.
- Revisar el estado del arte sobre posibles metodologías a aplicar.
- Seleccionar una infraestructura hardware que cumpla adecuadamente las condiciones del entorno de este problema.
- Lograr componer microservicios que, aprovechando las ventajas que ofrece esta tecnología, consigan en conjunto un sistema escalable, fiable y flexible de monitoreo de salud.
- Utilizar un almacenamiento de datos que se adecúe a las necesidades de este proyecto.
- Procesar y almacenar la información recopilada adecuadamente para que esta pueda ser representada gráficamente y pueda determinarse cuando un parámetro es anormal y pueda suponer un riesgo en la salud del paciente.
- Desarrollar un servicio frontend en forma de aplicación web en el que visualizar adecuadamente toda esta información recopilada por la composición de microservicios.

1.3. Estructura del resto de la memoria

En este trabajo se describirá una propuesta de sistema creado a partir de un clúster con Kubernetes en una infraestructura creada con placas Raspberrys y una serie de sensores propuestos a continuación. Clúster en el que se compondrán microservicios que en conjunto, se ocuparán de trasladar la información de los sensores, procesarla y enviarla adecuadamente a una base de datos para así ser visualizada en una aplicación web.

1.3.1. Revisión bibliográfica

Antes de comenzar este trabajo, se ha realizado un análisis de los principales artículos que describen las tecnologías que se utilizarán, además de

comparar sus principales ventajas y características con el campo de la salud. En este análisis se tratará a las arquitecturas microservicio, al Cloud Computing, y por último la composición de servicios web.

1.3.2. Diseño

En este apartado se describirá todo el desarrollo de este trabajo, desde la infraestructura propuesta a utilizar, hasta los detalles sobre la composición de microservicios utilizada.

1.3.2.1. Infraestructura

Se requiere de una infraestructura hardware que se adapte al contexto del problema. Esta infraestructura contiene desde el conjunto de sensores que se utilizarán para conocer el estado en tiempo real del paciente (Temperatura del cuerpo, glucómetro, pulso cardíaco, respiraciones por minuto...), hasta la plataforma de gestión de contenedores que se utilizará para administrar los diferentes servicios y protocolos de comunicación que seguirán cada uno de los nodos encargados para cada labor sensorial. Respecto a estos sensores, una opción que se plantea es la plataforma e-Health, que permite extraer información vital de pacientes a través de sensores desarrollados exclusivamente para esta plataforma. Para utilizarla sólo es necesario conectarla a una placa Arduino o Raspberry Pi [23]. En este trabajo se presenta una serie de propuestas hardware de cómo implementar la solución planteada, pero se trabajará con datos que simulan el estado de un paciente real a través de servicios que generan aleatoriamente esta información, en vez de captarla a través de sensores.

1.3.2.2. Microservicios y estructura

Después de obtener esta información a partir de la infraestructura o siendo esta generada, es necesario procesarla adecuadamente y enviarla a una base datos donde esta pueda ser visualizada y utilizada para diversos propósitos. Esto se conseguirá gracias al uso de Kubernetes [24], que permite la composición de contenedores en clústeres, y a través de esta composición los datos serán enviados a un servicio de Redis [32] que actuará a modo de caché, para después ser leidos por un servicio almacenador que los organizará y enviará adecuadamente a la base de datos, utilizando para ello Cloud Firestore [37]. Se utilizará además Kubernetes [24] como gestor de contenedores y como tecnología de orquestación de servicios. Permitiendo conectar entre sí todos estos puntos de origen de datos, llevando toda la información a un único lugar donde ser procesada, y solucionando los problemas de comunicación y paso de mensajes que puedan ocurrir. Además de los sensores, se utilizarán una serie de APIs para obtener variables en el entorno de la vivienda del paciente: a través de la ubicación de esta, se llamará a estos

servicios y se podrán conocer variables como el índice de rayos ultravioleta o la contaminación del aire, para así notificar al paciente si corre un riesgo en el exterior en un momento determinado.

1.3.3. Servicio web

Con el fin de demostrar una utilización de estos datos obtenidos a través de la composición de microservicios descrita anteriormente, se ha desarrollado una sencilla aplicación web con TypeScript y ReactJS [46] que los obtendrá a partir de la base de datos y estos se representarán en gráficas mostrando el progreso de la salud de la persona y además estos son actualizados en tiempo real, para determinar en qué estado se encuentra el paciente en cada momento, además de mostrar en la interfaz cuando un parámetro está fuera de lo normal, lo que supondría un problema en la salud de este paciente.

1.3.4. Apéndices

Tras exponer en la memoria todo lo relacionado con el proceso llevado a cabo para desarrollar este trabajo, se describe en estos apéndices información adicional sobre los detalles de implementación que se han escogido como relevantes. Estos apéndices explican: el proceso seguido para crear un clúster de Kubernetes en la infraestructura propuesta, cómo crear un microservicio que simule la toma de datos de un paciente, y por último, cómo crear un componente en React que sea capaz de leer los datos recopilados y mostrarlos gráficamente en tiempo real.

Capítulo 2

Revisión biblográfica

2.1. Arquitecturas Microservicio

En esta sección se resumirá la definición de lo que es una arquitectura microservicio, la forma de escalado que tiene, y por último se enfocará este tipo de arquitecturas al Internet de las Cosas, descrubiendo sus características y diferencias en común.

2.1.1. Definición

Aunque hay muchas definiciones de microservicio, no existe ninguna definición exacta de este término, ya que hay una gran variedad de tipos de arquitecturas orientadas a servicios que podrían cumplir las condiciones necesarias para ser consideradas arquitecturas "microservicio". En general son servicios autónomos y pequeños que trabajan en conjunto para conseguir un objetivo común entre todos. Son servicios débilmente acoplados limitados a un contexto en particular. Se trata de una tecnología que está orientada al escalado de sistemas y a unos determinados objetivos y centrados en la reemplazabilidad. Se podrían destacar estas características fundamentales [1]:

1. Pequeño tamaño
2. Escalabilidad
3. Replicabilidad
4. Facilidad en el mantenimiento
5. Paso de mensajes y protocolos de comunicación asíncronos
6. Limitados a contextos
7. Desarrollados de forma autónoma

8. Desplegados independientemente

9. Descentralizados

Además, se destaca que el valor real de los microservicios se alcanza cuando nos centramos en dos aspectos clave, la velocidad y la seguridad, cada decisión que se tome en el desarrollo debe jugar un importante papel en esos dos aspectos.

La arquitectura microservicio se ha convertido en un enfoque para soportar modelos de empresa muy escalables y cambiantes, como Netflix o Amazon. Por lo general las principales razones por las que las empresas optan por esta arquitectura son la escalabilidad y el mantenimiento. Además, el tiempo de despliegue reducido y la ventaja de escoger mejores tecnologías para partes individuales de cada aplicación juegan un rol muy importante [2].

2.1.2. Escalabilidad

Es una característica clave para el paradigma de los microservicios, en función de la razón de por qué la escalabilidad es necesaria, existen diferentes enfoques [3].

2.1.2.1. Replicabilidad

Gracias al pequeño tamaño de los microservicios, pueden llevar esta característica al extremo: cada capacidad del negocio, incluyendo su funcionalidad y datos relacionados, se lleva a cabo en un servicio independiente, el cual puede ser desplegado por un anfitrión diferente ubicado en otro de los microservicios de la misma aplicación. Esto hace al sistema más eficiente y hace a la arquitectura más disponible, ya que el fallo de un microservicio no desemboca necesariamente el fallo de otros. La habilidad de replicar servicios individuales junto con la de localizar uno o más microservicios en un único anfitrión hace a la arquitectura microservicio elástica, es decir, escalable dinámicamente en función de la carga.

2.1.2.2. Escalado no uniforme

Es difícil analizar qué componentes de una arquitectura necesitan escalar ante la demanda de trabajo, ya que el sistema se ejecuta en un único proceso, por tanto, aunque solo un componente experimente la carga, todo el conjunto debería de escalar. Implementando microservicios independientemente unos de otros, estos pueden ser monitoreados y escalados independientemente.

2.1.2.3. Despliegue y portabilidad

Los microservicios normalmente se empaquetan en contenedores, esto es una unidad estándar de software que empaqueta en un "bloque"todo su código y sus dependencias para que la aplicación se ejecute rápidamente en diferentes entornos. Gracias a las ventajas de los contenedores, al incluir todo su entorno, consiguen que su comportamiento sea totalmente uniforme en otros sistemas. Esta portabilidad asegura la recolocación y replicación de un microservicio a lo largo de plataformas heterogéneas.

2.1.2.4. Disponibilidad

Esta característica se consigue con los microservicios debido a la habilidad de replicación y propagación a través de centros de datos ubicados en diferentes ubicaciones, permitiendo propagar cargas a través de todo el hardware disponible. Además, el poder actualizar cada microservicio independientemente del resto supone una gran ventaja, hablando en términos de tiempo de disponibilidad, ante las aplicaciones que necesitan actualizarse y necesitan parar todo el sistema y desplegarlo de nuevo.

2.1.2.5. Robustez

Al igual que con la disponibilidad, la robustez también es conseguida con el enfoque microservicio. La tolerancia a fallos es mejorada debido al uso de contenedores, siendo cada uno de los microservicios aislado de los otros, y, por tanto, al aparecer errores estos solo involucran al mismo único servicio en el que se produjeron, y no provocarán fallos en cadena en el resto de servicios.

2.1.3. Enfoque de los microservicios para el Internet de las Cosas

Los frameworks limitan la elección de tecnologías con el objetivo de facilitar el desarrollo de aplicaciones del Internet of Things (IoT), aunque usando estos sólo se facilita realmente el desarrollo de servicios individualmente, y el conjunto de todos estos servicios, que se caracterizan por diferenciarse unos de otros en términos de hardware y tecnologías usadas [2].

Cuando se habla de ecosistemas de IoT, estos limitan mucho más los servicios e interfaces que se deben implementar, pero por otra parte garantizan mucha más interoperabilidad. Sin embargo, esta interoperabilidad sólo se da en ese ecosistema, ejemplos de esto son Google o Apple, cada uno en sus respectivos dominios de Smart home.

Para enfocar la tecnología de los microservicios al Internet de las cosas, se compararán los diferentes patrones y prácticas del enfoque microservicio con las del IoT. Se compararán aspectos como el de contenedor propio, el

monitoreo y prevención de los fallos en cascada, la coreografía y orquestación, las tecnologías de los contenedores y la gestión de versiones.

2.1.3.1. Contenedor propio

Se trata de una característica esencial en el enfoque microservicio. Un contenedor es un servicio que se despliega junto todas sus dependencias y aislados del resto del resto del software del sistema gracias a la tecnología de los contenedores. Gracias a esto los servicios pueden ser escalados individualmente y crear varias instancias a la vez de ellos, además de que se reducen las dependencias con otros servicios. Aunque sin embargo los servicios tampoco pueden ser muy amplios ya que esto dificulta su mantenimiento; este tamaño de servicios aún no está perfectamente definido [2]. Este tamaño en este caso se refiere a una única funcinalidad, y no a varias. Es decir, un servicio es considerado pequeño si posee una funcionalidad muy limitada, aunque ocupe mucho espacio o tenga muchas líneas de código.

Mientras tanto, en el internet de las cosas, muchas aplicaciones orientadas a servicios ya estaban muy cerca de este principio, ya que la naturaleza de, por ejemplo, sensores y actuadores ya está desarrollada alrededor de este principio (la información de cada sensor está almacenada tanto tiempo como permita el sensor).

Si este principio se aplicase totalmente al IoT se tendrán las siguientes ventajas [2]:

1. Al tener la parte back-end dentro del servicio, en los microservicios, se intenta eliminar las dependencias externas lo máximo posible. Así, los datos no estarían tan accesibles para las peticiones externas y estas se limitarían a una API que no permitiría a los consumidores de datos externos conocer de la representación interna de estos datos.

2. Si cada servicio tiene su propia interfaz de usuario se potencia la adaptación individual de este, además en el caso de un front-end centralizado, no se tiene que tener en cuenta cada nuevo servicio que se agregue a este, ya que cada servicio tiene su forma de representación y se pueden representar directamente en un panel dinámico en este front-end centralizado.

3. Al insertarse las librerías requeridas dentro del servicio el despliegue de este es mucho más fácil, eliminando la instalación de dependencias.

2.1.3.2. Monitoreo y prevención de fallos en cadena

Cada servicio debe de disponer de una interfaz para manejar la información de monitoreo; es de vital importancia que se sepa el estado de este servicio (todo correcto o apariciones de errores). Esto es muy importante,

ya que otros servicios pueden percatarse de esto y se previenen las llamadas a servicios no disponibles, y se declaran a sí mismos como no disponibles. Para esto se utilizan patrones como el "Circuit Breaker" [4], que comprueba el estado del servicio continuamente o recuerda el número de llamadas sin éxito hasta que se alcanza un umbral determinado, cuando detecta fallo devolverá un error y cesará las llamadas a servicios externos. Este patrón funciona muy bien con el balanceador de carga: este distribuye el peso de trabajo entre un conjunto de servicios similares. El "Circuit Breaker" activa el balanceador de carga para poner en activo a servicios que tienen buen estado, es decir, servicios que su "Circuit Breaker" está abierto y por tanto los servicios rotos no serán usados.

Por otro lado, en el IoT se pueden utilizar ambos (el balanceador de carga y el Circuit breaker) o cualquiera individualmente. Ambos patrones han demostrado ser una buena forma de manejar la falla de servicios remotos. Con respecto al Circuit Breaker, previene el envío de mensajes innecesarios a servicios no disponibles, reduciendo la carga de la red y en consecuencia el ahorro de energía, y respecto al balanceador de carga incrementa la vida útil de los sensores ya que la carga de trabajo está compartida entre varios dispositivos, permitiendo estar más tiempo en el modo de bajo consumo de energía.

Y en relación con el sistema de "logging", en la arquitectura microservicio se recomienda un único formato en todos los servicios, para conseguir una perspectiva global de todo el sistema. Mientras que, en el IoT, no es posible ya que no es posible tanto control en este escenario compuesto de diversos dispositivos, sin embargo, se debería utilizar un formato común con facilidades de integración con otros registros.

2.1.3.3. Coreografía sobre orquestación

Una orquestación se define como un conjunto en el que uno o varios servicios centralizados, mientras que en una orquestación, cada servicio que la compone define con quién interactúa y cuando interactúa. Cada una de estas opciones tiene una serie de ventajas y desventajas.

En la **arquitectura microservicio**, una coreografía implica un mayor grado de libertad en la forma en que las cosas pueden ser tratadas, no se requiere de un "director" en el caso de agregar un nuevo servicio al conjunto, esto simplifica las tareas sobre todo cuando ese director es un producto de otro vendedor en el caso de un escenario IoT. Sin embargo, utilizando una coreografía no hay una instancia que pueda comprobar si todas las acciones requeridas se han realizado correctamente desde un primer momento. Para solucionar este aspecto, es posible añadir un servicio adicional que solo monitoree los servicios que se ejecuten.

Desde el punto de vista de los **servicios IoT**, normalmente son combinados utilizando el método orquesta, debido a su facilidad de implementación

y a que protocolos como HTTP no tienen soporte nativo para la comunicación basada en eventos. La idea de la coreografía en los microservicios puede servir de punto de partida para los servicios y aplicaciones del internet de las cosas. Los servicios básicos son modelados con independencia usando comunicación basada en eventos. Los servicios de valor añadido deben comprobar que todos los servicios de una aplicación están presentes y escuchando a sus correspondientes eventos, así, la aplicación se puede dar cuenta de los fallos y solventarlos.

2.1.3.4. Espacios aislados

Para esto se utilizan los denominados "namespaces", es decir, agrupar los recursos del sistema para representar a los procesos como si tuvieran su propia instancia, tablas de routing, identificadores de procesos, interfaces de red... Los cambios en un recurso solo son visibles a los procesos que se utilizan esa agrupación. En la perspectiva de los microservicios es de gran ayuda que los servicios individuales se puedan tratar como un contenedor individual. Como ventajas se pueden diferenciar:

1. **Mejores pruebas**. Las pruebas se podrán hacer en todo el contenedor.

2. **Facilidad de despliegue**. Como cada contenedor incluye al servicio y todas sus dependencias, no es necesario preocuparse por las diferentes librerías o versiones que necesitan ser instaladas, cada servicio tiene ya todo empaquetado

3. **Mejor escalabilidad**. Siendo así de desacoplados los servicios, cada servicio puede escalar individualmente empezando o terminando múltiples instancias.

En el internet de las cosas también se puede poner en práctica el uso de contenedores, pero depende más de las tecnologías utilizadas y el escenario, existe mucha limitación de memoria en los pequeños dispositivos empotrados. Un escenario en el que poder utilizar contenedores sería el Edge computing, los dispositivos sí serían capaces de ejecutar contenedores.

2.1.4. Desventajas de los microservicios

Todas estas ventajas descritas en los apartados anteriores pueden hacer creer que los microservicios son la perfecta solución, pero esto no siempre es así. El hecho de tener un gran conjunto de entidades independientes añade una mayor dificultar a la hora de gestionar y distribuir cada una de ellas. Además de la dificultad de despliegue, el monitoreo de cada una de estas entidades, o la seguridad de ellas. Es necesario valorar el entorno en el que se

quieren utilizar los microservicios y si presenta más ventajas que problemas utilizar esta tecnología.

2.2. Cloud Computing

En esta sección se analizará la definición de **Cloud Computing**, de sus atributos, modelos y formas de despliegue, además de los obstáculos y ventajas a los que se enfrenta esta tecnología; y, por último, al igual que en el apartado anterior, se enfocará esta tecnología al IoT, debido a su estrecha relación con la temática de este Trabajo final de Máster.

2.2.1. Definiciones

Armbrust et al. [6] definían el término de Cloud Computing como la combinación de las aplicaciones publicadas como servicios y el hardware y el software de los sistemas del centro de datos que provee estos servicios. Se habla de las ventajas que tiene, en resumen, la posibilidad de adquirir unos recursos en base a la demanda, de modo que conforme se requiere más potencia, se paga por este incremento y no antes de necesitarlo. Se puede diferenciar lo que es y lo que no es Cloud Computing en base al tiempo en el que es posible escalar los recursos en función de la demanda: debe de ser en términos de minutos.

Se describen las nuevas "cloud functions", que están a la orden del día, siendo el núcleo de la computación sin servidor ("serverless computing") y constituyendo un modelo de programación orientado a la nube. Se comparan además los dos términos de computación en la nube: "Serverless Cloud" y "Serverful Cloud", donde el primer término se refiere al desarrollo de aplicaciones sin tener en cuenta la mayoría de los procesos que ocurren debajo de cada acción que lleva a cabo el usuario y a través de un lenguaje de programación de alto nivel; y el segundo hace referencia al desarrollo de aplicaciones en las que el desarrollador es responsable de todo el escalado, monitoreo, registros, instancias de servidor... [7]

El Cloud Computing es un modelo para activar el acceso en red ubicuo, conveniente y por demanda a un conjunto de recursos de computación configurables que pueden ser rápidamente almacenados y publicados con la mínima gestión y esfuerzo y con la mínima interacción con el proveedor de servicio [8].

2.2.2. Atributos, modelos y formas de despliegue

El National Institute of Standards and Technology (NIST) describe cinco características de un modelo de computación en la nube [9]:

- **Servicio automático por demanda**. El tiempo del servidor, el uso de la red, y otros recursos de computación se consiguen conforme se necesiten, no es necesaria la interacción con el proveedor de servicios.
- **Amplio acceso a la red**. Los recursos están disponibles en la red y pueden ser accedidos desde cualquier tipo de dispositivo conectado a la red.
- **Puesta en común de recursos**. Los recursos se distribuyen entre múltiples usuarios, se asignan dinámicamente en función de la demanda.
- **Rápida elasticidad**. Los recursos se escalan en función de la demanda.
- **Servicio medido**. Para optimizar el uso de los recursos se miden parámetros como el almacenamiento, procesamiento, ancho de banda, cuentas de usuario activas...

En función de las necesidades del consumidor, existen cinco alternativas en la nube que se diferencian entre sí en la complejidad y limitación que ofrecen en su infraestructura, siendo el software como servicio el más utilizado por pequeños negocios, y el resto para otros requerimientos más complejos [10]:

- **Database as a Service (DBaaS)**. Se trata de un servicio en la nube donde la base de datos se ejecuta en la infraestructura del proveedor de servicios. Este modelo ofrece escalabilidad instantánea, garantías de funcionamiento, última tecnología, control de fallos y precio reducido.
- **Desktop as a Service (DaaS)**. Este modelo de servicios permite desplegar un sistema operativo de escritorio vía remota junto a sus aplicaciones correspondientes.
- **Software as a Service (SaaS)**. Sólo ejecuta aplicaciones desarrolladas por el proveedor de servicio. Los usuarios de este servicio no tienen control de la infraestructura, y estos servicios pueden ser accedidos desde un gran rango de dispositivos móviles y tipos de clientes. Esta es la modalidad más utilizada por los pequeños negocios.
- **Platform as a Service (PaaS)**. Este ofrece desplegar aplicaciones desarrolladas usando herramientas y lenguajes de programación que da el proveedor, el usuario no gestiona la capa inferior (redes, servidores, sistemas operativos...). Esto limita la portabilidad de las aplicaciones, ya que dependen de estas herramientas y lenguajes del proveedor para funcionar correctamente.

- **Infraestructure as a service (IaaS)**. Ofrece la capacidad de provisionar procesamiento, almacenamiento, gestión de redes y otros recursos computacionales fundamentales. El consumidor puede desplegar software que puede incluir sistemas operativos y aplicaciones. Ejemplos de servicios que ofrecen este modelo son hosting de servidores, servidores web, almacenamiento, hardware computacional, sistemas operativos...

En cuanto a las formas de desplegar una nube, se pueden diferenciar cuatro formas (Marinescu, 2017) [8]:

- **Nube privada** (*Private Cloud*). Provee los servicios de computación necesarios por una gran organización, por ejemplo, una institución de investigación, una universidad o una corporación.
- **Nube comunitaria** (*Community Cloud*). Es compartida por varias organizaciones que tienen objetivos comunes.
- **Nube pública** (*Public Cloud*). La infraestructura es abierta al público general o a un gran conjunto de industrias y es administrada por una organización que vende servicios cloud, como por ejemplo: AWS, Microsoft Azure, Google Apps...
- **Nube híbrida** (*Hybrid Cloud*). Su infraestructura es una composición de dos o más nubes (privadas, comunitarias o públicas), que son individuales, pero a la vez forman un conjunto a través de una tecnología propietaria que permite la portabilidad de aplicaciones.

2.2.3. Obstáculos para el Cloud Computing

A la hora de desplegar un servicio de Cloud Computing, se presentan una serie de obstáculos que se deben tener en cuenta y sus posibles soluciones [6]:

- **Disponibilidad del servicio**. Los usuarios tienen unas expectativas muy altas hablando de la disponibilidad de los servicios. Cualquier solicitud que no sea atendida es suficiente para suponer un gran problema para la empresa. Como solución a esto, se recomienda que las empresas tengan diferentes proovedores de servicios; así, si se da algún problema con uno de ellos, no supondrá la caída de todo el servicio.
- **Extracción de datos**. Como solución al problema de no poder extraer los datos en un determinado momento al ser almacenados por una empresa externa, se puede estandarizar una API para que extraiga los mismos datos de diferentes empresas que almacenen esos datos.

- **Responsabilidad en los datos**. En función del nivel de cloud computing en el que se esté trabajando, la responsabilidad sobre este aspecto recaerá en mayor o menor medida o bien en el desarrollador de la aplicación, o bien en la empresa que mantiene la aplicación.
- **Escalabilidad en el almacenamiento**. Para conseguir esto se puede crear un sistema de almacenamiento que combine ciertos aspectos, como la robustez y alta disponibilidad de los datos, y la habilidad de administrar y obtener datos, con las ventajas que ofrece la nube. Por ejemplo, el escalado en función de la demanda.
- **Errores en los sistemas distribuidos de gran escala**. Muchos errores aparecen en la fase de producción, en los centros de datos, y no en las pequeñas configuraciones en la fase de desarrollo. Para solucionar esto se propone el uso de máquinas de virtualización que puedan recrear este tipo de errores.

2.2.4. Riesgos de seguridad

Respecto a los riesgos o amenazas para la seguridad del sistema a los que se enfrentan los usuarios de la nube, se pueden diferenciar tres clases [8]:

- **Amenazas tradicionales**. Son las que ha tenido cualquier sistema conectado a Internet, pero con algunos añadidos por el factor de la nube. El impacto de estas es amplificado debido a la gran cantidad de recursos que ofrece la nube y la gran cantidad de usuarios que puedan verse afectados. Se deben gestionar adecuadamente los roles de acceso a una nube, para evitar mayores daños en el sistema.
- No **disponibilidad de los servicios**. Cualquier fallo del sistema, fallo en la electricidad o cualquier evento similar puede provocar largos periodos de desactivación del servicio.
- **Control de terceros**. Un proveedor de la nube puede subcontratar recursos de terceras empresas cuyo nivel de confianza puede ser cuestionable, y por tanto, suponer una amenaza para todo el sistema.

2.2.5. Cloud Computing y los sistemas de salud

Por todo lo indicado hasta ahora, el Cloud Computing ofrece la necesaria potencia de computación, almacenamiento, aplicaciones y sistema de conexión para soportar aplicaciones que manejan una gran cantidad de información, como requieren los sistemas de salud. Sistemas como, por ejemplo, el que desarrolló IBM junto con el proveedor de servicios de salud estadounidense *Active Health Management*, que provee información accesible basada en el Cloud Computing, información de diferentes orígenes como datos de

laboratorio y registros electrónicos médicos. Se introducen reacciones a tratamientos, y se alerta automáticamente de conflictos o problemas en los medicamentos, además se añaden analíticas para medir la efectividad de los tratamientos aplicados a los pacientes entre diferentes hospitales [11].

2.2.5.1. Sistemas de salud e Internet de las Cosas

Desde que se comenzó a implantar el IoT en las áreas médicas, ha demostrado tener mucho potencial y ser una tecnología muy efectiva en este campo. Aunque aún quedan objetivos por conseguir, como, por ejemplo, sistemas de automejora, dispositivos hardware menos intrusivos (como vestibles y sensores implantables), estandarización, privacidad y seguridad. Otro aspecto crítico es la integración de este sistema con otras tecnologías. Las principales ventajas de esto son:

- **Servicio personalizado**: las peticiones de los pacientes son atendidas sin necesidad de que una una persona medie en el servicio prestado.
- **Acceso a la red**: las aplicaciones usadas por los pacientes pueden ser accedidas desde cualquier dispositivo.
- **Acceso remoto**: los recursos de la nube están ocultos ante los pacientes, estos acceden a los servicios sin saber la ubicación de los datos.

2.2.5.2. Obstáculos del Cloud Computing en los sistemas de salud

Respecto a los obstáculos o desventajas que tiene esta tecnología para los sistemas de salud, se pueden diferenciar varios aspectos [11]:

- **Recursos compartidos**. Al permitir compartir sistemas e infraestructuras, la integridad del sistema en la nube se puede ver comprometida, dado que se procesan datos personales, y cualquier conflicto puede poner en riesgo la privacidad y seguridad de estos datos.
- **Falta de confidencialidad**. Aumenta el riesgo de que los datos personales puedan ser revelados a agencias extranjeras con diferentes leyes de protección de datos y, por tanto, puedan provocar brechas de datos.
- **Necesidad de subcontratación de otros proovedores**. Debido al alto nivel de complejidad y dinamismo, se requiere la subcontratación de modo que un servicio de un proveedor de cloud incluye una combinación de servicios de varios proveedores.
- **Falta de acceso**. Un proveedor de la nube puede no dar a los clientes métodos adecuados para gestionar y trabajar con los datos que necesitan.

- **Falta de informacción sobre las operaciones de procesamiento**. El nivel de abstracción que presenta esta tecnología, además de ser una ventaja, es una desventaja ya que el desarrollor desconoce las operaciones de procesamiento que se llevan a cabo en capas inferiores. Esto incrementa el riesgo para los clientes de la nube si no se tiene en cuenta este aspecto.

- **Falta de privacidad**. Un proveedor de la nube podría tratar la información personal de clientes de los sistemas para utilizarlos con otros fines, como por ejemplo la venta a terceros. A diferencia de los administradores de los sistemas de la salud que tienen sólo ciertos permisos de acceso.

2.2.5.3. Integración del IoT en el Cloud Computing: CloudIoT

En 2016 se introdujo el concepto de "**CloudIoT**" como integración del Cloud Computing con el IoT. Se trata de una combinación de herramientas para conectar, procesar, almacenar y analizar datos del entorno en la nube. [11].

Como desventaja, el IoT tiene una gran limitación de seguridad debido a la naturaleza de su red e infraestructura, mientras que el Cloud Computing tiene un gran nivel de seguridad, debido a la forma de almacenar la información en los centros de datos. Por esto, con el paso de los años se han creado una gran cantidad de proyectos de investigación en diversas áreas, y como consecuencia han emergido múltiples plataformas, protocolos y sistemas.

Los desafíos a los que se enfrentan los escenarios IoT en la nube son, en primer lugar, el análisis efectivo de los datos generados por los sensores en tiempo real, y en segundo lugar, realizar análisis de datos históricos útiles en forma de información estructurada o desestructurada.

Ambas tecnologías, el Cloud Computing y el IoT, se caracterizan por ser muy complementarias, por ello existe tanta necesidad de integrarlas entre sí. El IoT puede utilizar los recursos del Cloud Computing para compensar sus carencias, como las comunicaciones, almacenamiento y procesamiento; y además el Cloud Computing ofrece una forma efectiva para explotar los datos producidos por los sensores. Por otro lado, el Cloud Computing ofrece una capa intermedia entre las aplicaciones y los objetos.

El IoT se caracteriza por el grado de heterogeneidad que tiene, debido a su elevado número de dispositivos y protocolos, por lo que carece de propiedades que sí tiene el Cloud Computing, como son la escalabilidad, interoperabilidad, eficiencia, disponibilidad y seguridad. Por tanto, el CloudIoT tiene las siguientes ventajas [11]:

- **Comunicación**. Debido al paradigma del CloudIoT, se ofrece una forma barata y efectiva de gestionar las aplicaciones, conectando y monitorizando todo a través de pasos de mensajes entre dispositivos.

- **Almacenamiento**. El IoT produce una gran cantidad de información sin estructurar o semiestructurada. Esta información se almacena de forma efectiva, gracias a las capacidades del Cloud Computing: almacenamiento a gran escala, a bajo coste y por demanda.
- **Computación**. Los dispositivos del IoT tienen capacidad muy limitada de procesamiento, que no permite que este sea complejo. Por tanto, los datos recogidos se suelen enviar a nodos más potentes, pero la escalabilidad es un gran desafío a superar sin una buena infraestructura. Por esto, la nube ofrece una capacidad virtual ilimitada para procesar toda esta información.
- **Alcance**. Los dispositivos están en constante desarrollo y mejora, por lo que más tipos de información deben de ser tenidos en cuenta. Por ello adoptando este nuevo paradigma permite nuevos servicios inteligentes basados en la extensión de la nube que hace posible enfrentarse con nuevos escenarios en la vida real, dando lugar al paradigma de *Things as a Service*, que es un paradigma que se basa en las necesidades características de los dispositivos del IoT, como por ejemplo el descubrimiento de servicio o los mecanismos de composición.

2.2.5.4. Ejemplos de aplicaciones CloudIoT

Existen dos propuestas de aplicaciones CloudIoT dirigidas a la salud:

- **OpenIoT** [12]. Middleware de código abierto para obtener información de los dispositivos a través del paradigma de *Things as a Service*.
- El proyecto **SENSEI** [13]. Integra sensores y actuadores inalámbricos heterogéneos en una arquitectura abierta que ofrece servicios y aplicaciones. Entorno seguro donde los usuarios pueden administrar y acceder a información sobre estos dispositivos.

2.2.5.5. CloudIoT en el campo de la salud

Hoy en día es una necesidad reducir los costes en los hospitales y trasladar servicios, como ciertas revisiones médicas, a casa, además de trabajar en la toma automática de decisiones médicas a través de los datos y diagnósticos de los pacientes. El paradigma del CloudIoT integrado en el campo de la salud propone analizar las constantes vitales que pueden prevenir situaciones anormales, revisar los historiales médicos para prevenir futuras enfermedades, además de proponer tratamientos personales basados en el historial médico y genérico. Este paradigma, aplicado al campo de la salud, permite mejorar la calidad del sistema médico al facilitar la cooperación entre las diferentes entidades involucradas en este. Provee unos servicios médicos de alta calidad y de bajo coste. Sus ventajas en este campo son las siguientes:

- **Rastreo**. Identifica al paciente en movimiento, así como sus posiciones en tiempo real, mejorando el flujo de trabajo en los hospitales.
- **Identificación y autentificación**. Esto reduce las equivocaciones con respecto a medicamentos, dosis, intervalos...
- **Toma de datos**. Al ser automática y cuando se precisa, se reduce el tiempo de procesamiento, así como una atención instantánea por parte de los sistemas de salud.
- **Sensores**. Estos dispositivos permiten al personal sanitario dar diagnósticos particulares centrados en cada paciente, ofreciendo información en tiempo real de los indicadores de salud del paciente.

2.3. Web Service Composition

Dentro de una composición de Servicios Web, estos independientemente llevan a cabo actividades individualmente y en conjunto tienen un objetivo final, donde el éxito de la composición depende de la correcta sincronización entre sus correspondientes servicios. En este apartado trataré sus características básicas, sus beneficios, sus requisitos, de cómo funciona el descubrimiento y composición de servicios, de varios ejemplos de enfoques actuales, y para finalizar enfocaré también este tema con el Internet de las Cosas, hablando de su enfoque con respecto a la composición de servicios .

Una composición de servicios permite la comunicación, cooperación y coordinación entre aplicaciones separadas desde diferentes orígenes en Internet, y junto con la agregación de servicios se pueden llegar a crear servicios integrados a medida para trabajar con tareas repetitivas. Esta tecnología está presente en cada vez más sistemas que se utilizan a diario.

2.3.1. Características

Para que una composición de servicios sea buena, las secuencias de procesos para llevar a cabo diferentes tareas no se deben de definir cada vez que se realizan, sino que se describe una forma de instanciar y gestionar los procesos para generar una solución más eficiente y flexible. Cada composición de servicios define un proceso de negocio concreto. Cada servicio es una tarea, y la composición de las mismas da lugar a un proceso completo. Algunas de las características fundamentales de las composiciones de servicios son [14] [15]:

- Los servicios web no son como librerías de aplicaciones que tienen que ser compiladas y enlazadas como parte de una aplicación. Son componentes que deben de tener "forma"de ejecutable.
- Los componentes básicos (servicios individuales) permanecen separados del servicio compuesto.
- Una composición de servicios Web involucra la especificación de los servicios que necesitan ser invocados, en qué orden, como tratar excepciones...
- Una composición de servicios Web puede ser anidada, estos servicios pueden ser considerados como bloques de construcción que pueden ser ensamblados. Esto permite construir aplicaciones complejas agregando progresivamente componentes. Cuando un servicio compuesto queda revelado como servicio web, puede ser considerado como otro bloque de construcción y, en consecuencia, puede ser usado en otra composición.

Estas aplicaciones son rápidamente desplegables y ofrecen a los desarrolladores la posibilidad de reusar y ofrecer a los usuarios el acceso a una gran variedad de servicios complejos. Hay varios enfoques de composición de servicios, desde métodos abstractos a los estándares industriales.

2.3.2. Beneficios de una composición de servicios web

Las razones para adoptar la composición de servicios como el método para desarrollar nuevas aplicaciones se pueden enumerar en 5 puntos [14]:

- **Integración**. Es un requisito general que permite a los servicios interactuar y trabajar juntos. Con las tecnologías de composición de servicios las organizaciones pueden interoperar y trabajar como si fueran un único sistema.
- **Vista global**. Sin componer los servicios no hay una vista global, cada servicio trabaja por su cuenta; componiéndolos se crea una, lo cual

incrementa la productividad y reduce costes. Esto se da normalmente en conjuntos de servicios compuestos mediante una orquestación. La coreografía a veces no permite derivar en una vista global del sistema.

- **Rol extendido de las tecnologías de la información** (IT). El hecho de proveer entornos donde los servicios pueden ser compuestos facilita la participación de gente no necesariamente de IT en el proceso. Como se indicó al comienzo de esta sección (2.3.1), cada composición se encarga de un proceso de negocio.
- **Estandarización**. La composición de servicios Web se basa en estándares, que permiten un entorno donde todas las partes pueden interactuar de manera homogénea, lo cual reduce riesgos y costes.

2.3.3. Requisitos

Las características de una composición ideal serían las siguientes [15]:

- **Exactitud**. Uno de los mayores requisitos para un conjunto de servicio ideal es producir resultados exactos. Los servicios descubiertos y compuestos deben de satisfacer todos los requisitos de una consulta, además el motor debe de ser capaz de encontrar todos los servicios que satisfagan los requisitos de la consulta.
- **Mínimo tiempo de ejecución de la consulta**. Consultar un repositorio de servicios para un servicio solicitado debería ser realizado en escaso tiempo (a nivel de milisegundos), asumiendo que el repositorio de servicios debe estar preprocesado y preparado para la consulta.
- **Actualizaciones incrementales**. Añadir o actualizar un servicio debe durar lo menos posible.
- **Función de coste**. Si hay costes asociados a cada servicio en el repositorio, entonces un motor ideal de composición debe ser capaz de dar resultados basados en los requisitos en función de los costes.

2.3.4. Descubrimiento y composición automática de servicios web

El descubrimiento y la composición son dos importantes tareas relacionadas con los servicios web (Bansal et al., 2014) [15].

2.3.4.1. Descubrimiento

El problema del descubrimiento de servicios web se basa en, dado un repositorio de servicios web y una consulta solicitando un servicio, buscar automáticamente un servicio del repositorio que encaje con los requisitos

funcionales determinados. Sólo aquellos servicios que producen al menos los parámetros de salida que satisfacen las postcondiciones y sólo utilizan los parámetros de entrada que satisfacen las precondiciones y a la vez producen una solución válida son aptos para la consulta.

2.3.4.2. Composición

En este caso, el problema de la composición de servicios se basa en, dado un repositorio de descripciones de servicios, y una consulta con los requisitos de un servicio solicitado y en el caso de no encontrar el servicio, buscar un grafo de servicios dirigido acíclico que puedan ser compuestos entre sí para obtener el servicio deseado.

2.3.5. Técnicas para la composición de servicios

Se presentan diversos enfoques para la composición de servicios [16]:

- **BPEL**. Es un lenguaje XML que soporta composición de servicios orientado a procesos. Fue desarrollado por BEA, IBM, Microsoft, SAP y Siebel y está siendo estandarizado por la Organization for the Advancement of Structured Information Standards (*OASIS*). Interactua con un conjunto de servicios web para conseguir una tarea.

- **Semantic Web** (OWL-S). Su objetivo es hacer a los recursos web accesibles por el contenido, así como por palabras clave. Los servicios web juegan un gran papel en esto: los usuarios y los agentes software deben de ser capaces de descubrir, componer y invocar contenido utilizando servicios complejos. OWL-S es una ontología de servicios que permite el descubrimiento, invocación, composición, interoperación, ejecución y monitorieo automático de servicios.

- **Componentes Web**. Este enfoque trata a los servicios como componentes para soportar principios básicos de desarrollo de software como por ejemplo la reutilización, especialización y extensión. La principal idea es la de encapsular información de composición dentro de la definición de una clase, que representa un componente web. La interfaz pública de un componente puede ser publicada y utilizada para el descubrimiento y reutilización.

- **Algebraic Process Composition**. La composición de servicios algebraica introduce descripciones más simples que otros enfoques, y modelar servicios como si fueran procesos móviles para verificar propiedades como la seguridad, los ciclos de vida y la gestión de recursos.

- **Redes de Petri**. Se trata de un enfoque de modelado de procesos bien establecido. Es un gráfico dirigido, conectado y bipartido con nodos

que representan lugares y transiciones, y un token que ocupa estos lugares. Cuando hay al menos un token en cada lugar conectado a una transición, esa transición está disponible. Una conexión disponible se activa eliminando un token de cada lugar de entrada, y poniendo un token en cada salida. Es posible modelar servicios como redes de Petri asignando transiciones a métodos y lugares a estados.

- **Comprobación de modelos y máquinas de estados finitos**. Otros enfoques para la composición de servicios web incluyen comprobación de modelos, modelamiento de composición de servicios y composición automática de máquinas de estados finitos. La comprobación de modelos se utiliza para verificar formalmente los sistemas concurrentes de estados finitos, se puede aplicar a la composición de servicios software para verificar que un flujo de trabajo está correcto.

Los enfoques de la composición de servicios van desde los que aspiran a convertirse en estándares industriales (BPEL y OWL-S) a otros más específicos. Un enfoque ideal sería aquel que cubra los cuatro requisitos fundamentales explicados en la sección anterior, pero el principal problema de los enfoques industriales es la correcta verificación. Los enfoques formales son normalmente difíciles de aplicar a nivel industrial, además de enfrentarse a numerosos problemas de escalabilidad.

Estos problemas estarán presentes en la actualidad durante un tiempo, pero a corto plazo una solución sería adoptar un estándar industrial; y a largo plazo seria incorporar mecanismos de verificación que escalen correctamente y permitan a los desarrolladores y usuarios enfocarse en efectuar tareas diarias usando los servicios web sin preocuparse de otros factores externos.

2.3.6. Enfoques de la composición de servicios en el IoT

Los procesos de composición de servicios permiten la interacción entre los requisitos del usuario y los objetos inteligentes del Internet de las cosas [18]. Consecuentemente, elegir servicios aptos es el principal desafío que abarca a la calidad de funcionamiento y calidad requerida para combinar varios servicios como el servicio integrado compuesto en el IoT.

Desde que la infraestructura IoT tiene una estructura dinámica y heterogénea, la composición de los servicios existentes de la nube no puede coordinar los factores de la Quality of Service (QoS) y los acuerdos del nivel de servicios (Service-Level Agreement, SLA).

El enfoque de la composición de servicios tiene una manera uniforme de considerar los requisitos de usuario basados en peticiones, proveedores y procedimientos de negocio. Cada servicio está relacionado con el dispositivo

IoT que ofrece la funcionalidad de ese servicio. En la mayoría de casos un único servicio no es suficiente para llevar a cabo los requisitos complejos del usuario, por eso algunos de los servicios complejos son ofrecidos por un conjunto de dispositivos IoT, por ello, es necesaria la composición de servicios. Una composición de servicios controla un conjunto de servicios en forma de uno único para llevar a cabo un objetivo común.

2.3.6.1. Factores clave en una composición de servicios en IoT

Cada composición de servicios tiene unos factores de calidad de servicio clave para ser considerada la mejor composición entre los servicios candidatos, para ello se tienen en cuenta cinco aspectos importantes:

- **Disponibilidad**. La disponibilidad de la composición de servicios debe de estar presente en varios servicios en cualquier intervalo de tiempo y en cualquier lugar en función de la tareas del usuario.
- **Tiempo de respuesta**. La duración del tiempo que tarda la composición en responder a una acción del usuario.
- **Escalabilidad**. Este factor describe la habilidad de añadir nuevas operaciones y nuevos dispositivos como nodos de servicio para las tareas del usuario sin decrementar la calidad de los servicios que ya se encontraban previamente en el conjunto.
- **Coste**. El precio total que tiene que ser pagado por el solicitador del servicio para llegar a la mejor composición de servicios.
- **Fiabilidad**. El objetivo de la fiabilidad es el de llegar a la mejor distribución de servicios.

2.3.6.2. Arquitectura por capas de IoT basado en servicios

Esta arquitectura consiste en cinco capas diferentes [18]:

- **Capa de percepción**. Se incluyen los sensores y los dispositivos inteligentes utilizados para obtener los datos de un entorno IoT.
- **Capa de red**. Conecta los dispositivos inteligentes y sensores a los otros servidores.
- **Capa de la nube**. Está compuesta de varios subservicios de varias nubes privadas o públicas.
- **Capa de composición de servicios**. Es la responsable de componer un número de subservicios juntos en función de los requisitos funcionales y no funcionales del usuario. Esta capa se ocupa de coordinar

las peticiones para enviarlas en función de las demandas del usuario a una o más nubes en esta capa y seleccionar finalmente los subservicios que serán combinados para componer la composición de servicios que será desplegada finalmente en la capa de aplicación.

- **Capa de aplicación**. Conduce las composiciones de servicios al usuario final en función de sus solicitudes.

2.3.6.3. Técnicas de composición de servicios en IoT

Se valorarán las principales técnicas de composición de servicios en el ámbito del internet de las cosas, tanto como sus discrepancias, ventajas y desventajas. La mayoría de artículos de esta área consisten en estas cuatro categorías. Estas se diferenciarán en función de parámetros como el tiempo de ejecución, el tiempo de respuesta, la disponibilidad, la fiabilidad, la escalabilidad, el coste y la reputación en función de las experiencias de los usuarios [20].

- **Técnicas basadas en frameworks**. Se basan en una serie de suposiciones, valores, teorías y prácticas que se organizan para descubrir, seleccionar y componer servicios IoT. Por ejemplo, uno de los enfoques es el de F. Khoadadi et al. [19], que simplifica la comunicación entre las entidades formando un entorno ubiquio de objetos IoT y describe cómo usar archivos orientados al usuario para detectar y responder fácilmente y rápidamente usando interfaces de programación de aplicaciones web estándar. El objetivo principal es ganar en simpleza de diseño, apertura y seguridad al mismo tiempo. El framework debe de soportar el desarrollo de IoT completamente, ayudando a desarrollar, administrar y reutilizar patrones utilizados por muchos programadores en el IoT.

- **Técnicas basadas en heurísticas**. Aunque hay algoritmos eficientes basados en analítica que resuelven muchos problemas, no hay algoritmos eficientes para resolver problemas de optimización combinatorios discretos de and/or. Es esencial utilizar heurísticas o metaheurísticas desde que los algoritmos dependen de las necesidades computacionales. Un método heurístico que se utiliza para descubrir una solución a un problema de optimización no asegura obtener las mejores soluciones.

- **Técnicas basadas en modelos**. Se basan en los modelos computacionales para simular las acciones. En esta técnica, los modelos se aplican para describir y explicar los componentes y las interacciones entre ellos, además de las estructuras de información, los requisitos de usuario y las preocupaciones de negocio.

- **Técnicas basadas en SOA y RESTful**. SOA es un método utilizado para crear una arquitectura para usar servicios, estos realizan una pequeña operación como validar un cliente, producir datos, o dar un servicio simple. Los usuarios pueden organizar la colección de estos servicios en actividades de negocio. Este método se está convirtiendo en el método universal para el middleware de los sistemas distribuidos. Por otro lado, REST, incluye un conjunto coordinado de restricciones en la arquitectura aplicadas a conectores, componentes y elementos de información, en un sistema hipermedia distribuido. Ignora los detalles de implementación del componente y la sintasis del protocolo para enfocarse en las responsabilidades del componente, las restricciones de sus interacciones con otros componentes y la interpretación de los elementos de información importantes. Como beneficio, una arquitectura RESTful es muy escalable y muy flexible.

Capítulo 3

Diseño

3.1. Infraestructura

3.1.1. Hardware

Los sensores necesitan un punto intermedio en el que almacenen constantemente los datos obtenidos para que posteriormente estos sean enviados a otros nodos en los que ser utilizados. Teniendo en cuenta que esta tecnología debe de ser lo más accesible posible al público, y no disponiendose de un hardware específico para esto, la mejor alternativa es utilizar las plataformas más populares de microprocesamiento: Arduino y Raspberry Pi.

3.1.1.1. Alternativas: Arduino o Raspberry Pi

Para este proyecto, hay que tener en cuenta determinados aspectos que debe de tener la placa a utilizar. Es necesario que la placa tenga un nivel de procesamiento suficiente como para procesar señales en tiempo real procedentes de sensores cardíacos o respiratorios. También es necesario que el sistema operativo de la placa permita la instalación de paquetes que permitan la comunicación con otros nodos o servicios, como ya se tratará en siguientes secciones.

Estas dos alternativas propuestas: Arduino y Raspberry Pi (Figura 3.1), son conceptos muy diferentes entre sí, cada una tiene una funcionalidad distinta. **Arduino**, por un lado, tiene mucha facilidad a la hora de conectarse con el entorno, posee entradas analógicas y digitales y un sencillo control a la hora de activar y desactivarlas, por lo que es más versátil en este campo. Para conectar esta placa a otro nodo o servicio se requiere de añadidos adicionales como Bluetooth, WiFi o Ethernet. Respecto al sistema operativo, Arduino sólo se limita a ejecutar una tarea directamente. Por otro lado **Raspberry Pi**, está diseñada como un ordenador en sí, teniendo más potencia de calculo, aunque sea menos versátil que la alternativa anterior.

Respecto a la conectividad, mientras que Arduino requería de hardware complementario para las conexiones externas, las placas Raspberry cuentan con conectividad WiFi y Ethernet integradas. Finalmente, con respecto al sistema operativo, estas placas requieren de un sistema operativo completo para ser utilizadas (Electronics Hub, 2017) [21].

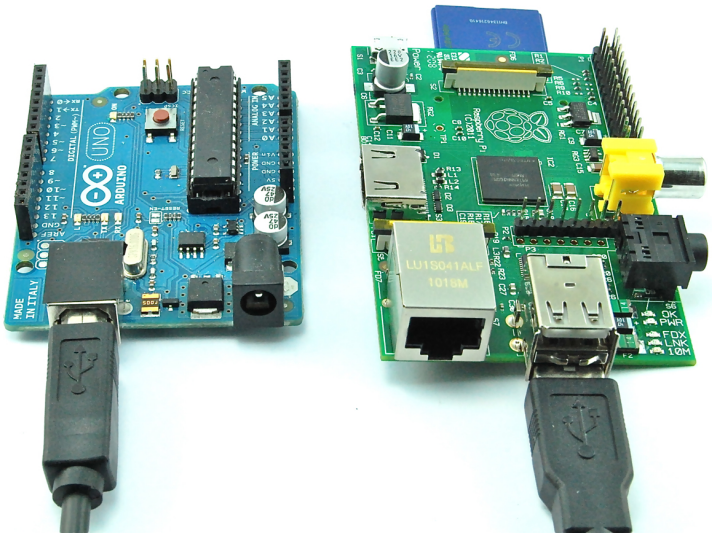


Figura 3.1: A la izquierda, una placa Arduino, y a la derecha una placa Raspberry Pi

Valorando ambas propuestas, con sus respectivas ventajas y desventajas, la alternativa más enfocada al objetivo de este proyecto es el uso de placas **Raspberry**, ya que, además de contar con el sistema operativo en el que poner en marcha todo el sistema de comunicación y contar con más potencia de computación, integra la conectividad en la misma placa, por lo que el coste es menor.

3.1.1.2. Disposición de las placas

Respecto al Hardware se utilizarían dos o más placas Raspberry que se dispondrán en forma de cluster. Conectándose entre sí a través de Kubernetes. Los sensores se dispondrán en estas placas transmisoras de datos, utilizando entre otros sensores los de la plataforma e-Health, que está preparada para este tipo de escenarios de monitoreo de la salud a través del IoT.

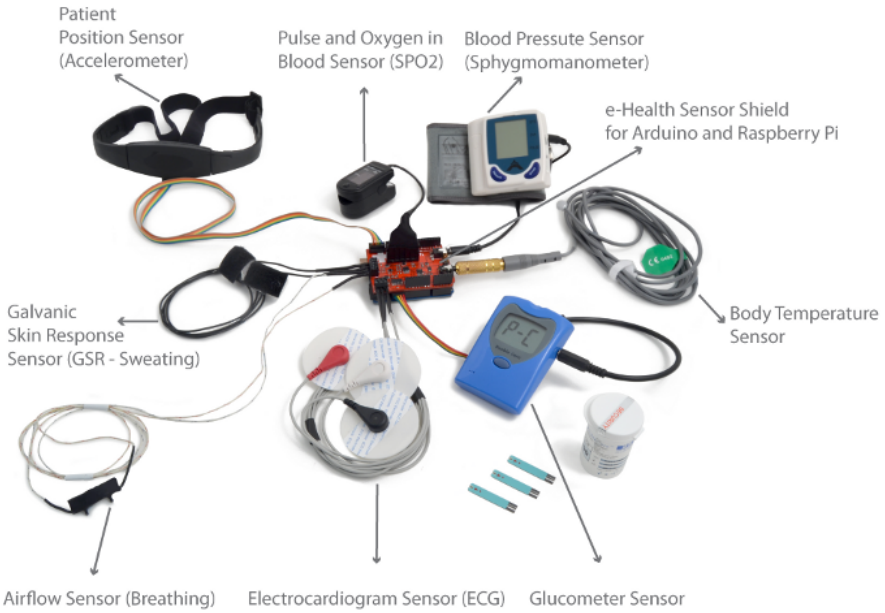


Figura 3.2: Sensores disponibles en la plataforma e-Health

3.1.2. Sensores

3.1.2.1. Plataforma e-Health

Se trata de una placa que permite tanto a las placas Raspberry Pi como Arduino efectuar aplicaciones biométricas y médicas al cuerpo del paciente a través de 9 diferentes sensores (Figura 3.2) (Coding Hacks) [23]:

- Pulso
- Oxígeno en sangre
- Flujo de aire (Respiración)
- Electrocardiograma
- Glucómetro
- Sudor
- Presión de la sangre
- Posición del paciente (Acelerómetro)

Esta placa se dispone sobre las placas Arduino o Raspberry a modo de "shield", interconectando sus entradas complementandolas con otras entradas nuevas que aporta esta plataforma (Figura 3.3). Esta información puede

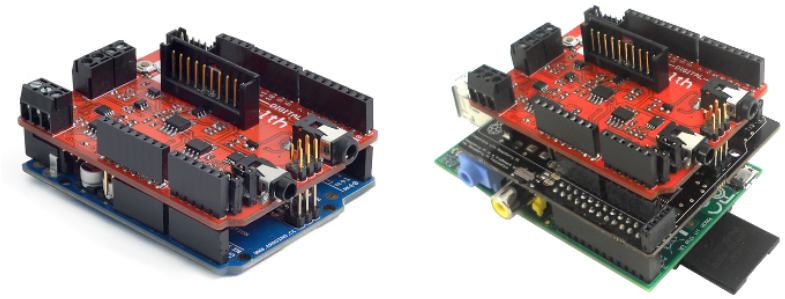


Figura 3.3: Disposición de la plataforma e-Health en una placa Arduino (izquierda) y en una Raspberry Pi (derecha)

monitorear en tiempo real el estado del paciente para después realizar un análisis médico. Además, estos datos pueden ser enviados a través de Wi-Fi, 3G, GPRS, Bluetooth, 802.15.4 y Zigbee.

3.1.3. Gestión de contenedores

Para llevar a cabo la comunicación entre los diferentes nodos que obtienen información, tanto del paciente como de su entorno, es necesario utilizar un gestor de contenedores y servicios, entre otros. Para ello se han valorado tres alternativas:

3.1.3.1. Docker Swarm

Se trata de una forma de agrupación de contenedores Docker, que viene incluido como modo en las últimas versiones de Docker [25]. Como ventajas, se pueden destacar la facilidad de puesta en marcha, ya que pueden ser ejecutados con un simple comando y Swarm se ocupará del resto; también se destaca la sencilla comunicación con otras herramientas de Docker (CLI, Compose, Krane...), además de que es el más ligero entre su competencia. Por otro lado, presenta una serie de desventajas, como por ejemplo, si un nodo muere en el clúster Swarm, los contenedores de ese nodo serán iniciados en otro nodo diferente; tampoco es tolerante a fallas, por lo que si un nodo muere en un clúster, los contenedores no serán reiniciados; además, presenta limitaciones por parte de la API de Docker: si la API de Docker no soporta algo, tampoco lo hará Docker Swarm [27].

3.1.3.2. Kubernetes

Kubernetes es un gestor de clústeres de código abierto orientado a aplicaciones basadas en contenedores a través de un número de servidores físicos o virtuales, que proveen despliegue, escalado y mantenimiento automático de

aplicaciones. Fue desarrollado por Google en 2014 con ideas y experiencias de la comunidad (Kubernetes) [24].

3.1.3.3. Valoración de las alternativas

Después de desarrollar anteriormente las tres alternativas más utilizadas en este campo, es necesario valorar sus diferentes ventajas y desventajas respecto al problema que se aborda en este trabajo. En cuanto a escalabilidad, Docker Swarm puede desplegar contenedores más rápido, permitiendo así una rápida reacción, para Kubernetes esta tarea es más lenta y compleja, pero da fuertes niveles de garantía respecto al estado del clúster y a las APIs. En nuestro caso, la velocidad de reacción no es un problema, y son preferibles las garantías que ofrece Kubernetes a la hora del escalado (PhoenixNap) [29].

En cuanto a la puesta en marcha de nodos al darse un fallo, Docker Swarm no inicia ni reinicia los nodos correctamente, por lo que si un nodo de un sensor o servicio tuviera un problema, este no sería gestionado automáticamente, por lo que en este apartado Kubernetes es la mejor opción, ya que sí que posee esta característica.

Por lo tanto, la opción elegida es **Kubernetes**, sus características y ventajas se aproximan más al problema de este proyecto que sus diferentes alternativas. Además, Kubernetes posee un gran volumen de documentación y muchos proyectos en línea que facilitarán el desarrollo de este trabajo.

3.2. Microservicios y estructura

Como ya se ha tratado en apartados anteriores, Kubernetes es la mejor opción para administrar contenedores a través de conjuntos de microservicios, ofrece muchas herramientas de gestión de contenedores (autoescalado, despliegue alternado, recursos de computación, gestión de volumenes...) [28]. Estas características encajan a la perfección con este proyecto, por ello se ha escogido esta alternativa.

3.2.1. Introducción a Kubernetes

Kubernetes se estructura en dos partes:

- **Masters**. Es el núcleo de Kubernetes, controla y planifica todas las actividades y procesos del clúster. Incluye el servidor API, el administrador de controladores, el planificador y el *etcd* (Figura 3.5).
  - **Servidor API** (*kube-apiserver*). Aporta un servidor HTTP/HTTPS, el cual tiene una API RESTful para todos los componentes en el master de Kubernetes. Por ejemplo, es posible obtener el estado de un recurso a través de una solicitud GET o crear un nuevo recurso a través de la solicitud POST.
  - **Gestor de controladores** (kube-controller-manager). Controla todo tipo de objetos en el cluster.
  - **etcd**. Kubernetes almacena todos los objetos de la API RESTful en este sitio, siendo responsable del almacenamiento y la replicación de datos.
  - **Planificador** (*kube-scheduler*). Decide que nodo es apto para ejecutar pods en él, en función de la capacidad del recurso o la utilización del nodo.
- **Nodos**. Son los trabajadores que ejecutan los contenedores (Figura 3.4). Que se compone de:
  - **Kubelet**. Es el proceso principal en los nodos, que reporta las actividades de los nodos al servidor API de kubernetes periódicamente, tales como la salud de los pods, la salud de los nodos... Ejecuta los contenedores a través de un gestor de contenedores como Docker o rkt [31].
  - **Proxy** (*kube-proxy*). Maneja las rutas entre los balanceadores de carga de los pods y los pods, además de ocuparse de las rutas desde fuera del servicio.
  - **Docker**. Es una implementación de contenedores, y es utilizado por Kubernetes como motor de contenedores por defecto.

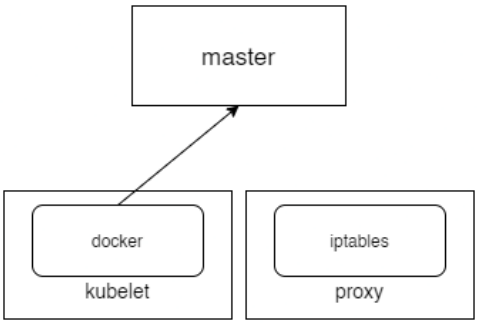


Figura 3.4: Componentes de un nodo en Kubernetes

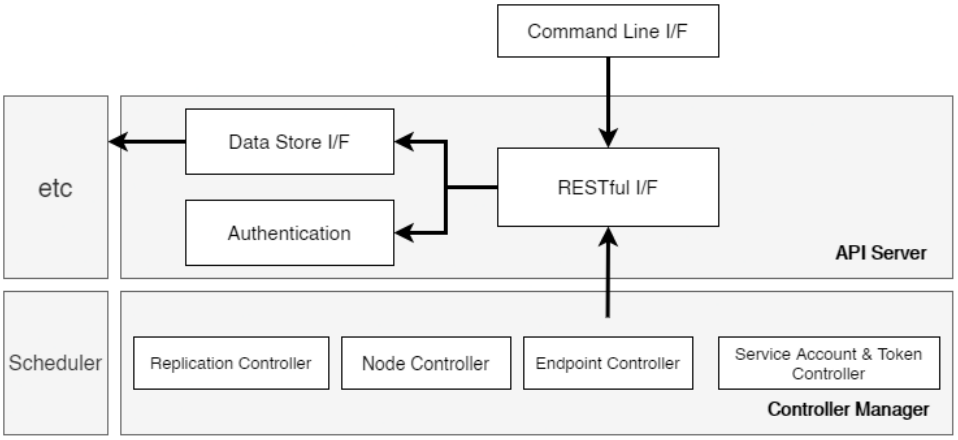


Figura 3.5: Componentes del master de componentes

3.2.2. Recursos y objetos de Kubernetes

Kubernetes contiene una gran variedad de recursos y objetos a utilizar, cada uno tiene su propósito específico en función del problema a resolver. Se enumeran a continuación los principales recursos y objetos que se utilizarán en este proyecto.

3.2.2.1. Namespaces

Los espacios de nombres de Kubernetes, en adelante denominados en inglés, *namespaces*, se utilizan para aislar los objetos que los utilicen en clusters virtuales separados. Esto es útil para tratar con conjuntos de recursos en los que cada conjunto pertenezca a un equipo o a un proyecto. En este caso se podría crear un *namespace* llamado *app* que compartirán todos los recursos utilizados en este proyecto.

3.2.2.2. Etiquetas y selectores

Las etiquetas son conjuntos de pares de claves y valores que se vinculan a los objetos de Kubernetes que se utilizan para identificar información del objeto. Por ejemplo, describir el nombre del microservicio, el rol que tiene el objeto y la versión. Se utilizan normalmente para utilizarse a través de los selectores, que seleccionan conjuntos de objetos que tengan esas mismas etiquetas.

3.2.2.3. Pods

Un pod es la unidad de Kubernetes desplegable más pequeña, que puede a su vez contener uno o más contenedores, aunque por lo general sólo es necesario uno. Están diseñados para ser mortales, es decir, pensados para ser terminados, inicializados y sustituidos unos con otros constantemente, sin alterar al conjunto de la aplicación. Kubernetes utiliza controladores para crear y administrar los pods, y estos no tienen control para terminarse o iniciarse.

3.2.2.4. Despliegues

Los despliegues son la mejor primitiva para administrar y desplegar aplicaciones. Permiten el despliegue, la actualización y el retroceso de las versiones en los pods. Cuando sea necesaria una actualización de la aplicación, estos despliegues se ocuparán de actualizar cada uno de los pods progresivamente.

3.2.2.5. Servicios

Los servicios de Kubernetes son capas de abstracción para enrutar tráfico a un conjunto lógico de pods, haciendo que con un servicio, no sea necesario trazar la dirección IP de cada pod. Los servicios utilizan selectores de etiquetas para seleccionar los pods que se necesiten. Soportan TCP y UDP. Se dividen en cuatro tipos: *ClusterIP*, *NodePort*, *LoadBalancer* y *ExternalName*.

3.2.3. Estructura general y composición

Para trasladar la tecnología microservicio y la composición de servicios a este problema a través de Kubernetes, se ha utilizado una serie de contenedores, cada uno con una función específica, que trabajaran en conjunto para medir las constantes vitales del paciente, para después ser procesadas y enviadas donde sea necesario . He utilizado una estructura simple donde se pueden diferenciar varios tipos de tipos de microservicios como generadores de información, solicitores a través de llamadas a APIs, un servicio Redis

que actúa de caché, un almacenador de esta información entre otros (Figura 3.6).

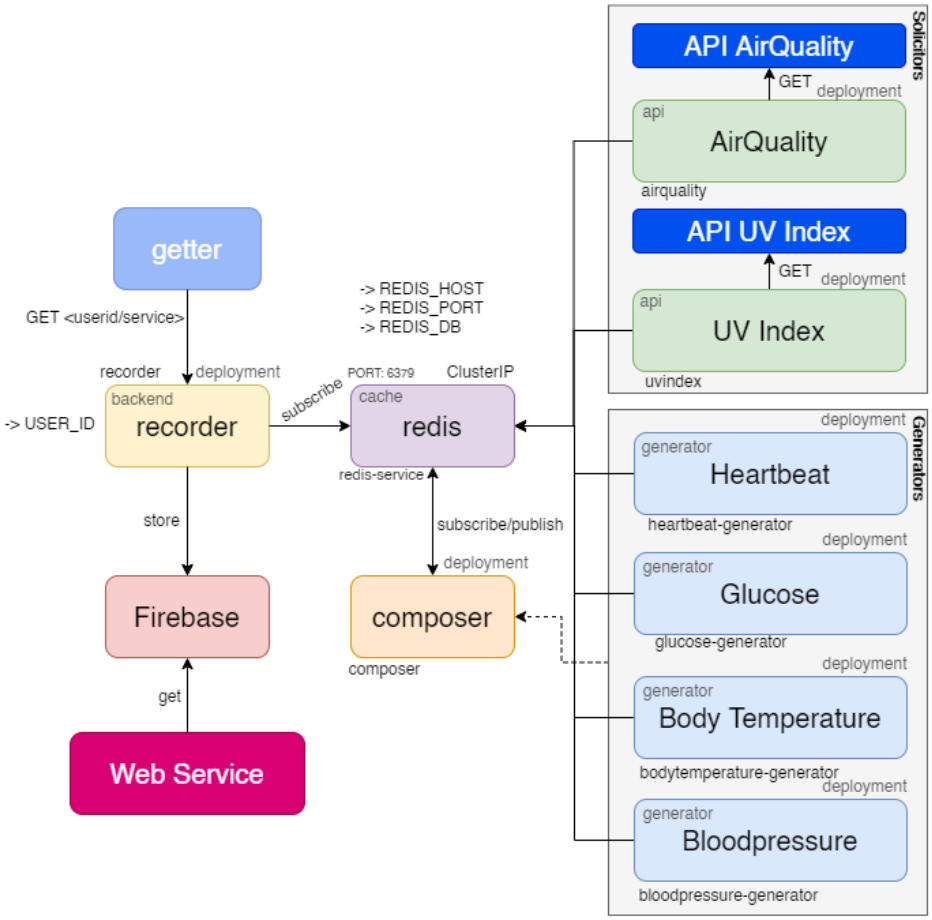


Figura 3.6: Estructura de la composición de microservicios de Kubernetes

- **Generadores**. Se trata de un conjunto de contenedores que contienen imágenes que generan variables aleatorias que simulan el estado de un paciente real.
- **Solicitantes**. Contenedores que envían solicitudes a varios servicios en la Web para conseguir una serie de datos, como por ejemplo, la calidad del aire o el índice de rayos UV. Tanto estos como los generadores solicitan o generan datos entre intervalos de tiempo.
- **Servicio Redis**. Almacen de datos en memoria que actua como caché entre los generadores de datos y el almacenador.
- **Almacenador**. Se ocupa de procesar los datos que están en la caché y almacenarlos adecuadamente en la base de datos.

3.2.4. Servicio Redis y por qué se utiliza

Redis [32] es un tipo de base de datos en memoria, que soporta estructuras de datos como conjuntos, hasshes, bitmaps... entre otros. Se ha escogido este servicio debido a la gestión de colas que puede ser útil para ese proyecto, ya que permite que el almacenamiento de datos sea muy escalable. Utilizando este servicio a través de Python [33], los generadores envian continuamente datos a un conjunto de colas preestablecidas, y estas serán procesadas en tiempo real por el servicio Redis, si estos datos son enviados y no pueden ser procesados por la base de datos, simplemente se almacenan en las colas y el almacenador las procesa en el orden conforme han sido recibidas, por lo que no hay errores. Se utiliza un servicio de suscripciones y publicaciones, el cual permite que el almacenador se suscriba a una serie de colas, y los generadores publiquen estos datos a través de estas colas (Figura 3.7).

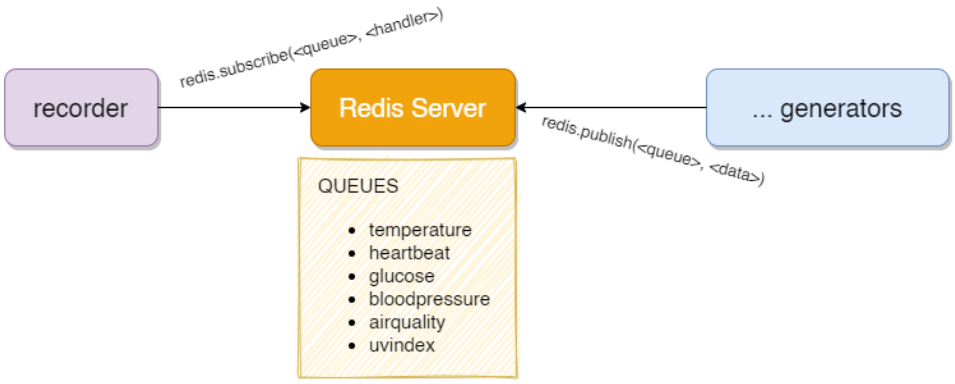


Figura 3.7: Estructura simplificada del servicio Redis

Gracias a este servicio, el sistema puede escalar el procesamiento de datos, ya que a través de estas colas, si fuera necesario en un ámbito más amplio, podrían replicarse los almacenadores encargándose y especializándose en un tipo de colas, y así cada dato se procesaría una vez en el almacenador que esté libre en primer lugar. Por otro lado, además, también es posible escalar en el número de generadores que envían datos, ya que estos generadores enviarían en función de su propósito los datos a una cola determinada y pueden coincidir enviando un tipo de datos a una misma cola, por ejemplo, dos sensores que toman la temperatura corporal del usuario.

3.2.5. Almacenador

Para procesar los datos que se envían a estas colas desde los generadores y solicitantes, es necesario un servicio adicional que escuche todas estas diferentes colas para enviarlas en orden a la base de datos correspondiente.

Este servicio, por lo tanto deberá de tener acceso, en primer lugar, al servidor **Redis**, tratado en el apartado anterior (3.2.4), a partir de la dirección del servidor, el puerto y la base de datos en la que se almacenan las colas. Además, también debe de inicializar el servicio **Cloud Firestore** [37], utilizando las credenciales de la base de datos para así conectarse y enviar la información.

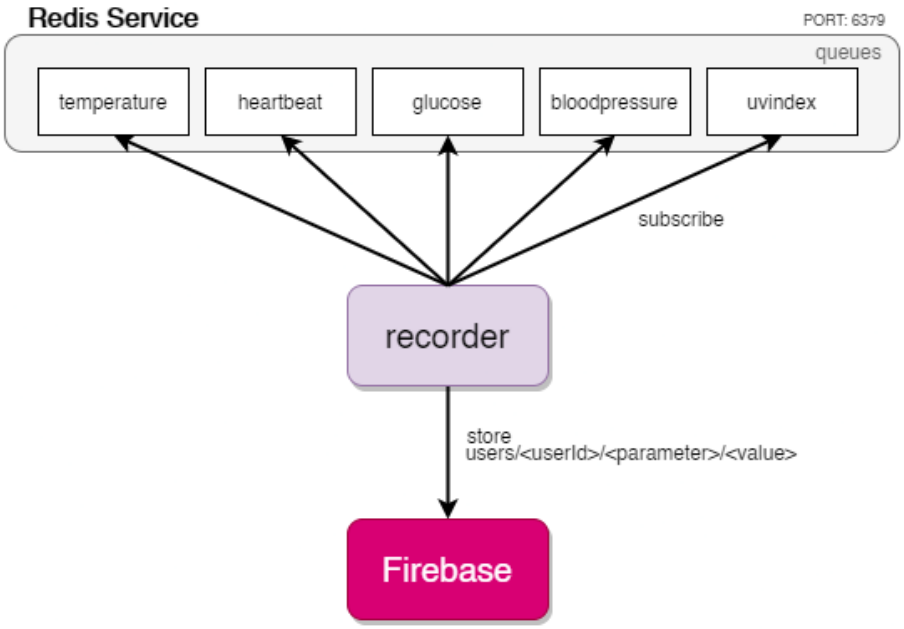


Figura 3.8: Suscripción a colas del servicio Redis por parte del almacenador

3.2.5.1. Suscripción a colas

Una vez inicializados estos dos servicios (Cloud Firestore y Redis), el almacenador se suscribirá a las colas especificadas (Figura 3.8) en base a un patrón que las identificará y se vincularán todos los datos que reciba a través de esas colas a una función que los manejará para así enviarlos a una tabla u otra en la base de datos [34]. Por ejemplo, todos los datos que provengan de colas que comiencen por la cadena de texto *temperature* se enlazarán con la función manejadora de la temperatura corporal, y enviará cada uno de estos datos a la tabla de la base de datos que almacena estas temperaturas (Código 3.1).

Código 3.1: Ejemplo de suscripción a las colas por parte del almacenador

```
1
2     def temperature_handler(message):
3     if (message['type'] == 'pmessage'):
```

```
4           SEND TO DATABASE
5
6       sub.psubscribe(**{'temperature*':temperature_handler})
7       sub.run_in_thread(sleep_time=.01)
```

Cada una de estas suscripciones se ejecutará en una hebra diferente gracias a la función de Redis `run-in-thread` y permanecerá el proceso en bucle esperando llegadas de nuevos datos hasta que se finalice el servicio.

3.2.5.2. Envío a la base de datos

Es necesario un lugar seguro, fiable y accesible donde almacenar toda la información recolectada de la caché del servicio Redis. Para este proyecto he utilizado como base de datos **Cloud Firestore**, de Firebase (Google) [37]. Esto es una base de datos NoSQL flexible, escalable y en la nube, accesible desde todo tipo de servicios, además de tener sincronización de los datos en tiempo real y una gran integración con una gran variedad de servicios, como por ejemplo, Google Cloud Platform.

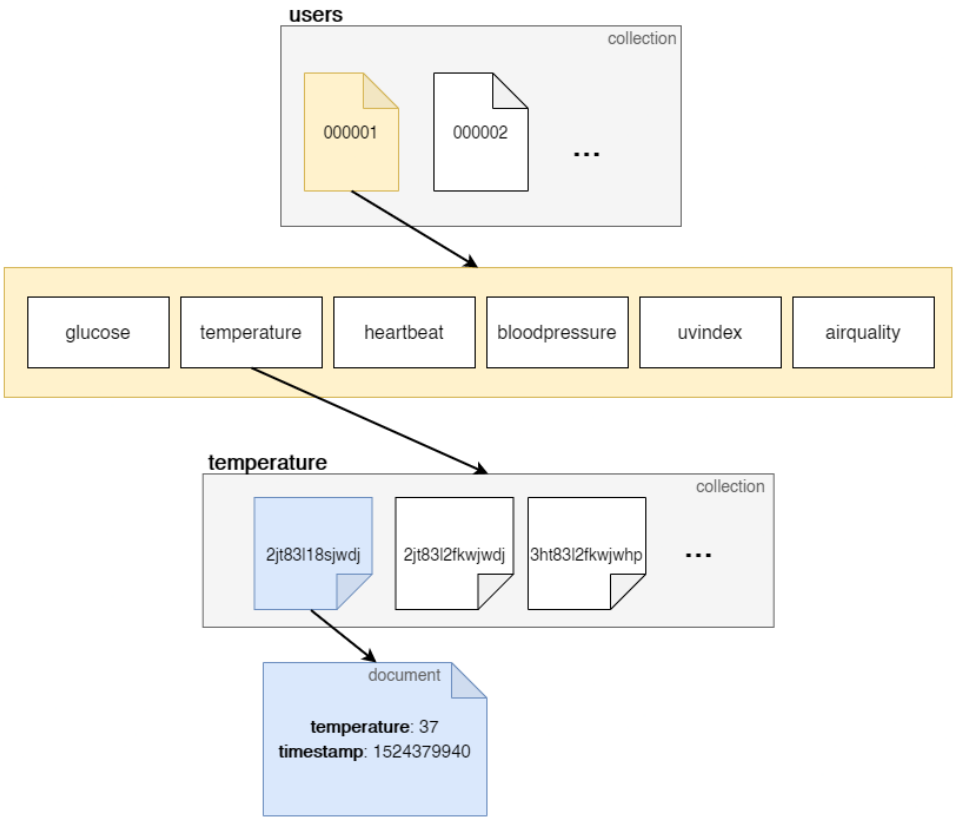


Figura 3.9: Diagrama de colecciones y documentos de la base de datos

Se ha decidido utilizar esta alternativa en lugar de otras tradicionales, como MySQL, debido a que Cloud Firestore es la opción perfecta para servicios que actualizan constantemente sus datos en tiempo real, y esto es útil a la hora de enviar y visualizar toda esta información en aplicaciones web, por ejemplo. Además, es muy escalable y flexible, admitiendo muchos tipos de estructuras de datos, pudiendo almacenar todo tipo de objetos anidados complejos en colecciones o subcolecciones.

Para utilizar esta base de datos desde este almacenador, es necesario integrar los SDK de Cloud Firestore en la imagen de este almacenador en Python, para así después pasar a inicializar el servicio, utilizando las credenciales que se facilitan desde la interfaz web de Firebase del proyecto. Después de conectar con éxito el servicio con la base de datos, se procederá a recoger cada uno de los mensajes de las colas para enviarlos a la colección correspondiente de Cloud Firestore.

Respecto a la estructura que se sigue en esta base de datos, se ha dividido toda la información en una primera colección de usuarios, cada uno de ellos identificado por una cadena de texto aleatoria, y después, dentro de cada usuario, se clasificarán los datos en subcolecciones en función de los servicios que utilice este usuario en el sistema (Código 3.2), por ejemplo temperatura corporal, glucosa, índice de rayos UV, etc .

Código 3.2: Almacenamiento de un dato del ritmo cardíaco en Cloud Firestore

```
1
2     users / 000001 / heartbeat / 03e9Bx0LTcKeKPgfdrwy
3      : {
4         heartbeat: 65,
5         timestamp: 10 de septiembre de 2020: 13:19:24 UTC+2
6     }
```

Por ejemplo, si se obtiene un ritmo cardíaco del cliente identificado como *000001*, este ritmo cardíaco se almacenará en la subcolección `000001 / heartbeat`, identificado con una cadena de texto generada aleatoriamente, y con un campo numérico que describe el ritmo cardíaco del paciente en ese momento y la marca de tiempo correspondiente.

3.2.6. Solicitantes

Hay datos que se utilizarán en el sistema que no dependen del estado del paciente, por lo que se deberán conseguir a través de solicitudes a recursos externos. Esto se consigue a través de solicitudes GET con los parámetros necesarios para especificar el lugar donde se desean conocer los datos. A través de la librería de Python *request*, es posible enviar una solicitud GET y, a partir de la respuesta, devolver un campo determinado de ella para enviarlo

a la cola determinada del servicio Redis y así almacenarla adecuadamente (Figura 3.10).

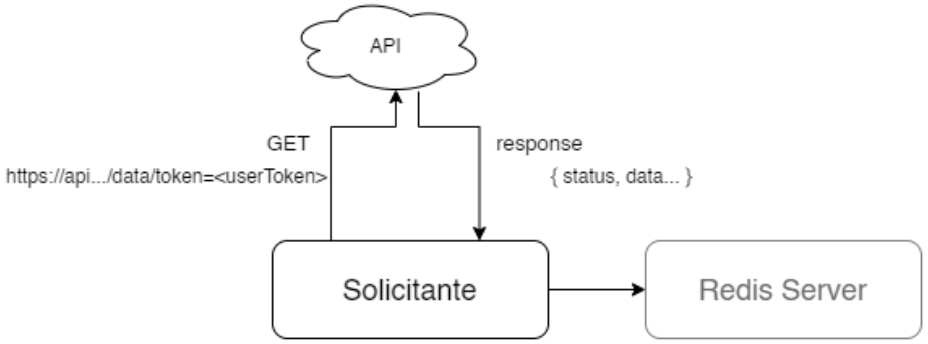


Figura 3.10: Solicitudes al servicio y respuesta

3.2.7. Generadores

Al no tener los datos reales de un paciente, debido a la complejidad que supone y a la falta de tiempo y material en este proyecto, se simularán los datos como si del estado de un paciente real se tratara, a través de servicios generadores, cada uno simulará una variable de forma que se asemeje lo máximo posible a como variaría una variable real del estado del paciente. Por ejemplo, los datos obtenidos de la temperatura de un paciente por lo general estarán en un rango entre 35 y 37º aproximadamente, por lo que no tendría sentido que el paciente tuviera de forma normal una temperatura de 42º.

Para conseguir esto, he utilizado la función de la librería *numpy* que permite escoger aleatoriamente un número de un conjunto con una serie de probabilidades asociadas a cada uno de los posibles valores (SciPy.org) [35]. Por lo que una solución rápida es generar es partir de un valor normal del rango (36º en la temperatura de un paciente, por ejemplo), y sumar o restar valores aleatorios provocando pequeñas o grandes variaciones en la temperatura del paciente ficticio, haciendo que los cambios repentinos sean más improbables y los cambios sutiles más frecuentes. Así, se puede obtener una aproximación sencilla que funciona bien en este escenario.

Para que estos valores no superen los rangos normales de una persona, se ha añadido una serie de condicionales que comprobarán si el valor actual está fuera del rango normal, y si esto ocurre, habrá más probabilidad de que se sumen o se resten valores que hagan que la variable entre en el rango de nuevo (Figura 3.11). Por ejemplo, en el caso de la temperatura corporal de un paciente, cuando se alcancen los 39 grados, se sumarán o restarán valores

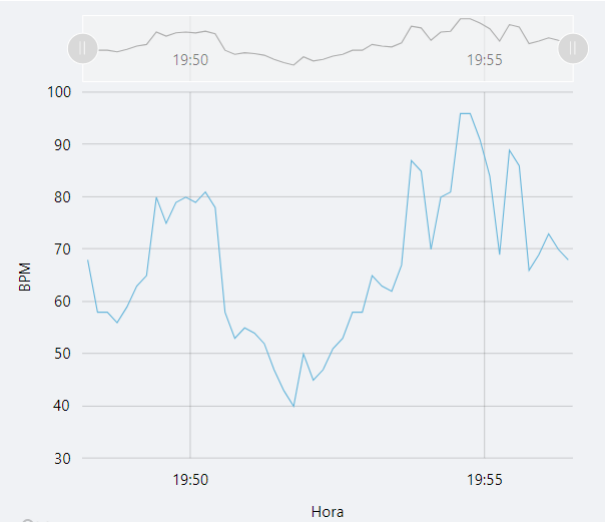


Figura 3.11: Ejemplo de generación de constantes cardíacas aleatorias

haciendo que los valores -1, -2, y -3 sean más probables que +1 o +2.

3.2.7.1. Temperatura corporal

La temperatura es uno de los indicadores más importantes en el estado de la salud de una persona. A través de esta medida se puede conocer rápidamente cuando un paciente necesita atención urgente. Este servicio generará temperaturas en los rangos descritos a continuación [38].

- Rango normal (35º - 37,5º)
- Hipotermia: alerta (Menos de 35º)
- Fiebre: alerta (Más de 37,5º)

3.2.7.2. Glucosa

Conocer el nivel de azúcar en la sangre es especialmente necesario en personas con diabetes o con ries (Healthline) [39]

- Rango normal (70 - 120 mg/dL) (Antes de comer)
- Hiperglucemia: alerta y notificación para tomar insulina (Más de 120 mg/dL)
- Hipoglucemia: alerta (Menos de 70 mg/dL)

3.2.7.3. Pulso cardíaco

También conocida como la frecuencia cardíaca. Este dependerá fundamentalmente de la actividad física del paciente, por lo que para tomar estas medidas se supondrá que el paciente se encuentra en reposo y no después de realizar una actividad física importante [40].

- Rango normal [60 - 100 bpm].
- Baja tensión: alerta (Menos de 60 bpm)

3.2.7.4. Respiraciones por minuto

Se trata del número de respiraciones que una persona hace en un minuto. Estas se miden en completo reposo.

- Rango normal (12 - 20 respiraciones por minuto)
- Bradipnea: alerta (Menos de 10 respiraciones por minuto)
- Taquipnea: alerta (Más de 20 respiraciones por minuto)

3.2.7.5. Calidad del aire

La calidad del aire en un lugar en concreto se define como la concentración de contaminante que contiene el aire. A medida que este valor es mayor, indica que el aire presenta mayor cantidad de contaminantes, por lo que esto llega a ser un gran riesgo para la salud pública. Este microservicio trabajará con la ubicación del paciente y, a través de la API proporcionada por *Air Quality Programmatic APIs* [42], podrá obtener el valor numérico de esta contaminación y así alertar al paciente en el caso que exista riesgo para su salud al salir al exterior.

Se llamará a este servidor con el método *GET* adjuntando la ubicación de la vivienda en forma de latitud y longitud, con la siguiente estructura:

```
1    /feed/geo::lat;:lng/?token=:token
```

Donde *lat* es la latidud, *lng* es la longitud, y *token* es el token obtenido en la plataforma de la API para realizar las llamadas. El servidor responderá, en caso de éxito, con *status*: ok.

- Buena calidad (0 - 49)
- Calidad moderada (50 - 99)
- Insalubre para grupos de riesgo: alerta (100 - 149)
- Insalubre: alerta (Más de 150)

3.2.7.6. Índice Ultravioleta

Este índice permite conocer el impacto que ocasiona el sol en la piel en un lugar y momento determinado. Para medir este índice, se utilizará la escala del índice UV establecida por la Organización Mundial de la Salud [43], que establece, por ejemplo, que con un índice de 8 a 9 una persona con piel sensible puede quemarse en menos de 15 minutos sin protección alguna. Este valor se obtendrá de la *Weather API*, que permite, a través de parámetros como la ciudad o la latidud y longitud de un lugar, conocer su índice UV [44].

```
1    http://api.weatherbit.io/v2.0/current&:lat&:lon&:key
```

Donde *lat* es la latidud, *lng* es la longitud, y *key* es el token obtenido en la plataforma de la API para realizar las llamadas. El servidor responderá, en caso de éxito, con un campo *data* con los datos a partir de los cuales se extraerá el índice UV en su parámetro *uv*. A partir del cual, se determinará el peligro del paciente en función del rango en el que se encuentra este valor [45]

- Rango normal (0 - 2)
- Rango moderado (3 - 5)
- Rango alto: alerta (6 - 7)
- Rango muy alto: alerta (8 - 10)
- Rango extremo: alerta (Más de 11)

3.2.8. Compositor

Los generadores, además de enviar los datos al almacenador para que este los almacene en la base de datos, también los enviarán al servicio compositor, que como su nombre describe, los comparará entre sí y determinará en función de unos rangos especificados, cuál es el nivel de salud en el que se encuentra el paciente. Este servicio actuará como cliente de los servicios generadores escuchando a las colas del servicio Redis para determinar el estado general en el que se encuentra la persona con el objetivo de enviar este estado en forma de porcentaje al servicio almacenador por medio del servicio Redis.

3.2.8.1. Escucha de los generadores

Al igual que en el almacenador descrito previamente (3.2.5), este servicio compositor también escuchará las colas de los servicios generadores,

almacenará sus valores y una vez tenga el estado actual de todos serán comparados. Se suscribirá a todas las colas y una vez se reciban los mensajes los filtrará para así determinar cuáles de ellos proceden de los generadores, y serán filtrados una vez más para almacenar los valores de cada sensor en un array de enteros.

3.2.8.2. Nivel de salud del paciente

Para comparar constantemente los valores de todos los sensores, en las primeras iteraciones se almacena un valor nulo, y sólo cuando se tienen los valores de cada uno de los generadores, se comparan entre sí y se comprueba el rango en el que se encuentra cada uno de los parámetros. Inicializando el valor del nivel de salud del paciente a 100, se irá decrementando este porcentaje por cada parámetro de los sensores que esté fuera del rango normal de la salud de una persona adulta, tal y como se describe anteriormente en el apartado de los generadores (3.2.7).

3.2.9. Los despliegues y la continua actualización

Los despliegues de Kubernetes, como se ha descrito anteriormente, permiten especificar cómo exactamente deben actualizarse o inicializarse los pods vincultados a estos. En esta composición, mayoritariamente se han utilizado despliegues con una estrategia de continua actualización, es decir, actualizar la imagen de un contenedor en un pod en particular sin perder la disponibilidad del conjunto; esta estrategia se denomina *RollingUpdate* [28].

Para conseguir esto, hay que especificar una serie de parámetros para indicar cómo los pods reaccionarán ante estos cambios.

- **Tiempo mínimo en el que el pod se inicializa**: *minReadySeconds*. El tiempo que necesita como mínimo el pod para estar disponible. Se ha estimado que son necesarios unos 4 segundos.
- **Número de pods no disponibles permitidos**. Cuantos pods como máximo podrán estar no disponibles de fondo. Se establece uno.
- **Número de pods inicializándose permitidos**. Pods permitidos estar de fondo iniciándose.

En la práctica, por ejemplo, en el caso en el que haya que actualizar el servicio que toma (o genera) la temperatura del paciente debido a que este no está bien calibrado, hay que especificar la versión nueva de la imagen actualizada de este servicio, por ejemplo *temperature-sensor:2.1*. Kubernetes inicializará en segundo plano la versión nueva del pod mientras mantiene disponible la anterior, y cuando el tiempo mínimo en el que el pod está

disponible (*minReadySeconds*) haya transcurrido, el pod que contenía la versión anterior no deseada se terminará, y su lugar será reemplazado por la nueva versión a través de otro pod nuevo.

3.3. Servicio Web

Tras conseguir el objetivo de obtener, procesar y almacenar de forma ordenada en la base de datos toda la información del paciente, es necesario desarrollar una aplicación que pueda representar gráficamente toda la información recopilada, así como mostrar sus variaciones a lo largo del tiempo y poder determinar el estado del paciente en el pasado y en tiempo real, además de alertar cuando se ha captado algún valor que esté fuera del rango de normalidad para algún parámetro de los que se obtengan de un determinado usuario (Figura 3.12).

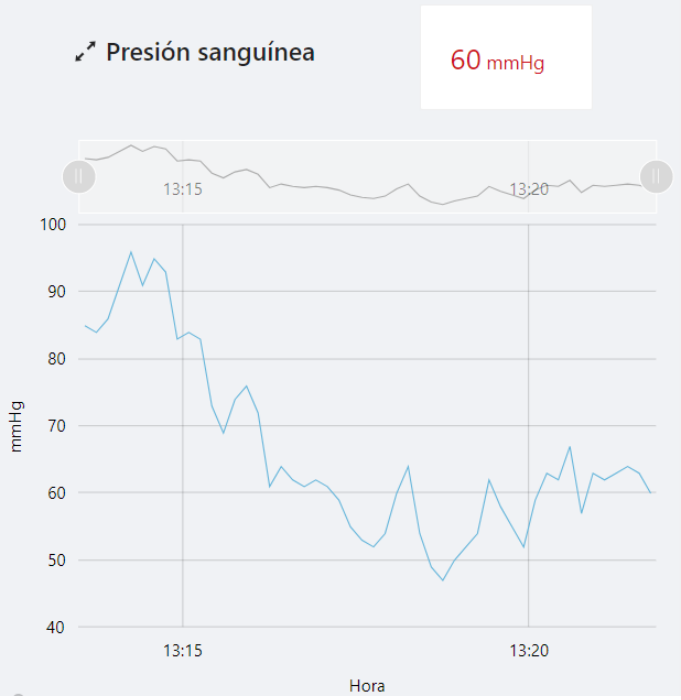


Figura 3.12: Ejemplo de representación gráfica del histórico de la presión sanguínea

Para ello, se ha utilizado el framework de Javascript **React** [46], que es una de las bibliotecas más utilizadas y útiles a la hora de desarrollar interfaces de usuario simples. React actualiza y renderiza de forma de forma eficiente los componentes cuando hay cambios en sus datos correspondientes. Así, en este trabajo, se actualizarán cada uno de los componentes (gráficas de temperatura corporal, constantes cardíacas...) de forma independiente y sin modificar el estado de los componentes en los que no se produzcan cambios.

Además de React, se ha desarrollado la interfaz con la librería de diseño **Ant Design** [47], que es muy útil para crear interfaces fáciles y simples.

3.3.1. Representación gráfica

Uno de los objetivos que tiene esta aplicación web es el de representar el histórico de valores que tiene y ha tenido el paciente, para así permitir posibles diagnósticos que se hagan. Para esto, existe una infinidad de librerías que permiten la inserción de datos y su correspondiente representación para así integrarla en una web. He utilizado una conocida librería de representación de gráficas, **AmCharts** [48] (Figura 3.16).

Aprovechando esta librería, a partir de los datos, que se almacenan en forma de pares de valor y marca de tiempo correspondiente, se representarán en el eje X los momentos en los que se ha llevado a cabo la medida de los datos y en el eje Y, el valor de cada uno de estos datos. Además, se permite ampliar una zona concreta de la gráfica o ver qué valor exacto hay en cada uno de los puntos con el puntero del ratón.

3.3.2. Componentes

Aprovechando el potencial de React, es trivial dividir en este escenario cada una de las diferentes métricas que se obtienen en componentes distintos, permitiendo que todos estos componentes, al ser modificados, no afecten al renderizado del resto (Figura 3.13). Así, cada componente accederá independientemente a la base de datos Cloud Firestore, para obtener los datos en tiempo real que le corresponden, y actualizará con ellos su estado individual. Por ejemplo, en el caso de la temperatura del paciente, el componente *Temperature* descargará los datos del servidor y además serán actualizados si se reciben nuevos; estos datos se reciben en forma de vector de temperaturas e inicializará el gráfico correspondiente.

3.3.3. Actualización en tiempo real

La alteración de los estados de estos componentes en tiempo real es una característica que debía de tener esta aplicación, debido a que se desea conocer el estado anterior del paciente., pero también el actual, sin necesidad de actualizar la página continuamente, con el fin de saber cómo avanza el paciente. Esto se consigue gracias a la ventaja del almacenamiento Firestore de Firebase, que permite escuchar los eventos que alteren una colección en la base de datos, pudiendo conocer si se trata de un nuevo documento, una modificación de uno ya existente, o la eliminación de uno de ellos [49].

Aprovechando esta característica, el estado de cada uno de los componentes será alterado en tiempo real en función de los cambios que ocurran en la base de datos, obteniendo así las métricas del paciente en directo, y en consecuencia, actualizándose cada una de las gráficas.

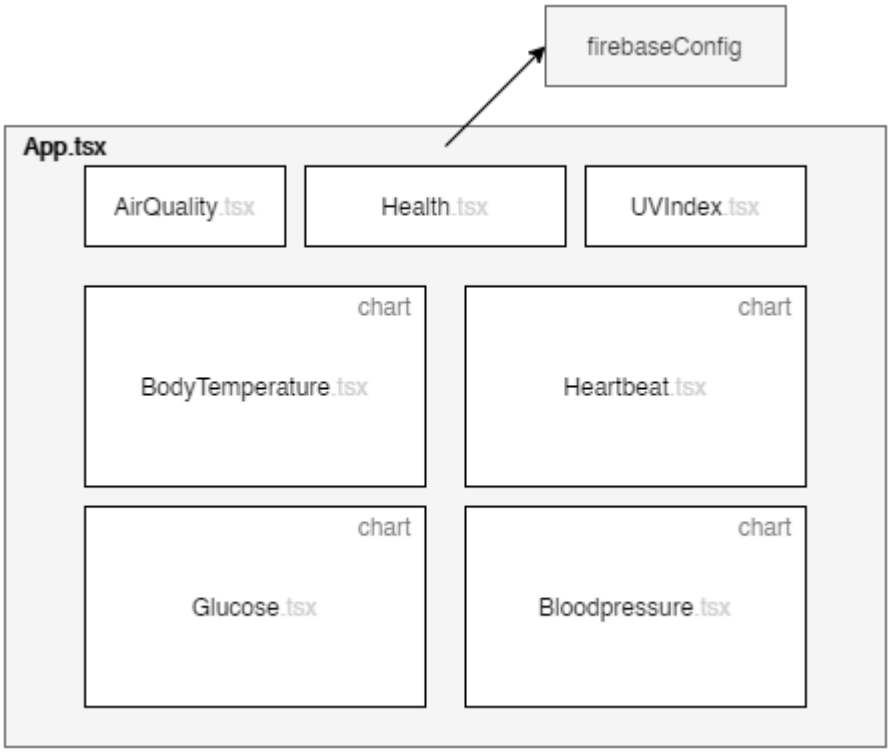


Figura 3.13: Diagrama de componentes React que indica cómo se presentarán en la interfaz

3.3.4. Alertas y valores fuera del rango normal

Para hacer más util esta aplicación, teniendo en cuenta los rangos de parámetros que se consideran adecuados para una persona, se mostrará gráficamente en tiempo real cuando se está obteniendo un parámetro fuera de estos rangos. En el momento que un valor sobrepasa el rango que le corresponde, este aparecerá en rojo y con fuente resaltada.

3.3.5. Valores sin gráfica

En el caso de los parámetros que son más estáticos que el resto, en su representación no se requiere de una gráfica que muestre el progreso, ya que este no aporta más información, debido a que se trata de métricas independientes al paciente: la calidad del aire exterior, el índice de rayos UV (Figura 3.15) y el porcentaje de salud del paciente (Figura 3.14).

Estas métricas se engloban cada una independendientemente como componentes, y sus valores también se obtienen en tiempo real, pero no se muestra una gráfica como para el resto de componentes. Sin embargo, sí se muestra cuando esta métrica que se ha obtenido es mayor o menor que

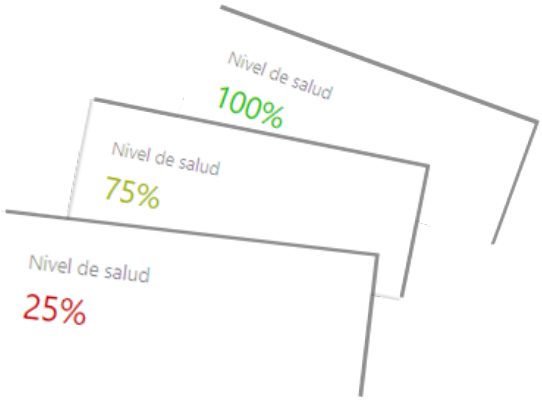


Figura 3.14: Porcentaje de salud del paciente, en diferentes niveles

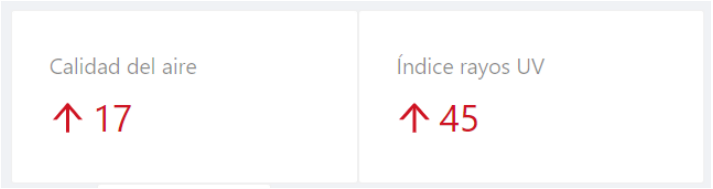


Figura 3.15: Valores sin gráfica; aparecen con un color rojo cuando el último cambio supone un empeoramiento

la anterior almacenada, notificando así de forma rápida si esta está disminuyendo o aumentando con colores verdes o rojos, dependiendo de lo que esto suponga en la salud del paciente. Por ejemplo, en el caso de la contaminación del aire, si esta está obteniendo valores mayores a los que se han obtenido previamente, se muestra en el panel con un color rojo y una flecha indicando que ha aumentado desde el valor anterior.

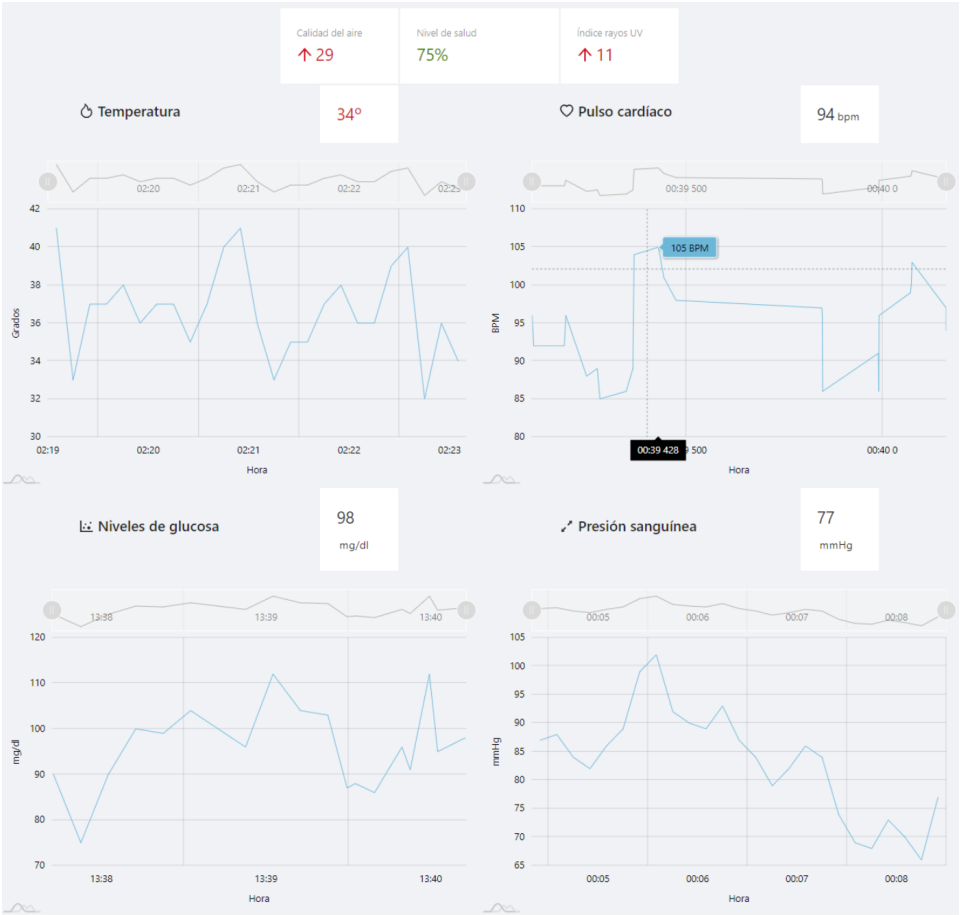


Figura 3.16: Captura de pantalla de la interfaz del servicio web

Capítulo 4

Conclusiones y Trabajos Futuros

4.1. Conclusiones

Las principales dificultades de este trabajo han sido lograr una adecuada comunicación entre microservicios, consiguiendo que fuera lo más escalable posible, logrando que cada sensor o servicio pueda acoplarse o desacoplarse sin que esto afecte al sistema en conjunto. La tecnología microservicio es un paradigma de computación muy diferente al resto, por lo que, al comienzo del desarrollo de este trabajo, el esfuerzo se ha enfocado en conseguir trasladar una idea sencilla en otros paradigmas, a este escenario, a partir de la tecnología microservicio de Kubernetes.

4.1.1. Infraestructura

Kubernetes ha jugado un aspecto clave en este proyecto, ya que ha sido la base de toda esta composición de microservicios. Esta plataforma, tal y como se analizó durante la etapa de planificación este trabajo (3.1.3.3), ha llevado a cabo su función sin problemas y, gracias a su extensa y accesible documentación, no se ha tardado mucho tiempo a la hora de aprender a utilizar esta plataforma.

4.1.2. Microservicios

Una preocupación que se tuvo durante la planificación y parte del desarrollo de este sistema fue la de lograr una buena composición de microservicios que cumpliese con el objetivo y a la vez que pudiera cumplir la mayoría de condiciones necesarias para que cada uno de estos microsericios fuesen considerados como tal (2.1.1). Finalmente, esto se consigió gracias a la utilización de Redis como servicio caché a la hora de almacenar los datos antes de ser procesados para enviarlos a la base de datos. Además gracias

a la utilización de Cloud Firestore como base de datos donde almacenar y estructurar toda la información final preparada para ser utilizada por los servicios que la necesiten.

4.1.3. Servicio web

Esta parte del sistema fue desarrollada en menor tiempo que el resto ya que no hizo falta el mismo tiempo a la hora de aprender una nueva tecnología como son los microservicios y su composición, debido a que previamente había utilizado tecnologías como React para desarrollar servicios web.

Además, es de destacar la facilidad de representar los datos gracias a que estos se almacenan en una base de datos preparada para actualizaciones en tiempo real (Cloud Firestore), por lo que se ha conseguido un servicio web simple, modularizado y preparado para ser consultado en tiempo real.

4.2. Trabajos futuros

Como trabajo futuro, existe un gran abanico de posibilidades para un proyecto así. Esta es una propuesta que demuestra cómo componer un conjunto de microservicios para trabajar entre sí con parámetros de la salud de una persona, pero no habiendo sido posible la extracción de estos por medio de una infraestructura física, esto es un aspecto que queda por resolver adecuadamente para conseguir una propuesta completa de un sistema de monitorización.

También, en este proyecto se ha trabajado con un número reducido de magnitudes captadas del paciente, esto es un ejemplo, existe un gran número de parámetros que pueden también ser obtenidos a partir de sensores y similares. Podrían añadirse más gracias a las características de este sistema propuesto, ya que este se ha desarrollado teniendo en cuenta la facilidad a la hora de añadir nuevos puntos de extracción de datos. Por ejemplo, podría añadirse un nuevo microservicio que tome la tensión del paciente y se agrege a todo este sistema de forma sencilla.

Este sistema, además de poder funcionar en el ámbito de la salud, también es posible trasladarlo a otras áreas diferentes. Por ejemplo, en el caso de querer trasladar este sistema a la gestión de un almacén de alimentos, se necesita monitorear una serie de parámetros como son la caducidad de alimentos, los pedidos automáticos, gestión de cámaras frigoríficas junto a su temperatura... Estos parámetros serían utilizados por la composición de microservicios para así ser utilizados para la toma de decisiones a través del servicio web. Todo esto sería posible llevando a cabo un proceso simple de adaptación a este nuevo entorno.

Por otro lado, es necesaria una revisión de la seguridad de todo este sistema, consiguiendo que diferentes pacientes utilicen el mismo sistema sin interferir un paciente en el sistema del otro.

Bibliografía

[1] Nadareishvili, I., Mitra, R., McLarty, M., & Amundsen, M. (2016). Microservice architecture: aligning principles, practices, and culture. Reilly Media, Inc.".

[2] Butzin, B., Golatowski, F., & Timmermann, D. (2016, September). Microservices approach for the internet of things. In 2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA) (pp. 1-6). IEEE.

[3] Dragoni, N., Lanese, I., Larsen, S. T., Mazzara, M., Mustafin, R., & Safina, L. (2017, June). Microservices: How to make your application scale. In International Andrei Ershov Memorial Conference on Perspectives of System Informatics (pp. 95-104). Springer, Cham.

[4] Circuit breaker pattern. Microsoft Docs. https://msdn.microsoft.com/de-de/library/dn589784.aspx

[5] Di Francesco, P. (2017, April). Architecting microservices. In 2017 IEEE International Conference on Software Architecture Workshops (ICSAW) (pp. 224-229). IEEE.

[6] Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., et al. & Zaharia, M. (2010). A view of cloud computing. Communications of the ACM, 53(4), 50-58.

[7] Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C. C., Khandelwal, A., Pu, Q., et al. & Gonzalez, J. E. (2019). Cloud programming simplified: A berkeley view on serverless computing. arXiv preprint arXiv:1902.03383.

[8] Marinescu, D. C. (2017). Cloud computing: theory and practice. Morgan Kaufmann.

[9] Mell, P., & Grance, T. (2011). The NIST definition of cloud computing.

[10] Attaran, M., & Woods, J. (2019). Cloud computing technology: improving small business performance using the Internet. Journal of Small Business & Entrepreneurship, 31(6), 495-519.

[11] Darwish, A., Hassanien, A. E., Elhoseny, M., Sangaiah, A. K., & Muhammad, K. (2019). The impact of the hybrid platform of internet of things and cloud computing on healthcare systems: opportunities, challenges, and open problems. Journal of Ambient Intelligence and Humanized Computing, 10(10), 4151-4166.

[12] OpenIoT. http://www.openiot.eu/

[13] Sensei Project Solutions. https://www.senseiprojectsolutions.com/solutions/ppmbeacon/

[14] Paik, H. Y., Lemos, A. L., Barukh, M. C., Benatallah, B., & Natarajan, A. (2017). Web Service Implementation and Composition Techniques (Vol. 256). Springer International Publishing.

[15] Bansal, S., Bansal, A., Gupta, G., & Blake, M. B. (2016). Generalized semantic Web service composition. Service Oriented Computing and Applications, 10(2), 111-133.

[16] Milanovic, N., & Malek, M. (2004). Current solutions for web service composition. IEEE Internet Computing, 8(6), 51-59.

[17] Limthanmaphon, B., & Zhang, Y. (2003, January). Web service composition with case-based reasoning. In Proceedings of the 14th Australasian database conference-Volume 17 (pp. 201-208).

[18] Asghari, P., Rahmani, A. M., & Javadi, H. H. S. (2018). Service composition approaches in IoT: A systematic review. Journal of Network and Computer Applications, 120, 61-77.

[19] F. Khodadadi, A. V. Dastjerdi, and R. Buyya, "Simurgh: A framework for effective discovery, programming, and integration of services exposed in IoT," in Proc. Int. Conf. Recent Adv. Internet Things (RIoT), 2015, pp. 1–6.

[20] Hamzei, M., & Navimipour, N. J. (2018). Toward efficient service composition techniques in the internet of things. IEEE Internet of Things Journal, 5(5), 3774-3787.

[21] (2017). What are the differences between Raspberry Pi and Arduino? Electronics Hub. https://www.electronicshub.org/raspberry-pi-vs-arduino/

[22] Kumar, R., & Rajasekaran, M. P. (2016, January). An IoT based patient monitoring system using raspberry Pi. In 2016 International Conference on Computing Technologies and Intelligent Data Engineering (ICCTIDE'16) (pp. 1-4). IEEE.

[23] e-Health Sensor Shield V2.0 for Arduino, Raspberry Pi and Intel Galileo [Biometric / Medical Applications].
Coding Hacks. https://www.cooking-hacks.com/ehealth-sensor-shield-biometric-medical-arduino-raspberry-pi.html

[24] Kubernetes. ¿Qué es Kubernetes?
https://kubernetes.io/es/docs/concepts/overview/what-is-kubernetes/

[25] Swarm mode overview. Docker Docs.
https://docs.docker.com/engine/swarm/

[26] Nomad. https://www.nomadproject.io/

[27] What is the best alternative to Kubernetes? Slant. https://www.slant.co/options/11649/alternatives/ kubernetes-alternatives

[28] Saito, H., Lee, H. C. C., & Wu, C. Y. (2019). DevOps with Kubernetes: Accelerating software delivery with container orchestrators. Packt Publishing Ltd.

[29] Kubernetes vs Docker Swarm: What are the Differences? PhoenixNAP.
https://phoenixnap.com/blog/kubernetes-vs-docker-swarm

[30] Service. Kubernetes.
https://kubernetes.io/docs/concepts/services-networking/service/

[31] rkt. A security-minded, standards-based container engine. CoreOS.
https://coreos.com/rkt/

[32] Redis. https://redis.io/

[33] Python client for Redis key-value store. Pypi.
https://pypi.org/project/redis/

[34] Basic Redis Usage Example. Part 1: Exploring PUB-SUB with Redis & Python
https://kb.objectrocket.com/redis/basic-redis-usage-example-part-1-exploring-pub-sub-with-redis-python-583

[35] numpy.random.choice. https://docs.scipy.org/doc//numpy-1.10.4/reference/generated/numpy.random.choice.html

[36] Firebase. https://firebase.google.com/

[37] Cloud Firestore. Firebase.
https://firebase.google.com/docs/firestore?hl=es-419

[38] Body temperature norms. MedlinePlus.
https://medlineplus.gov/ency/article/001982.htm

[39] ¿Qué es la glucosa?. Healthline.
https://www.healthline.com/health/es/glucosa

[40] What should my heart rate be? MedicalNewsToday.
https://www.medicalnewstoday.com/articles/235710

[41] Vital signs. Cleveland Clinic.
https://my.clevelandclinic.org/health/articles/10881-vital-signs

[42] API - Air Quality Programmatic APIs. https://aqicn.org/api/es/

[43] UV Index Scale- United States Environmental Protection Agency EPA
https://www.epa.gov/sunsafety/uv-index-scale-0

[44] Weather Api Documentation. https://www.weatherbit.io/api

[45] Ultraviolet Index. Wikipedia.
https://en.wikipedia.org/wiki/Ultraviolet_index

[46] ReactJS. https://es.reactjs.org/

[47] Ant Design. https://ant.design/

[48] JavaScript Charts & Maps. AmCharts. https://www.amcharts.com/

[49] Get realtime updates with Cloud Firestore. Firebase.
https://firebase.google.com/docs/firestore/query-data/listen

[50] Instalar y configurar kubectl. Kubernetes.
https://kubernetes.io/es/docs/tasks/tools/install-kubectl/

[51] Hello world. Express.
https://expressjs.com/es/starter/hello-world.html

[52] David Jöch. Functional vs Class-Componentes in React. Medium. 2018.
https://medium.com/@Zwenza/functional-vs-class-components-in-react-231e3fbd7108

Apéndice A

Configurando el servidor en Kubernetes

A.1. Configurando los servidores

Se deben de configura Kubernetes en las placas Raspberry, y para ello se utilizan los contenedores de Docker, utilizando la versión 20.04 de Ubuntu con la versión de Docker v19.

```
$ sudo apt intall -y docker.io
```

Después de instalarlo se deben de regular los grupos de control (Control Groups) para limitar y tener un mayor control de los recursos. Para ello se cambia el driver por defecto de Docker de "cgroups" a "system" para permitir a "systemd" actuar como el administrador de cgroups. Esto es recomendado por Kubernetes; para ello se reemplaza el archivo "*/etc/docker/daemon.json*" con:

```
$ sudo cat > /etc/docker/daemon.json <<EOF
{
"exec-opts": ["native.cgroupdriver=systemd"],
"log-driver": "json-file",
"log-opts": {
"max-size": "100m"
},
"storage-driver": "overlay2"
}
EO
```

Para activar los límites de soporte de cgroup es necesario modificar la línea de comandos del kernel para activar estas opciones al inicio. Se debe de añadir lo siguiente al archivo "/boot/firmware/cmdline.txt".

```
cgroup_enable=cpuset
cgroup_enable=memory
cgroup_memory=1
swapaccount=1
```

Después de llevar a cabo estos cambios, Docker y el kernel deben de estar configurados adecuadamente para Kubernetes; se reiniciarán las Raspberries y deberían de estar los drivers modificados.

A.2. Permitiendo a iptables ver el tráfico

Para el correcto funcionamiento de Kubernetes se necesita configurar iptables para ver el tráfico de la red. Esto se hace cambiando la configuración "sysctl".

```
cat <<EOF | sudo tee /etc/sysctl.d/k8s.conf
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
EOF
$ sudo sysctl --system
```

A.3. Instalando los paquetes Kubernetes en Ubuntu

Se añaden los repositorios de Kubernetes a los códigos de ubuntu, y una vez añadidos se añaden tres paquetes requeridos por Kubernetes: *kubelet, kubeadm* y *kubectl*.

```
$ curl -s https://packages.cloud.google.com/apt/doc/apt-key
    .gpg | sudo
apt-key add -
```

```
$ cat <<EOF | sudo tee /etc/apt/sources.list.d/kubernetes.
    list
deb https://apt.kubernetes.io/ kubernetes-xenial main
```

Una vez añadido el repositorio se añaden los tres paquetes requeridos por Kubernetes: kubelet, kubeadm y kubectl.

```
$ sudo apt update && sudo apt install -y kubelet kubeadm
    kubectl
```

Finalmente se deben de desactivar las actualizaciones regulares de estos paquetes, ya que podrían desestabilizar todo el sistema y para actualizar estos paquetes antes se deberían de hacer modificaciones manuales.

```
1    $ sudo apt-mark hold kubelet kubeadm kubectl
```

A.4. Creando el cluster de Kubernetes

Después de instalar los paquetes de Kubernetes, se puede proceder a crear un cluster. Para ello se debe de seleccionar una de las Raspberries como panel de control (o nodo primario), y el resto serán nodos de computación en los que se encontrarán los diversos sensores que se utilizarán en el trabajo.

También se debe de establecer un Classless Inter-Domain Routing (CIDR), para usar los pods en el cluster de Kubernetes.

A.5. Iniciando el nodo principal

Para unir nodos al clúster, Kubernetes utiliza un token para autenticarlos. Este token necesita ser pasado al comando "kubeadm init" cuando se inicialice este nodo principal. Para generar el token se utiliza el comando "kubeadm token generate".

```
1    $ TOKEN=$(sudo kubeadm token generate)
```

Después de esto, ya se puede inicializar el nodo, usando el comando "kubeadm init".

```
1    $ sudo kubeadm init --token=${TOKEN} --kubernetes-version=
         v1.18.2
2    --pod-network-cidr=10.244.0.0/16
```

Si no ha habido ningún problema, se podrá obtener del mensaje que aparecerá a continuación la información de conexión. Se copiará el contenido del archivo resultante "/etc/kubernetes/admin.conf" al archivo " /.kube/config", esto permitirá controlar el cluster con el comando "kubectl".

Después, también en el mensaje resultante del comando anterior podemos extraer el comando "kubernetes join [IP] ..." para añadir más nodos al cluster.

Finalmente, podemos visualizar que el nodo se ha instalado adecuadamente con el comando "kubectl get nodes".

A.6. Uniendo los nodos transmisores (nodos sensoriales)

Una vez iniciado el nodo principal, los nodos cliente se podrán unir al clúster simplemente introduciendo la IP obtenida en el paso interior pasándola como argumento al comando *kubernetes join*. Para visualizar los nodos que están en la red se puede utilizar el comando *kubectl get nodes* [50].

Apéndice B

Creando un microservicio simulador en Python

En este trabajo, al no disponer de sensores que midan los parámetros de salud del paciente, es necesario recurrir a desarrollar un simulador básico que los genere en tiempo real, como si de un paciente real se tratara. Para ello se ha utilizado Python, desarrollando un pequeño código que genera aleatoriamente cifras en función del rango en el que se encuentra el valor actual. Por ejemplo, si el ritmo cardíaco se encuentra en los umbrales superiores será más probable que este disminuya a que siga incrementándose.

Además de generar estos valores, será necesario conectarse con el servicio correspondiente de Redis para almacenarlos en la caché para así poder ser almacenados correctamente en la base de datos.

B.1. Conexión con el servicio Redis

Conectarse al servicio Redis será posible a través de tres parámetros, la dirección del servidor, el puerto, y la base de datos en cuestión. Para que este microservicio pueda ser utilizado con diferentes direcciones, puertos y bases de datos, se podrán modificar estos valores sin alterar el código, gracias a las variables de entorno. En el código se intentará acceder a estas variables de entorno *REDIS_HOST, REDIS_PORT y REDIS_DB*, y si no se consigue, se utilizarán los valores por defecto: el host será el mismo servidor y el puerto será el puerto por defecto del servicio Redis: 6379 (Código B.1).

Código B.1: Inicialización de la conexión al servicio Redis

```
1    redis_params = {
2        'host': os.environ.get('REDIS_HOST', 'localhost'),
3        'port': int(os.environ.get('REDIS_PORT', 6379)),
4        'db': int(os.environ.get('REDIS_DB', 0)),
5    }
```

```
6
7     r = Redis(**redis_params)
```

B.2. Generación aleatoria de valores

Para conseguir que no fuera totalmente aleatoria esta generación he recurrido a la función de selección aleatoria mediante un conjunto de probabilidades de la librería de Python *numpy* [35].

En esta sección se generará, por ejemplo, el latido de corazón de un adulto medio, de forma básica y simple. Por lo tanto, en este ejemplo, se sabe que el pulso cardíaco de un adulto medio, en reposo, ronda entre el rango entre 60 y 100 latidos por minuto. Se deberá de generar valores aleatorios que por lo general, estén normalmente entre este rango normal. Para ello he estimado un conjunto de probabilidades en función de la probabilidad que hay de que se decremente o incremente el valor entre medidas, es decir, es más probable que el latido sea incrementado en unas pocas unidades.

Código B.2: Lista de valores a sumar al valor actual

```
1 sum_set = [-20, -15, -10, -5, -4, -3, -2, -1, 1, 2, 3, 4, 5,
      10, 15, 20]
```

Código B.3: Lista de pesos (probabilidades) asignadas a cada posible valor

```
1 prob = [0.04, 0.06, 0.06, 0.06, 0.06, 0.06, 0.07, 0.09, 0.09,
      0.07, 0.06, 0.06, 0.06, 0.06, 0.06, 0.04]
```

Teniendo las probabilidades en un conjunto, y a su vez los valores del conjunto a seleccionar, se crea el bucle infinito que en cada iteración alterará el valor del ritmo cardíaco, comprobando en cada una de ellas si está fuera del rango establecido, y estableciendo también un margen de 10 segundos entre generación de valores (Código B.4).

Código B.4: Lista de pesos (probabilidades) asignadas a cada posible valor

```
1     heartbeat = 70
2     upper_limit = 90
3     lower_limit = 60
4
5     while True:
6         if (heartbeat > upper_limit):
7             heartbeat = heartbeat - random.randint(-5, 10)
8         elif (heartbeat < lower_limit):
9             heartbeat = heartbeat + random.randint(-5, 10)
10        else:
```

```
11                heartbeat = heartbeat + np.random.choice(
                      sum_set, p=prob)
12
13            send_heartbeat(heartbeat)
14            time.sleep(10)
```

B.3. Envío al servicio Redis

Para enviar en cada iteración el valor generado, una vez inicializado el servicio Redis, es tan simple como utilizar la variable de Redis y publicar el valor en la cola determinada, en este caso en 'heartbeat', para que este sea leido por el almacenador para así enviarlo a la base de datos (Código B.5).

Código B.5: Función para enviar un ritmo cardíaco al servicio Redis

```
1 def send_heartbeat(heartbeat):
2     r.publish('heartbeat', int(heartbeat))
```

Apéndice C

Creando un componente con una gráfica actualizada en tiempo real en React con Firestore

Desarrollar un componente que represente parámetros en una gráfica en tiempo real es algo muy util en este tipo de escenarios, donde además de poder visualizar un histórico de los datos sea posible también que los nuevos datos se representen en directo sin necesidad de actualizar la ventana. Para ello, utilizando la estructura de componentes de React [46], la base de datos no relacional Firestore, de Firebase (Google) [36], y la representación gráfica de la librería AmCharts [48], se puede generar de forma sencilla y rápida un servicio así.

C.1. Componente funcional de React

En esta aplicación se han utilizado componentes funcionales y no componentes de clase de React ya que los componentes funcionales son más fácil de leer y de probar, requieren de menos código y además en futuras versiones de React se mejorará el rendimiento de estos (David Jöch, 2018) [52].

Desde la actualización de React 16.8, se permite agregar un estado a estos componentes funcionales que en versiones anteriores no tenían. Este estado permitirá al componente tener un control de los cambios en la información que representa, así como tener en cuenta cuando aún no se ha descargado la información, cuando se tiene esta información, o cuando esta es modificada (Código C.1).

Código C.1: Ejemplo del estado de un componente funcional

```
1
2   const Componente : React.FC = () => {
3     const prueba = useState<number>(0)
4
5     return (
6         <>
7             <h1>Componente</h1>
8             <p>{ prueba }</p>
9         </>
10    )
11  }
```

Para desarrollar este componente es necesario contar con un estado que almacene un vector de objetos que tengan un valor y una referencia, este vector será actualizado conforme se reciban nuevos datos y en consecuencia, el componente se volverá a renderizar con la nueva información.

C.2. Actualizaciones en tiempo real de Firestore

Con el objetivo de que este estado sea actualizado en tiempo real conforme se reciban nuevos datos a representar, es necesario utilizar suscribirse a la colección que corresponda con el almacenamiento de Cloud Firestore. Para ello se utiliza la función *onSnapshot* que permite obtener actualizaciones en las colecciones de la base de datos [49].

Para integrar esta función en el componente y hacer que su estado se modifique a partir de cada actualización de la colección es necesario crear una función adicional en el componente que se suscriba a la colección seleccionada dentro de la función *useEffect* que a su vez nos permite ejecutar su contenido cuando el componente se haya renderizado.

En el siguiente ejemplo (Código C.2) se muestra cómo obtener 50 documentos de la subcolección *temperatura*, dentro de un usuario con identificador *userId*, donde se obtendrá un vector de objetos de temperatura ordenado en función del momento de la toma de datos, y será devuelto para actualizar el estado del componente.

Código C.2: Función para obtener actualizaciones en tiempo real con Firestore

```
1
2     const GetTemperaturesSnapshot = (userId: string) => {
3         const [temperatures, setTemperatures] = useState<
              any[]>();
4         const upper_range = 38;
5         const lower_range = 34;
6         useEffect(() => {
7           app
```

```
            .firestore()
            .collection("users")
            .doc(userId)
            .collection("temperature")
            .orderBy("timestamp", "asc")
            .limit(50)
            .onSnapshot((querySnapshot) => {
              const newTemperatures: any[] = querySnapshot.
                  docs.map((doc: any) => ({
                value: doc.data().temperature,
                date: doc.data().timestamp.toDate(),
                key: doc.id,
                alert:
                  doc.data().temperature > upper_range ||
                  doc.data().temperature < lower_range,
              }));

              setTemperatures(newTemperatures);
            });
        }, []);

        return temperatures;
      };
```

Después, ya se podrá utilizar esta función dentro del cuerpo del componente y en cada actualización de los datos, se actualizará el estado de este componente (Código C.3).

Código C.3: Llamada a la función anterior desde el componente

```

    const TemperatureComponent: React.FC = () => {
        const temperatures = GetTemperaturesSnapshot("000001")

        ...


    }
```

C.3. Representación en una gráfica con AmCharts

Utilizando esta librería, es posible generar una gráfica interactiva totalmente funcional a partir de un conjunto de datos. Así, utilizando el estado del componente, es posible inicializar una gráfica a partir del estado inicial del componente, y conforme lleguen nuevas actualizaciones, esta misma gráfica agregará los nuevos datos en tiempo real (Código C.4).

Código C.4: Inicialización de la gráfica a partir de los datos obtenidos

```
1
2     const initChart = (temperatures: any[]) => {
3         let chart = am4core.create("chartdiv", am4charts.
              XYChart);
4
5         chart.data = temperatures;
6
7         let dateAxis = chart.xAxes.push(new am4charts.
              DateAxis());
8         let valueAxis = chart.yAxes.push(new am4charts.
              ValueAxis());
9
10        let series = chart.series.push(new am4charts.
              LineSeries());
11        series.dataFields.dateX = "date";
12        series.dataFields.valueY = "value";
13
14        setChart(chart);
15      };
```

Finalmente, para representar la gráfica inicializada, es necesario crear un elemento div con el identificador especificado enteriormente (Código C.3).

```
1     <div id="chartdiv" style={{ width: "100%", height: "500
          px" }}></div>
```