FACULTAD
DE INGENIERIA
Universidad de Buenos Aires

CARRERA DE ESPECIALIZACIÓN EN
INTERNET DE LAS COSAS

MEMORIA DEL TRABAJO FINAL

Monitoreo ambiental integrado a Enterprise Buildings Integrator de Honeywell

Autor:
Ing. Gonzalo Nahuel Vaca

Director:
Esp. Ing. Pablo Almada (FIUBA-UTN)

Jurados:
Mg. Ing. Christian Yanez Flores (FIUBA)
Esp. Ing. Lucas Fabricio Monzón Languasco (FIUBA-UNNE)
Esp. Ing. Daniel Marquez (FIUBA-UC)

Este trabajo fue realizado en la Ciudad Autónoma de Buenos Aires, entre mayo de 2020 y abril de 2021.

Resumen

Esta memoria describe la implementación de una solución realizada para los laboratorios Gador, donde se adapta su sistema de automatización de edificios y gestión empresarial marca Honeywell. La finalidad es integrar una red de sensores que utilizan un protocolo de comunicación que este sistema no puede interpretar.

Se logró cumplir con las necesidades de Gador utilizando contenidos y habilidades desarrollados en las asignaturas de esta especialización. Se creó una arquitectura de datos, se implementaron protocolos de comunicaciones, se programaron servidores y se puso en funcionamiento un sistema de despliegue automático y orquestación de la aplicación.

Índice general

Índice de figuras

Índice de tablas

Dedicado a la memoria del Ing. Valeriy Omelchenko

Capítulo 1

Introducción general

El capítulo presenta las necesidades a satisfacer y una introducción técnica breve, con el objetivo de proveer los conceptos necesarios para comprender el resto de la memoria.

1.1. Motivación

La tendencia tecnológica actual es interconectar los dispositivos a través de la Internet, a tal punto que la cantidad de objetos en el año 2009 superó al número de personas conectadas, y llegado el 2020 la diferencia es de seis veces en favor de las cosas. [1] Los procesos industriales se ven beneficiados con nuevos métodos de control de inventarios y análisis de mediciones, además es posible gestionar los ambientes productivos para lograr una mayor calidad y comodidad. Los datos quedan disponibles para ser procesados por modelos de inteligencia artificial, y la información resultante puede ser vista desde cualquier ubicación y en múltiples plataformas.

Las empresas locales están retrasadas en su progreso tecnológico, muchas no incorporaron sistemas electrónicos en sus procesos o productos, es necesario crear un sistema que logre adaptar la tecnología en uso con el fin de incorporarlas a las nuevas prácticas de negocios. El avance tecnológico modifica el marco normativo de las naciones, para cumplir con los nuevos requerimientos jurídicos, se necesita tener un mínimo de capital. El retraso tecnológico ya no solo genera una pérdida de competitividad, sino que también impide que las empresas coloquen sus mercancías en otros países, por incumplimiento en normas de calidad o de protección del medio ambiente.

La situación de la industria argentina fue la primera razón que impulsó este trabajo, el siguiente paso fue buscar una empresa que quisiera participar de un proyecto adecuado para el pos-grado, y finalmente se logró un acuerdo con los laboratorios Gador S.A. La misión de la compañía es producir medicamentos para la salud humana, con la mejor calidad disponible, y ponerlos al alcance de la comunidad a precios accesibles. [2]

La empresa tiene la necesidad de acceder al mercado estadounidense y para lograrlo se deben satisfacer los requerimientos del *Code of Federal Regulations - Title 21 - Food and Drugs Chapter - Part 11 (21CFR11)*. La norma establece que los registros de las mediciones ambientales de los depósitos y cuartos productivos, se deben almacenar de forma electrónica, pero se debe demostrar que los registros tienen la misma validez y seguridad que aquellos hechos en papel. [3] Esto se

traduce en la necesidad de tener un sistema informático que se encuentre aprobado por la *Food and Drug Administration (FDA)*, que es el organismo encargado de controlar los medicamentos y alimentos que ingresan a los Estados Unidos. Por esa razón Gador tiene comprada una licencia del sistema *Enterprise Buildings Integrator (EBI)* de la marca *Honeywell*, ya que el producto se encuentra aprobado por la FDA. Si bien el programa fue adquirido hace varios años, es indispensable que continúe operando aún cuando los protocolos que utiliza son antiguos. Tampoco es económicamente viable reemplazarlo por un producto moderno que esté aprobado por la FDA, se debe lograr un salto tecnológico manteniendo la plataforma que actualmente está operando.

Los requerimientos que se deben cumplir en las mediciones ambientales de los depósitos y cuartos productivos, estipulan que los sensores se deben someter a un plan de calibración rutinario y realizarles periódicos estudios de perfiles térmicos. Un estudio de perfil térmico se logra tomando una serie de mediciones de temperatura en varios puntos de un ambiente, y con estos datos se procede a calcular las coordenadas de los puntos críticos del cuarto. [4] Los puntos críticos son aquellos lugares donde la temperatura es la más baja o más alta dentro de la habitación. Teniendo los puntos críticos identificados, se procede a colocar sensores de temperatura en esos lugares.

Los periódicos estudios de perfiles térmicos, tienen como consecuencia que cada seis meses se deben mover los sensores de temperatura. La tarea de migrar los dispositivos se vuelve costosa debido a que se encuentran cableados, y además los nuevos recorridos de los cables se deben certificar por el departamento de calidad. El tiempo de migración y de certificación se vería reducido sensiblemente si los equipos fuesen inalámbricos, pero la licencia de EBI que tiene Gador no es compatible con los protocolos de comunicaciones necesarios para lograrlo.

Las plantas de producción de la empresa siguen una arquitectura en donde los sensores reportan sus mediciones a unos controladores lógicos programables o PLC por sus siglas en inglés. Esa comunicación se logra a través de cables que los conectan. Los datos que adquieren los PLCs son entregados al sistema EBI utilizando un protocolo de comunicaciones llamado Modbus TCP. Este protocolo fue creado en el año 1979 y se diseñó teniendo en mente las limitaciones tecnológicas de la época, sin embargo la licencia de EBI que adquirió Gador solo acepta este formato. El modelo lógico de la arquitectura se puede visualizar en la figura 1.1, donde se puede ver una estructura del tipo árbol donde todo converge al sistema EBI.

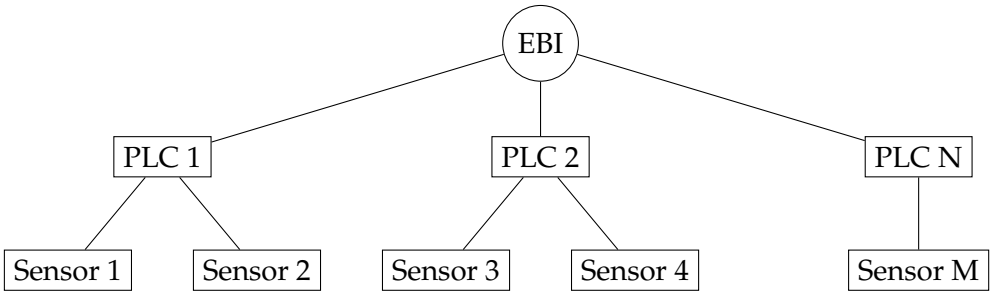

FIGURA 1.1. Red industrial Gador.

1.2. Introducción técnica

El proyecto realizado presentó una serie de desafíos a resolver, siendo el primero de ellos la variedad de tecnologías involucradas, tanto en la problemática a manipular como en la solución a implementada. Para introducir orden en la variedad de conocimientos que forman parte del sistema desarrollado, se introduce un modelo de capas de Internet de las cosas (IoT). El modelo tiene la ventaja que separa los temas en categorías relacionadas con la función o servicio que prestan en la solución, lo que facilita su estudio.

El modelo de capas seleccionado separa los conocimientos en cinco categorías, las cuales son las capas de negocio, aplicación, procesamiento, red y percepción. La capa de negocio agrupa todo lo relacionado con las reglas y el control del sistema, esto incluye supervisar el tráfico de información, verificar el estado de los equipos u otorgar permisos a los usuarios para interactuar con el programa. La capa de aplicación relaciona todas las tecnologías que se encargan de interactuar con el usuario final, es lo que las personas pueden ver como el sistema. La capa de procesamiento agrupa los conocimientos cuya responsabilidad es almacenar y analizar los datos que se generan. La capa de red tiene la finalidad de interconectar los dispositivos para permitir el flujo de datos entre todas las partes involucradas. Finalmente la capa de percepción se refiere a todos los artefactos que manipulan o miden algo que se encuentra en el ambiente, como un sensor o un actuador. Este modelo se encuentra resumido en la tabla 1.1.

TABLA 1.1. Modelo de capas IoT.

Capa	Función
Negocio	Establecer reglas y controlar el sistema
Aplicación	Interactuar con el usuario
Procesamiento	Almacenar y analizar los datos obtenidos
Red	Transportar los datos entre dispositivos
Percepción	Realizar mediciones o acciones en planta

Dependiendo del modelo viabilidad económica de un sistema y de como fue desplegado, la capa de negocio puede tener una funcionalidad contable y calcular los costos de operación. Esta capa puede ser la encargada de determinar y generar la facturación para cobrarle a los usuarios de la aplicación, como así también, de resolver operaciones de transferencia de dinero. La interacción en este nivel es con el personal que administra un sistema, se determina que permisos tiene cada usuario para manipular los servicios ofrecidos y se lleva adelante el registro de acciones y eventos relevantes para el normal funcionamiento del programa.

La experiencia que tiene el usuario al interactuar con la solución pertenece a la capa de aplicación. Aquí se define como se presenta la interfaz gráfica que utilizan las personas, y es común utilizar un formato de sitio web. Las páginas webs tienen la ventaja de ser indiferentes de la plataforma que utiliza el operador, solo importa que pueda ejecutar un navegador. Actualmente, se construyen las interfaces siguiendo un modelo de diseño según el tipo de operación a realizar por el programa, si la solución abarca una interfaz hombre-máquina industrial que debe ser atendida durante toda una jornada laboral, se suele implementar una norma de manejo de situaciones anormales o ASM; si la aplicación es de uso intermitente, se puede usar un esquema de diseño material o *Material Desing* que presenta

una experiencia moderna y fluida, como se puede apreciar en la figura 1.2. Para llevar a delante la interfaz seleccionada se utiliza un servidor que tiene como objetivo proveer los componentes gráficos al dispositivo utilizado, una manera de realizarlo es entregando al cliente una *Single Web Application (SWP)*, logrando que el servidor otorgue todo el código necesario para que el dispositivo del usuario genere por si mismo los componentes gráficos a mostrar. Es importante que el código entregado pueda ser visualizado en múltiples tamaños de pantallas, en la actualidad las personas utilizan ordenadores, tabletas y teléfonos móviles que presentan grandes diferencias en sus dimensiones, cuando una aplicación cumple con este requerimiento se dice que es responsiva.

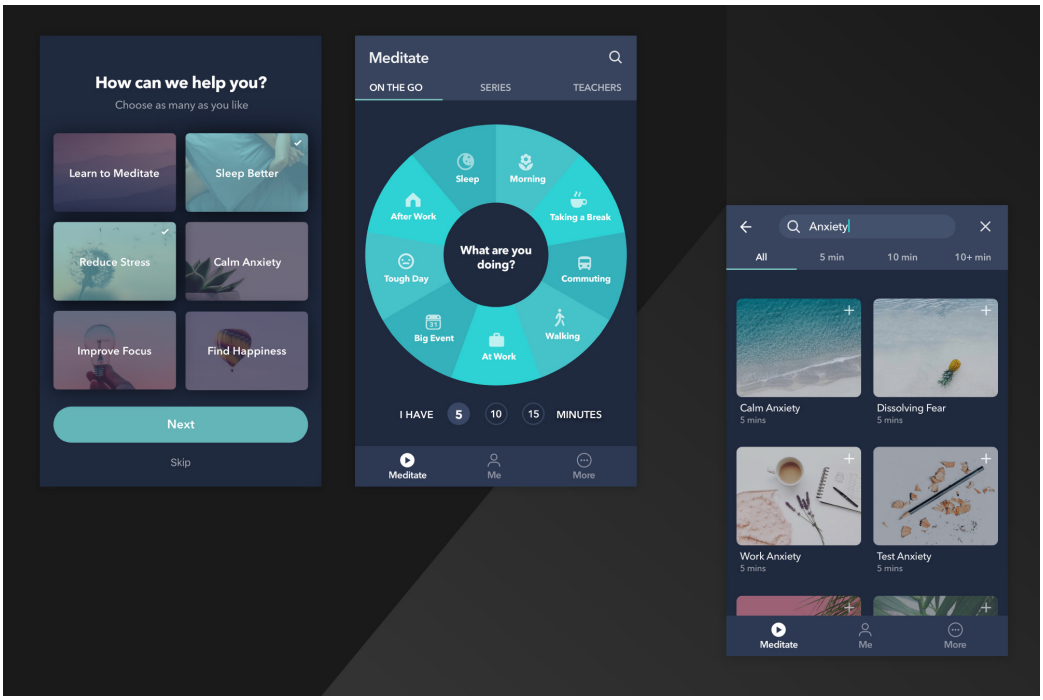

FIGURA 1.2. Ejemplo de interfaz de diseño material. [5]

Para alimentar de datos a la interfaz gráfica se necesita de la capa de procesamiento, que entrega el contenido a mostrar en pantalla. La información puede ser almacenada con distintas tecnologías, siendo una de las principales, las bases de datos relacionales. Este tipo de base de datos se basa en un esquema de tablas que se relacionan entre sí. Estas tecnologías se las suelen llamar SQL, y son utilizadas principalmente en datos de inventarios y sistemas de transacciones de dinero. Existe otro grupo de bases de datos que se denominan no relacionales o NoSQL, esta categoría contienen a las bases de datos tipo clave-valor, documental, de columnas y de grafos.

Las bases de datos clave-valor tratan los datos como una única colección que puede tener campos completamente distintos en cada registro, no existe entonces, ningún tipo de relación entre los miembros de la colección. El uso principal de esta tecnología es gestionar diccionarios dentro de la memoria volátil, ya que se pueden definir tiempos de vida para los datos. La muerte programada de un dato puede ser utilizada para gestionar las sesiones de usuario dentro del programa.

Almacenar los datos de manera documental significa que se agrupa la información siguiendo un criterio de entidades similares, lo cual no significa que exista

una estructura rígida, sino que los datos tienen una naturaleza similar. La persistencia se logra siguiendo un formato de codificación estandar como *XML*, *YAML* y *JSON*.

Las bases de datos orientadas a columnas están pensadas para minimizar el tiempo de búsqueda, principalmente en series temporales. La organización particular de este tipo de tecnologías es afín a los sistemas de IoT ya que los dispositivos de mediciones suelen generar un gran volumen de datos, que se pueden organizar como series temporales.

Una base de datos orientada a grafos presenta la información como nodos que se encuentran relacionados, la diferencia fundamental con los sistemas relacionales es que los nodos no están organizados en tablas, y las relaciones que unen los nodos tienen atributos y no poseen una estructura definida. Este tipo de tecnología permite utilizar la teoría de grafos y posibilita realizar consultas siguiendo modelos matemáticos que forman parte de esa rama de la ciencia.

Se dispone de un repertorio de protocolos pertenecientes a la capa de red para lograr que los dispositivos se comuniquen entre si. Entre los mencionados a lo largo de esta memoria se encuentran el protocolo Modbus, MQTT, HTTP y WebSocket. El manejo de estas tecnologías fue fundamental para lograr que las distintas partes del trabajo interactúen con el exterior.

Modbus es un protocolo que se diseñó teniendo en cuenta su uso para aplicaciones industriales, su prioridad es transmitir los datos manteniendo su integridad aún en ambientes donde el ruido eléctrico es elevado. El protocolo es público y gratuito, lo que provocó que se impusiera en un gran segmento del mercado ya que además es fácil de implementar y requiere poco desarrollo. Los dispositivos de una red Modbus tienen una dirección única y por lo general se asigna un equipo como maestro y el resto como esclavos. La arquitectura descripta presenta varias ventajas, pero la antigüedad del protocolo y su diseño para dispositivos del tipo PLC, hace que no sea adecuado para aplicaciones IoT.

Para interconectar a los dispositivos bajo un esquema de publicación-subscripción se utiliza el protocolo MQTT. El protocolo está diseñado para conexiones en lugares remotos donde los dispositivos funcionan con un ancho de banda limitado. El resultado es que los mensajes son pequeños y consumen poca batería de los equipos involucrados, por lo que se usa frecuentemente en los sistemas de IoT. El tráfico es gestionado por un servidor del tipo broker que decide quienes son los destinatarios de un mensaje en particular, el resto de los dispositivos son clientes del broker. Si un cliente desea transmitir datos, lo hace realizando una publicación a un determinado *topic* y el broker se encarga de determinar quienes deben recibir la información enviada. Quienes quieran obtener los datos publicados a un *topic* en particular, se deben suscribir a él ante el broker. Este al recibir una publicación de un cliente la transmite solo a los clientes que se encuentres subscritos, como se puede ver en el ejemplo de la figura 1.3, donde el cliente 2 no obtiene los datos del sensor porque no se encuentra subscrito.

El protocolo de transferencia de hipertexto (HTTP) está orientado a transacciones y sigue el esquema petición-respuesta entre un cliente y un servidor. El cliente inicia la comunicación enviando una petición al servidor, este último entrega una respuesta y se cierra el canal. Existe una variante del protocolo llamada HTTPS que agrega una capa de cifrado para que las comunicaciones sean seguras.

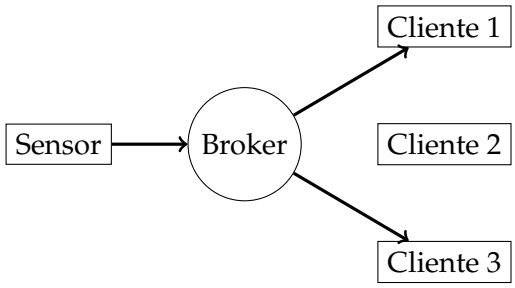

FIGURA 1.3. Ejemplo de comunicación MQTT.

WebSocket es un protocolo similar a HTTP pero con la diferencia que la conexión es bidireccional, esto quiere decir que cuando se logra la conectar al cliente con el servidor, ambos pueden enviar información espontáneamente. Esta cualidad permite realizar transferencias de datos en vivo, con lo que se pueden lograr servicios de *streaming* o *chats*.

Los sensores utilizados en una solución de IoT están incluidos en la capa de percepción, y para que puedan formar parte del sistema se necesita que sean capaces de soportar alguno de los protocolos de comunicaciones mencionados. Dado que el desarrollo de estos dispositivos no formaron parte del proyecto realizado, no se ampliará demasiado en este tema.

Teniendo definido los componentes de las capas, se necesita lograr que todas las partes funcionen como una única entidad. Para lograr este objetivo existen tecnologías de despliegue y orquestación, que cumplen la función de interconectar y mantener los servicios para que trabajen en equipo. Tradicionalmente se solían utilizar máquinas virtuales pero actualmente ese enfoque está quedando en desuso en favor de las tecnologías de contenedores. Una máquina virtual acapara parte de una computadora y funciona como un ordenador independiente, mientras que un contenedor funciona como un sistema operativo independiente pero no acapara los recursos de la computadora principal.

1.3. Estado del arte

En la sección 1.2 se presentó un modelo de capas para analizar las tecnologías. En esta sección se utiliza el mismo esquema para presentar las técnicas que conforman el estado del arte. Es importante mencionar el concepto de nube, ya que las soluciones modernas se basan en utilizar este tipo de plataforma. La nube se refiere a utilizar servicios y servidores provistos por un tercero. Entre los sistemas más representativos se encuentran *Amazon Web Service (AWS)*, *Google Cloud* y *Azure*. Estas empresas ofrecen su infraestructura y una serie de facilidades que promueven un rápido desarrollo y despliegue en el mercado.

Los sensores o actuadores que se utilizan corren un firmware específico para el ecosistema utilizado. Si se decidió utilizar AWS, por ejemplo, lo más probable es que la capa de percepción ejecute *AWS IoT Core* en sus dispositivos. Este esquema es ampliamente utilizado a nivel *enterprise*, por ejemplo, los laboratorios Bayer utilizan el ecosistema de AWS. [6]

En la capa de transporte, los dispositivos se comunican usualmente utilizando los protocolos *LoRaWAN*, *Sigfox*, *ZigBee* o *Bluetooth*. La selección del protocolo

depende de las distancias a cubrir y de las necesidades energéticas. Los sensores convergen luego a un punto de agregación. Desde los puntos de agregación se suelen transmiten los datos al servidor en la nube utilizando el protocolo MQTT.

En la capa de procesamiento se utiliza un esquema de datos de alta disponibilidad. Esto se logra creando réplicas de los datos en distintos servidores. Una de las réplicas se configura como maestro y el resto como esclavos. El servidor maestro es quien se comunica con el exterior de la réplica y retransmite los nuevos datos a los esclavos. Si un servidor maestro sufre un problema, uno de los esclavos se convierte en el nuevo maestro y se mantiene a la réplica funcionando sin interrupciones.

Los datos pueden ser divididos en *shards*, esto se hace para dividir la base de datos según la aplicación. Un ejemplo es separar los datos por región geográfica, de esta manera los clientes de una región en particular pueden tener los servidores con los datos que suelen utilizar cerca de ellos. Para que los *shards* funcionen como una única base de datos, se dispone de un servidor *router* que es la interfaz con el exterior. El *router* recibe las consultas o ingresos de nuevos datos y se encarga de utilizar el *shard* correspondiente. La configuración de este sistema se maneja desde un grupo de servidores destinados para tal fin. Suelen conformar una réplica donde solo se almacenan los datos de configuración. Esta arquitectura de alta disponibilidad se la conoce como granja de datos, se la puede construir con mongoDB [7] y se encuentra visualizada en la figura 1.4.

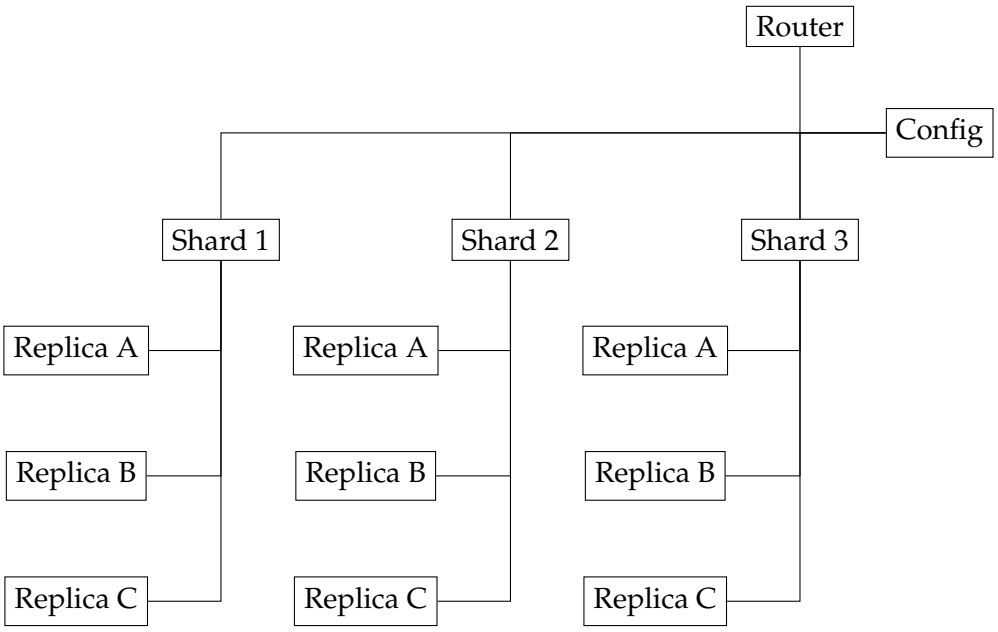

FIGURA 1.4. Arquitectura de datos de alta disponibilidad.

La capa de aplicación se suele diseñar rápidamente con un framework dedicado a la construcción de interfaces gráficas. Uno de los más utilizados es Angular, desarrollado por la empresa Google. El estilo gráfico de diseño material presentado en la sección 1.2 es la más utilizada. Las plataformas de nube ofrecen sus propios sistemas para diseñar la aplicación sin necesidad de escribir demasiado código, pero estas facilidades generan erogaciones adicionales.

La plataforma de nube presenta una capa de negocios donde se puede controlar el tráfico del sistema en ejecución. Desde allí se puede ver la facturación estimada o

el consumo de crédito para mantener en funcionamiento el proyecto. En esta capa se pueden cambiar las variables de entorno del sistema y se pueden controlar el estado de los componentes. Es posible montar servicios que corran programas como *Checkmk* o *Grafana* para visualizar el estado de los dispositivos en campo. O se puede optar por usar los servicios que ofrezca la empresa de nube.

La tecnología que se suele utilizar para orquestar toda la solución es Kubernetes, ya que además de automatizar el despliegue, también permite ajustar la escala. Ajustar la escala se refiere a la capacidad de crear una réplica de un servicio cuando uno de ellos está trabajando cerca de su límite de procesamiento. Es un sistema basado en contenedores y crea un *clúster* a partir de una plantilla donde se definen las reglas de escalamiento.

1.4. Objetivos y alcance

El objetivo principal que cumplió este proyecto fue demostrarle al cliente el potencial de las nuevas tecnologías y la posibilidad de integrarlas a sus actuales sistemas. Se propuso crear una prueba de concepto para evaluar la viabilidad de futuros proyectos. La creación de un sistema que pueda unir equipos que utilizan Modbus con aquellos que usan MQTT, es de relevancia en general para la industria local. Otro objetivo importante fue la de utilizar las técnicas adquiridas durante la cursada de la especialización. Con la finalidad de sementar los conocimientos a través de la práctica.

El proyecto se limitó a desarrollar el software a desplegar en un servidor que fue nombrado Nodos. Esto significa que no se contempló el desarrollo del hardware, en particular los sensores y los puntos de agregación. E esquema del servidor en la red de Gador puede ser visualizado en la figura 1.5. El servidor puede comunicarse con EBI de la misma manera que lo logra un PLC. Además tiene la capacidad de utilizar el protocolo MQTT para conectarse directamente con los sensores o a través de puntos de agregación.

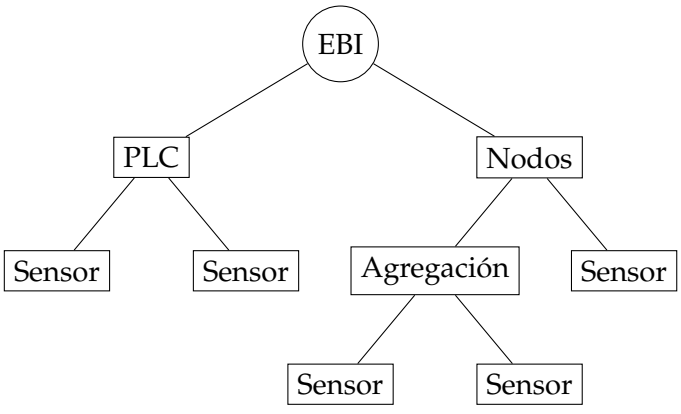

FIGURA 1.5. Red industrial Gador.

Cuando se tuvo definidos los objetivos y el alcance del proyecto, se inició un proceso de negociación con el cliente. Se buscó determinar cuales eran sus necesidades y sus temores respecto del proyecto. Las conversaciones con el cliente dieron como resultado la siguiente lista de requerimientos:

- Debe integrarse a la infraestructura de Gador S.A. sin generar conflictos en otros sistemas.
- Debe crear tramas en el formato *Enterprise Buildings Integrator* y enviarlas al servidor.
- Debe interpretar eventuales mensajes del servidor *Honeywell*.
- Debe interpretar los mensajes de los sensores.
- Debe poder cambiar la frecuencia de lectura de mediciones.
- Debe poseer la capacidad de gestionar los ingresos de usuarios de forma segura.
- Debe permitir que por lo menos cinco usuarios accedan al sistema simultáneamente.
- Debe presentar una interfaz donde se monitoree el estado de los sensores
- Debe permitir elegir un sensor en particular para editarlo.
- Debe poseer un módulo de gestión de usuarios.
- Debe ser compatible con ordenadores de escritorio y smartphones.
- Las contraseñas no persistirán como texto plano.
- Debe persistir todas las modificaciones realizadas a la configuración de los sensores.
- Debe persistir las mediciones obtenidas.

Capítulo 2

Introducción específica

Este capítulo trata sobre los recursos tecnológicos de terceros que fueron utilizados en la producción del trabajo.

2.1. Tecnologías utilizadas

Para crear una arquitectura de microservicios se utilizó Docker, que es un software que permite el uso y creación de contenedores de Linux. Un contenedor es una unidad que empaqueta el código de un programa junto con sus dependencias para aislar su funcionamiento. Para crear un contenedor, el motor de Docker se vale de el concepto de imagen. Las imágenes son entidades inmutables que cumplen la función de ser plantillas para crear contenedores. Se puede visualizar a las imágenes como capturas del estado de un contenedor y a partir de estas capturas se pueden instanciar nuevos contenedores. Los contenedores son entonces, unas abstracciones de la capa de aplicación de los sistemas Linux, como se puede visualizar en la figura 2.1. Varios contenedores pueden correr en el mismo ordenador como procesos aislados en el espacio de usuario sin generar ningún tipo de interferencias entre si. La principal diferencia con las máquinas virtuales, es que estas son una abstracción del hardware del ordenador, transformando una única computadora en varios servidores. Los contenedores, en cambio, utilizan el kernel del sistema operativo del ordenador físico, no se abstrae un kernel, solo el espacio de aplicación o de usuario.

Docker puede ser utilizado para construir imágenes definidas por el usuario. Para lograrlo se usa un Dokerfile, que es un documento de texto que contiene todos los comandos que un usuario utilizaría para ensamblar una imagen. El programador puede correr automáticamente una serie de comandos en sucesión al ejecutar una única orden sobre el Dockerfile. Otra capacidad adicional es subir la imagen a un repositorio para ser descargado directamente sobre el entorno de producción. De esta manera se facilita el despliegue de la aplicación, faltaría solamente, orquestar los contenedores para que trabajen en conjunto.

La orquestación necesaria para la etapa de despliegue se logra utilizando Docker Compose. Que según se indica en su documentación [8], es una herramienta para definir y correr aplicaciones de Docker de múltiples contenedores. Permite utilizar un archivo *YAML* para configurar los servicios de la aplicación. Con esta tecnología se pueden crear y comenzar todos los servicios de la configuración utilizando un único comando y finalmente se logra tener orquestada la solución.

El trabajo utiliza tecnologías web y la plataforma utilizada para implementarlas fue Nodejs, que es un servidor asincrónico y orientado a eventos que ejecuta

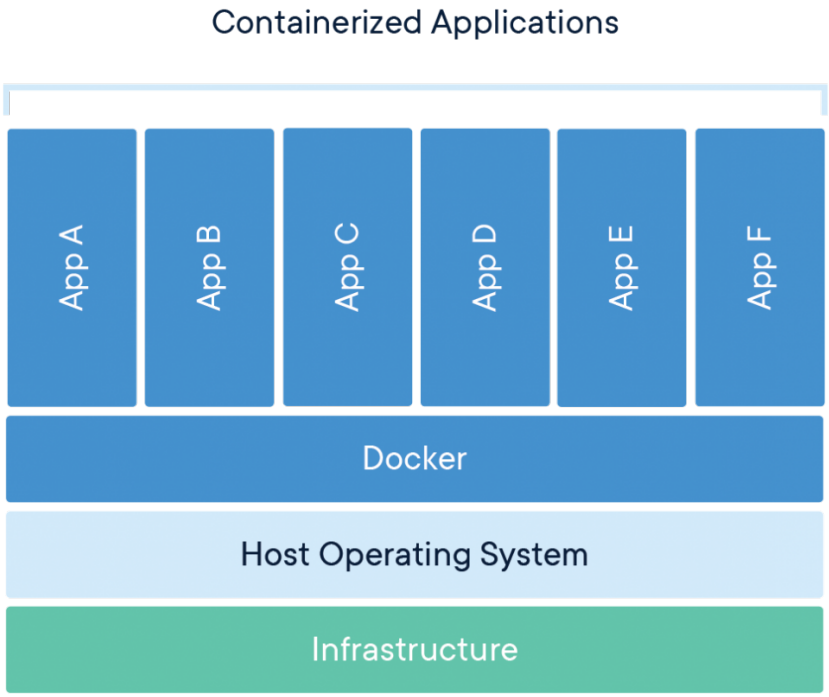

FIGURA 2.1. Arquitectura de Docker. [9]

JavaScript. La orientación a eventos se consideró como una ventaja frente a las aplicaciones de concurrencia de múltiples hilos en el sistema operativo. Principalmente porque no se utilizan candados y no existe la posibilidad de bloquear el servidor. Una particularidad adicional es que fue diseñado para construir aplicaciones de red escalables y provee un gestor de paquetes muy fácil de utilizar. Por estas razones y porque además fue una plataforma estudiada en la especialización, es que se la eligió para formar parte del trabajo.

No todos los servicios del trabajo se realizaron con tecnologías relacionadas a JavaScript. Particularmente se utilizaron una serie de bibliotecas y herramientas basadas en Python, que es un lenguaje de programación del tipo interpretado. Este lenguaje posee una gran cantidad de recursos hechos por terceros y que se encuentran a disposición con licencias libres. Si bien el lenguaje no fue visto con profundidad en la cursada, fue suficiente para entender su potencial. Específicamente porque muchas de sus bibliotecas fueron útiles para desarrollar algunos de los servicios más pequeños del trabajo. Es una tecnología que resultó muy útil y fue utilizada para acelerar la creación de las partes más livianas del sistema. Partes que no justificaron el uso de una plataforma como Nodejs.

Para implementar el protocolo MQTT se decidió utilizar paquetes y bibliotecas desarrollados por terceros. La primera herramienta fue Eclipse Mosquitto que es un broker del protocolo. Es liviano, no demanda grandes recursos y puede ser ejecutado en ordenadores monoplaca. Es flexible ya que permite ser configurado

de distintas maneras según las necesidades de la aplicación. Existen varios niveles de calidad de servicio para seleccionar y también se pueden agregar usuarios de manera opcional. Los usuarios poseen distintos permisos para publicar y suscribirse a diferentes *topics*. También es posible utilizar una capa de seguridad en los mensajes que se basa en la encriptación con llaves privadas y el uso de llaves públicas para realizar las lecturas. Uno de los atractivos de la plataforma es la serie de programas utilitarios que incluye como recursos adicionales y que se integran en la terminal del sistema operativo. Estas aplicaciones permiten realizar publicaciones y subscripciones para correr pruebas y además se pueden crear contraseñas encriptadas para los usuarios.

Para persistir la información durante la ejecución del trabajo se utilizó MongoDB. "MongoDB es una base de datos distribuida, basada en documentos y de uso general diseñada para desarrolladores de aplicaciones modernas y para la era de la nube". [10] Se decidió utilizar esta tecnología para la capa de procesamiento debido a la afinidad que posee para realizar aplicaciones IoT. Además es la base de datos documental más utilizada actualmente [11] y se evaluó como aspecto relevante, ganar experiencia con esta tecnología.

Los contenedores que forman la aplicación deben compartir una memoria volátil para que el trabajo pueda funcionar correctamente. Se eligió a Redis para que funcione como memoria compartida entre los componentes aislados dentro del trabajo. Redis es un almacén de estructuras de datos en memoria y puede ser usado como una base de datos, una *caché* de memoria y un broker para crear un bus de eventos. Esta tecnología permite hacer persistencia de datos pero solo en el esquema de clave-valor y permite también establecer los tiempos de vida para los datos. Los tiempos de vida de los datos se utilizaron en el trabajo para gestionar las sesiones de los usuarios. Si una persona pasa demasiado tiempo sin interactuar con la aplicación tiene que volver a ingresar su usuario y contraseña para seguir utilizando el sistema con normalidad.

2.2. Bibliotecas y paquetes de terceros

Para implementar la capa de aplicación se utilizó un framework de diseño de interfaces gráficas llamado Angular y se creó una SWA. Se desarrolla el código usando el lenguaje TypeScript que es un *superset* de JavaScript. Está basado en el paradigma de componentes para generar aplicaciones escalables y poder reutilizar el código escrito en otros trabajos. Pone a disposición una colección de bibliotecas para manejar formularios, comunicaciones cliente-servidor, *routing*, entre otras funcionalidades. Tiene entre sus facilidades ofrecidas, un cliente de terminal que acelera los tiempos de desarrollo. Por todas estas ventajas, se decidió inclinarse por este framework.

El estilo empleado en el trabajo fue el de diseño material. Para lograrlo se trabajó con Angular Material que es una biblioteca de componentes de interfaz gráfica para Angular. Está pensada para construir una experiencia consistente y funcional, adhiriendo con los principios de diseño más utilizados. Estos principios se refieren a la portabilidad entre navegadores y la independencia de dispositivos.

Se necesitó crear una *REST API* y para lograrlo se usó un framework para realizar servidores con Nodejs denominado Express. El framework está pensado para construir aplicaciones webs e interfaces de programación de aplicaciones (API).

Una API se encarga de definir las interacciones entre múltiples programas o puede intermediar entre componentes de hardware y software. Express es la biblioteca más utilizada para trabajar con Nodejs. [12]

Para solucionar el problema de intercambio de recursos de origen cruzado del frontend (CORS) se utilizó la biblioteca CORS para Nodejs. CORS es una solución que permite acceder a recursos restringidos de un dominio diferente al del frontend.

Eclipse Paho es una biblioteca MQTT que está disponible para varios lenguajes de programación. En el trabajo realizado se utilizó para darle conectividad a los servicios programados en Python. La biblioteca provee una clase cliente que habilita la comunicación con un broker MQTT. Se ofrece además, una serie de funciones auxiliares que permiten codificar el programa con mayor facilidad.

Se necesitó un recurso que ofrezca de capacidades MQTT al código escrito en JavaScript y para esa misión se usó MQTTjs. MQTTjs es una biblioteca que provee de las herramientas para crear un cliente MQTT en Nodejs y en un navegador. Se lo puede instalar de forma global en el sistema operativo para hacer uso de las herramientas de terminal que ofrece. Las herramientas permiten hacer pruebas de subscripción y publicación de mensajes.

Para cumplir con el requerimiento de la sección 1.4 que demanda ver en vivo las mediciones al calibrar un instrumento, se utilizó WS. Esta dependencia es una biblioteca de Nodejs que se usa para realizar servidores con la capacidad de entablar una conexión WebSocket. Es fácil de utilizar y es altamente configurable. Se decidió incorporarla al trabajo debido a que la documentación es completa y simple de entender.

Mongoose es una biblioteca de modelado de datos de objetos(ODM). Un ODM gestiona las relaciones entre los datos, provee de una validación de esquema y se usa para traducir los objetos del programa en ejecución en una representación dentro de la base de datos. La biblioteca se creó para trabajar con MongoDB y está disponible para la plataforma Nodejs.

Se necesitó cumplir con las necesidades de seguridad que se enumeraron en la sección 1.4, para lograrlo se usaron dos bibliotecas compatibles con Nodejs. Bcrypt es una biblioteca que contiene funciones para encriptar contraseñas. La encriptación se basa en un esquema de *hashing* e incorpora una *salt* para proteger los datos de un ataque *rainbow table*. Para darle resistencia a los ataques de búsqueda por fuerza bruta, la biblioteca implementa una función adaptativa que por cada iteración se vuelve más lenta. Esto hace que su resistencia sea fuerte aún cuando el ordenador que realiza el ataque sea potente. Por estas razones se decidió usar este recurso en el trabajo, con la idea de proteger las contraseñas de los usuarios evitando que persistan como texto plano.

La segunda bibliteca utilizada para cumplir con los requerimientos de seguridad fue Jsonwebtoken. Que es un paquete que permite crear un *JavaScript Web Token (JWT)*. JWT es un estándar que se utiliza para la fabricación de *tokens* de acceso que permiten identificar una entidad y determinar cuales son sus privilegios en el sistema. El token está formado por una cabecera, una carga útil y una firma. La cabecera identifica el algoritmo de encriptación y el tipo de token. La carga útil lleva consigo la información relevante para el funcionamiento de la aplicación.

Finalmente la firma cierra el token para certificar la llave privada del servidor. Es relevante mencionar que la biblioteca está diseñada para funcionar con Nodejs

La biblioteca Chai ofrece herramientas para hacer pruebas del software escrito. Fue diseñada para Nodejs y puede ser integrada a cualquier framework de JavaScript. En el trabajo fue combinado con Mocha, que es un framework de automatización de pruebas para Nodejs. Las pruebas obtenidas al combinar estos dos recursos fueron fundamentales para detectar comportamientos no deseados.

Para que el código escrito en Python pueda utilizar el protocolo ModbusTCP se usó la biblioteca PyModbusTCP. Este recurso permite crear un cliente para acceder a un servidor o bien crear una aplicación que se comporte como esclavo.

Oitc/modbus-server es un servidor ModbusTCP realizado en Python y disponible como imagen de Docker. Está configurado para utilizar el puerto 5020 para evitar problemas de permisos con el sistema operativo. [13] El sistema de orquestación corrige el puerto al conectar el 5020 del contenedor con el 502 del ordenador.

Todas las dependencias del trabajo se pueden visualizar en la tabla 2.1.

TABLA 2.1. Dependencias del trabajo.

Dependencia	**Función**
Docker	Motor de contenedores
Docker Compose	Orquestación de contenedores
Dockerfile	Creación de imágenes para Docker
Nodejs	Servidor para JavaScript
Python	Lenguaje para los servicios ligeros
Eclipse Mosquitto	Broker MQTT de la aplicación
MongoDB	Base de datos
Redis	Memoria compartida entre contenedores
Angular	Framework para crear la SWA
Angular Material	Componentes gráficos para Angular
Express	REST API para Nodejs
CORS	Intercambio de recursos de origen cruzado
Paho MQTT	Biblioteca MQTT para Python
MQTTjs	Biblioteca MQTT para Nodejs
WS	Funcionalidad WebSocket para Nodejs
Mongoose	ODM para Nodejs y MongoDB
Bycrypt	Seguridad de contraseñas
JsonWebToken	Seguridad de sesiones de usuarios
Chai	Pruebas unitarias para JavaScript
Mocha	Pruebas automáticas para Nodejs
PyModbusTCP	Biblioteca Modbus para Python
Oitc/modbus-server	Servidor Modbus

2.3. Sistema propietario del cliente

Como se explicó en el capítulo 1, el trabajo se basa en lograr integrarse al sistema que Gador tiene en ejecución. Esta sección explica con mayor detalle la naturaleza de este producto.

EBI es un sistema de automatización de edificios y gestión empresarial creado por la firma Honeywell. Ofrece herramientas para dotar a las dependencias de la empresa de la inteligencia necesaria para incrementar la comodidad, mejorar la seguridad y reducir los costos operativos.

La solución tiene la facultad de gestionar una red de edificios a través de una única interfaz gráfica. Pretende reducir los tiempos de respuesta frente a situaciones anormales y mejorar la seguridad. Esta tecnología es compatible con dispositivos y software de terceros, con la idea de ofrecer escalabilidad.

Este software es un ecosistema de módulos que pueden ser adquirido a través de licencias para agregar las siguientes funcionalidades:

- Gestión de consumo energético
- Seguridad de vida
- Control de acceso e intrusión
- Vídeo vigilancia

Las distintas licencias que vende Honeywell permiten que EBI utilice los protocolos BACNet, OPC, LonWorks y ModbusTCP. Gador adquirió el producto con la licencia ModbusTCP para conectar sus PLCs de variadas marcas. Como se puede observar en la figura 2.2, se utiliza el software para visualizar los sensores de los distintos cuartos de producción y depósitos. En la figura 2.3 se puede visualizar que EBI está a cargo del control ambiental de las plantas del cliente. Este sistema es el corazón de la gestión de edificios de la compañía y determinó los requerimientos que propuso Gador.

La importancia de EBI para mantener la operación diaria de las plantas de Gador hizo que se limitara el acceso para hacer pruebas. El servidor que corre este programa se encuentra custodiado con gran celo dentro de la compañía. No se pudieron correr pruebas directamente sobre el ambiente de producción, solo se pudo utilizar por algunas horas una máquina virtual entregada con una licencia de evaluación que tuvo seis horas de duración.

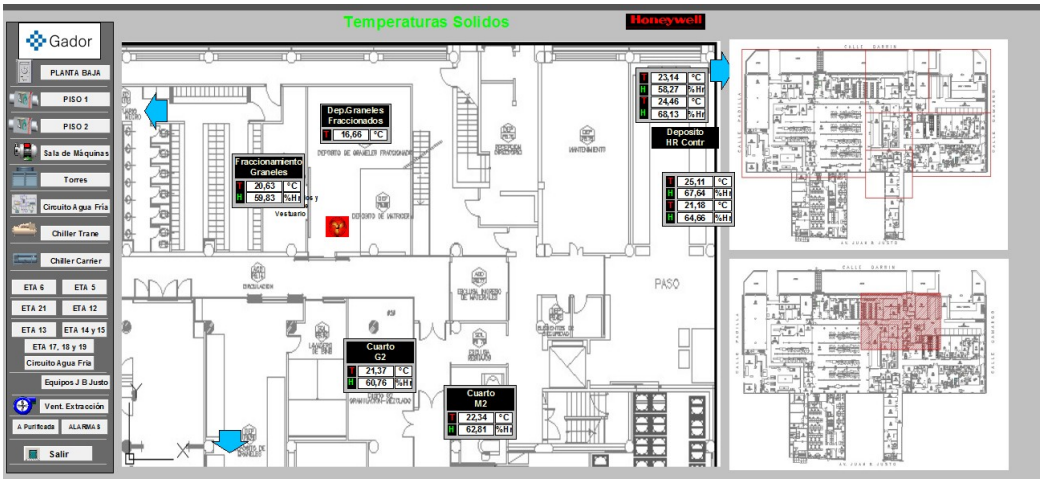

FIGURA 2.2. Control de temperatura de sólidos.

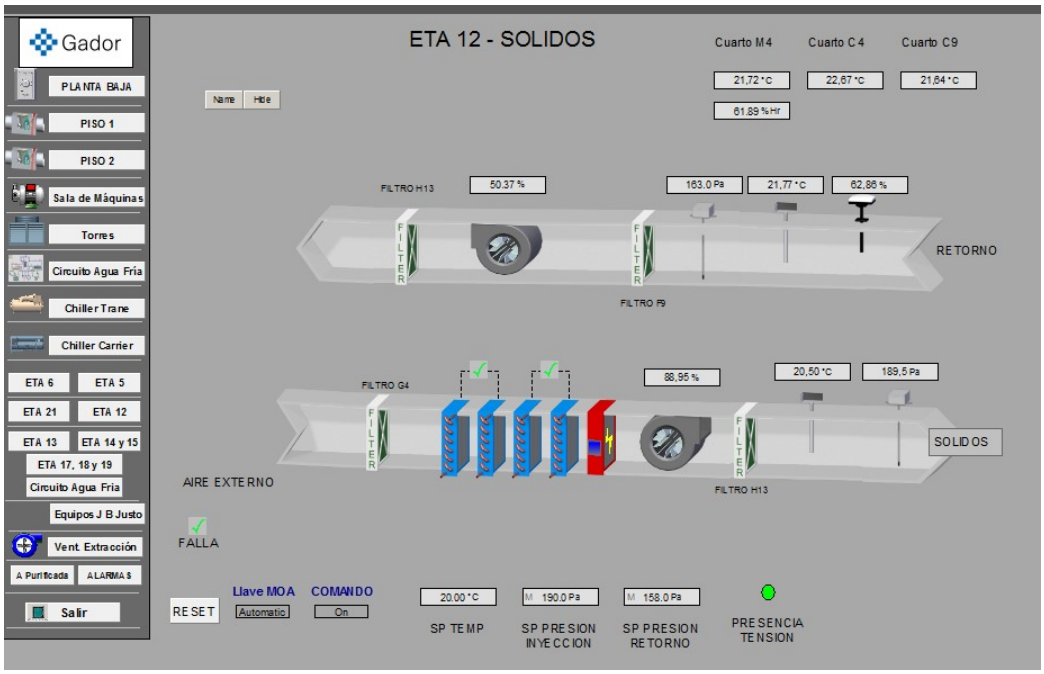

FIGURA 2.3. Unidad de tratamiento de aire de sólidos.

Capítulo 3

Diseño e implementación

En este capítulo se detallan los componentes realizados por el autor de esta memoria. Se explica como se crearon los servicios y como se interconectaron para lograr que todo funcione como una única solución.

3.1. Arquitectura y orquestación

Esta sección trata sobre la conexión entre los servicios del trabajo y su despliegue automático.

Para planificar la orquestación se analizaron los servicios que debían ser accesibles desde entidades externas al servidor. En la figura 3.1 se pueden observar las conexiones lógicas entre los contenedores. Destacadas en color rojo, se encuentran las entidades externas que interactúan con el servidor Nodos. Las interconexiones se simplificaron al crear una capa de puente de red que corre sobre el *Daemon* de Docker. El resultado es que cada contenedor pasa a tener una dirección de ip dentro del entorno. La creación de la red se logró con el código 3.1.

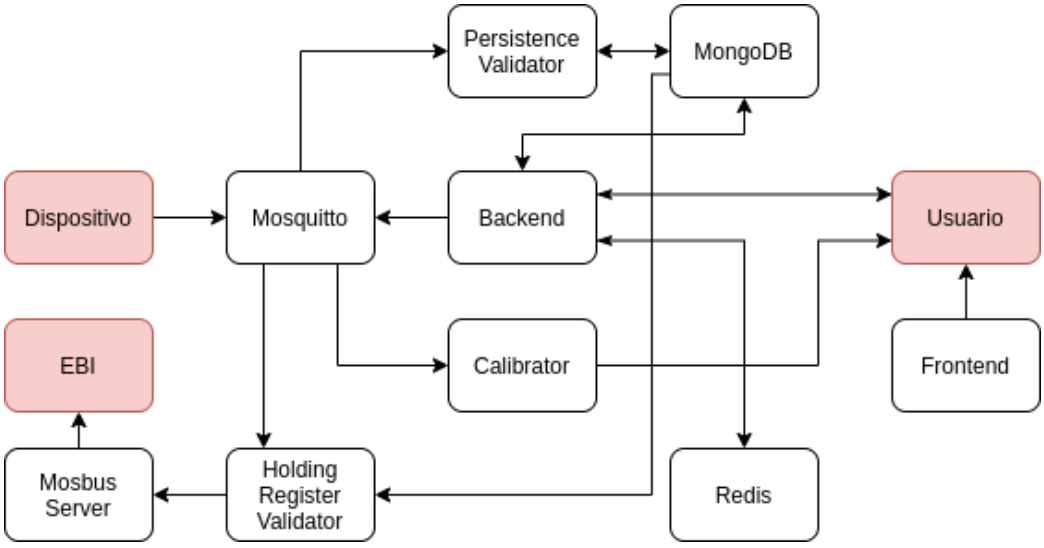

FIGURA 3.1. Esquema de conexión de los servicios.

```
networks:
  iot:
    driver: bridge
```

CÓDIGO 3.1. Red de interconexión Docker Compose.

El servicio modbus-server se comunica al exterior utilizando el puerto 502. El puerto pertenece a la lista de protocolos bien conocidos o puertos de sistema. Esta condición hace posible que existan problemas con los permisos que el usuario tiene dentro del sistema operativo. Como se puede observar en el código 3.2, se conectó el puerto 502 del ordenador con el 5020 dentro de la red de Docker. En general, es una buena práctica que los puertos internos de la red no sean puertos de sistema para no generar conflictos de permisos y conectarlos a puertos de protocolos según sea necesario. Para evitar usar direcciones ip en el código de los servicios se utilizó el parámetro hostname para utilizar el servicio de sistema de nombres de dominio (DNS) que corre dentro del *Daemon* de Docker.

```
modbus-server:
  image: oitc/modbus-server
  container_name: modbus-server
  hostname: modbus-server
  restart: always
  ports:
    - '502:5020'
  expose:
      - '5020'
  networks:
    - iot
```

CÓDIGO 3.2. Orquestación del servidor Modbus.

El servicio Mosquitto se configuró con tres volúmenes que conectan al contenedor con archivos no efímeros que persisten la información necesaria para que el contenedor muestre un comportamiento correcto. Como se puede apreciar en el código 3.3, se encuentran los archivos de configuración de usuarios y de lista de control de acceso (acl). Con esta configuración se evita que dispositivos anónimos puedan utilizar el broker y que además solo puedan utilizar los *topics* designados. Adicionalmente, los distintos usuarios tienen diferentes permisos según el *topic*. De esta manera se logra una mayor confiabilidad y seguridad en el manejo de los mensajes.

```
mosquitto:
  image: eclipse-mosquitto
  container_name: mosquitto
  hostname: mosquitto
  restart: always
  volumes:
    - ./mosquitto/mosquitto.conf:/mosquitto/config/mosquitto.conf
    - ./mosquitto/users.txt:/mosquitto/config/users.txt
    - ./mosquitto/acl.txt:/mosquitto/config/acl.txt
  expose:
    - '1883'
    - '9001'
  ports:
    - '1883:1883'
    - '9001:9001'
  networks:
    - iot
```

CÓDIGO 3.3. Orquestación del broker Mosquitto.

El servicio Holding Registers Validator (hrv) tiene la particularidad de depender de otros servicios, como se puede ver en el cógigo 3.4. El contenedor no puede ser creado hasta que los servicios listados como dependencias se encuentren activos. Esto se hace de esta manera para evitar que el contenedor genere excepciones y se

reinicie varias veces durante el despliegue se la solución. Además no es posible saber si el comportamiento final del contenedor puede quedar indefinido. Es importante mencionar que este servicio no tiene salida al exterior y no queda visible por la falta de campos *ports*.

La imagen para construir el contenedor no existe y debe ser creada al momento del despliegue. Para lograrlo se utiliza el campo *build*, en donde se especifica la ruta al Dockerfile que contiene la receta. La imagen queda guardada con el nombre vaca/hrv, de esta manera, no es necesario volver a construirla si se decide reiniciar la aplicación.

```
hrv:
  build: ./holdingRegistersValidator/
  image: vaca/hrv
  container_name: hrv
  hostname: hrv
  restart: always
  expose:
    - '1883'
    - '5020'
  depends_on:
    - 'mosquitto'
    - 'mongo'
    - 'modbus-server'
  networks:
    - iot
```

CÓDIGO 3.4. Orquestación del servicio hrv.

El Dockerfile que fabrica la imagen puede verse en el código 3.5. Este código es común para todos los Dockerfiles que construyen imágenes para los servicios realizados en Python. Se utiliza Alpine Linux como imagen base y se genera el usuario y grupo *pythonuser*. El usuario es quien corre el servicio dentro del contenedor y se definió un comando a ejecutar al momento de crearlo. Quedan indicados en este archivo cuales son puertos que se pueden usar para la red puente.

```
FROM python:3.8-alpine
LABEL maintainer="Gonzalo Nahuel Vaca <vacagonzalo@gmail.com>"
RUN addgroup -g 1000 -S pythonuser && \
  adduser -u 1000 -S pythonuser -G pythonuser && \
  mkdir -p /app && \
  pip3 install pyModbusTCP && \
  pip3 install paho-mqtt && \
  pip3 install pymongo
ADD --chown=root:root app/* /app/
USER pythonuser
EXPOSE 1883 27017 5020/tcp
CMD [ "python", "-u", "/app/service.py" ]
```

CÓDIGO 3.5. Dockerfile del servicio hrv.

El servicio MongoDB no tiene salida al exterior del servidor, su configuración se puede ver en el código 3.6. La configuración tiene la particularidad de introducir un comando a la hora de crear el contenedor. Se le indica al motor de MongoDB que puerto debe escuchar. Además se crea un volumen donde figuran una serie de archivos que pueblan la base de datos con información para construir una maqueta. Esta maqueta fue utilizada para realizar la demostración al cliente. Los archivos son devices.js, measurements.js, seed.js y users.js. El archivo devices.js crea una serie de dispositivos ficticios. El archivo measurements.js inserta una

serie de mediciones que provienen de los dispositivos ficticios creados por devices.js. El script users.js genera una serie de usuarios con distintos permisos, con el fin de probar la capacidad de autentificar las sesiones. Finalmente seed.js es quién carga todos los datos en MongoDB, según se puede ver en el código 3.7. Las mediciones fueron cargadas múltiples veces para generar un volumen de datos que sirviera para realizar pruebas.

```
mongo:
  image: mongo
  container_name: mongo
  hostname: mongo
  command: mongod --bind_ip_all --port 27017
  expose:
    - '27017'
  volumes:
    - ./mongodb/scripts:/scripts
  networks:
    - iot
```

CÓDIGO 3.6. Orquestación de MongoDB.

```
use gador;
load("scripts/devices.js")
load("scripts/users.js")
load("scripts/measurements.js")
load("scripts/measurements.js")
load("scripts/measurements.js")
load("scripts/measurements.js")
load("scripts/measurements.js")
load("scripts/measurements.js")
load("scripts/measurements.js")
load("scripts/measurements.js")
load("scripts/measurements.js")
```

CÓDIGO 3.7. Seed de la base de datos.

El servicio Persistence Validator (pv), está definido en el código 3.8. Se puede observar que la imagen no existe y debe ser construida. Como el servicio fue realizado en Python, el Dockerfile necesario para construir la imagen es prácticamente idéntico al visto en el código 3.5.

```
  pv:
    build: ./persistenceValidator
    image: vaca/pv
    container_name: pv
    hostname: pv
    restart: always
  expose:
    - '1883'
    - '27017'
  depends_on:
    - 'mosquitto'
    - 'mongo'
  networks:
    - iot
```

CÓDIGO 3.8. Orquestación del servicio pv.

Para orquestar el servicio backend se utilizó el puerto 8080 del ordenador. La razón es que se tiene una comunicación con una entidad externa, el usuario. La imagen para crear el contenedor debe ser construida y para tal fin se usó el Dockerfile que se puede observar en el código 3.10.

```
backend:
  build: ./backend
  image: vaca/backend
  container_name: backend
  hostname: backend
  expose:
    - '1883'
    - '6379'
    - '27017'
  ports:
    - '8080:8080'
  depends_on:
    - 'mosquitto'
    - 'mongo'
    - 'redis'
  networks:
    - iot
```

CÓDIGO 3.9. Orquestación del servicio Backend.

El Dockerfile parte de la imagen oficial de Nodejs y copia los archivos de dependencias dentro del contenedor auxiliar. Con este archivo se descargan las bibliotecas necesarias. Luego se copia el código fuente de la aplicación y finalmente se configura la inicialización del servidor Nodejs como comando por defecto.

```
FROM node
LABEL maintainer="Gonzalo Nahuel Vaca <vacagonzalo@gmail.com>"
WORKDIR /usr/src/app
COPY package*.json ./
RUN npm install
COPY . .
EXPOSE 1883 6379 8080 27017
CMD ["node", "./src/app.js"]
```

CÓDIGO 3.10. Dockerfile del servicio Backend.

El servicio Calibrator es una aplicación de Nodejs y fue orquestada de manera similar al servicio Backend. Su configuración se puede observar en el código 3.11. El Dockerfile necesario para crear su imagen es prácticamente idéntico al mostrado en el código 3.10.

```
calibrator:
  build: ./calibrator
  image: vaca/calibrator
  container_name: calibrator
  hostname: calibrator
  expose:
    - '1883'
    - '9999'
  ports:
    - '9999:9999'
  depends_on:
    - 'mosquitto'
  networks:
    - iot
```

CÓDIGO 3.11. Orquestación del servicio Calibrator.

El servicio Redis fue creado a partir de la imagen oficial de Redis que se encuentra disponible en Dockerhub. [14] Como no tiene exposición al exterior del servidor, no se necesitó realizar ninguna configuración adicional. Es importante aclarar que

si bien se puede aplicar una capa de seguridad, no es aconsejable exponer a Redis a la Internet. La orquestación se puede ver en el código 3.12.

```
redis:
  image: redis
  container_name: redis
  hostname: redis
  expose:
    - '6379'
  networks:
    - iot
```

CÓDIGO 3.12. Orquestación del servicio Redis.

El último servicio es el Frontend que fue orquestado como se puede observar en el código 3.13. Su imagen se crea usando el código fuente de la aplicación y un Dockerfile al momento de orquestar la solución. La construcción de esta imagen es la más sofisticada de todo el trabajo, como se puede ver en el código 3.14.

```
frontend:
  build: ./frontend/
  image: vaca/frontend
  container_name: frontend
  hostname: frontend
  restart: always
  ports:
    - '80:80'
```

CÓDIGO 3.13. Orquestación del servicio Frontend.

El Dockerfile se divide en dos grandes etapas. La primer parte es crear un contenedor auxiliar a partir de la imagen oficial de Nodejs y llamarla *builder*. Este contenedor temporal copia dentro suyo el código fuente del servicio e instala todas las dependencias. Entre las dependencias instaladas se encuentra el framework de Angular. Se utiliza el framework para compilar el código fuente de TypeScript y se obtienen archivos en JavaScript, que son ejecutables por un navegador. La segunda parte del proceso es crear un contenedor a partir de la imagen oficial de Nginx, que es un servidor web. Se transfieren los archivos compilados por el contenedor auxiliar hacia el contenedor de Nginx y se destruye el auxiliar. Finalmente se transforma el contenedor de Nginx con los archivos compilados en una imagen.

```
FROM node as builder
WORKDIR /src/app
COPY . ./
RUN npm install
RUN npm run ng build --prod
FROM nginx
COPY --from=builder /src/app/dist/frontend /usr/share/nginx/html
```

CÓDIGO 3.14. Dockerfile del servicio Frontend.

Teniendo definido los Dockerfiles y el archivo docker-compose.yml se puede utilizar un guión escrito en Bash que lanza la aplicación. Como se puede observar en el código 3.15.

Cuando se desea eliminar todo rastro del sistema del ordenador, se puede utilizar el código 3.16.

Los únicos requisitos para iniciar la aplicación es tener un ordenador que tenga Docker y Docker Compose instalados. No se necesita tener ninguna de las herramientas de desarrollo dentro del ambiente de producción. De esta manera se logró una solución altamente portátil y agnóstica de la arquitectura del hardware.

```
#!/bin/bash
chmod +x clean.sh
docker-compose up -d
printf "Waiting 10 seconds for internal connections to be made"
sleep 10
docker-compose exec mongo sh -c "mongo < /scripts/seed.js"
printf "checking collections"
docker-compose exec mongo sh -c "mongo < /scripts/seed.test.js"
```

CÓDIGO 3.15. Guión de inicialización.

```
#!/bin/bash
docker-compose down
docker rmi vaca/backend
docker rmi vaca/hrv
docker rmi vaca/pv
docker rmi vaca/auth-api
docker rmi vaca/calibrator
docker rmi vaca/frontend
clear
```

CÓDIGO 3.16. Guión de limpieza.

3.2. Servicios orientados a dispositivos

Mosquitto fue configurado siguiendo su documentación para lograr el máximo nivel de seguridad que no incluyera certificados *SSH*. Se decidió no utilizar certificados ya que no figuraban en los requerimientos y el broker no se encuentra expuesto a la Internet. Además uno de los motivos del trabajo es simplificar la interacción con los sensores. Agregar la tarea de controlar y renovar los certificados iba en contra del objetivo inicial. La configuración puede ser observada en el código 3.17.

```
allow_anonymous false
password_file /mosquitto/config/users.txt
acl_file /mosquitto/config/acl.txt
```

CÓDIGO 3.17. Archivo mosquitto.conf

Se creó una configuración de usuarios que tiene en su interior dos integrantes. El primer usuario se denominó docker y su contraseña es container. El segundo usuario se nombró device y su contraseña es thing. Las contraseñas se guardaron encriptadas utilizando la herramienta *mosquitto_passwd*. El usuario docker se utiliza para que los contenedores se comuniquen con el broker, mientras que device se asigna a los sensores en campo. Las diferencias entre contenedores y dispositivos se configuran en el archivo de acl, como se visualiza en el código 3.18. El topic cmnd se utiliza para los mensajes que alteran la configuración de los dispositivos. El topic data tiene la finalidad de llevar las mediciones tomadas por los sensores y hacerlas persistir en la base de datos. Además de escribir en una posición de memoria del servidor Modbus, según corresponda. Finalmente el topic live tiene la función de llevar las mediciones que se realizan durante las calibraciones.

```
user docker
topic readwrite cmnd/#
topic read data/#

user device
topic readwrite cmnd/#
topic write data/#
topic write live/#
```

CÓDIGO 3.18. Lista de control de acceso

El servicio Holding Register Validator (HRV) tiene la misión de determinar que mediciones se deben escribir en una posición de memoria del servidor Modbus. Para cumplir con esa función, HRV se subscribe al topic data y recibe todos los reportes de los sensores. Luego determina si las mediciones recibidas pertenecen a la lista de dispositivos que figuran en la base de datos y si corresponde escribir una posición de memoria. Finalmente escribe una posición de memoria del servidor según corresponda. Las funciones de cada paso fueron extraídas y se presentan en el código 3.19, escritas en Python.

```
# MQTT
def onMessage(client, userdata, msg):
    data = msg.payload.decode().split(",")
    addr = getAddr(data[0])
    if addr != -1:
        val = int(data[1])
        write_slave(addr, val)

# DATABASE
def getAddr(id):
    global devices
    d = devices.find_one({"tag": id})
    if d is None:
        return -1
    if 'modbus' in d:
        return int(d['modbus'])
    return -1

# MODBUS
def write_slave(addr, value):
    global master
    if master.write_single_register(addr, value):
        print('writing successful')
    else:
        print('writing error')
```

CÓDIGO 3.19. Funciones principales del servicio HRV

El servicio Persistence Validator (PV) tiene la función de recibir las mediciones de los sensores y decidir si deben persistir en la base de datos. Para tal fin se encuentra subscrito al topic data. Cuando recibe una medición, verifica que provenga de un sensor válido en la base de datos. Si se cumpla esta condición se procede a impactar en la colección Readings en MongoDB. El servicio fue escrito en Python. Se extrajeron las funciones principales y se las pueden ver en el código 3.20.

```
# DATABASE
def insertReading(id, value, unit):
    post = {
        "date": datetime.datetime.utcnow(),
        "tag": id,
        "val": value,
        "unit": unit
    }
    global measurements
    measurements.insert_one(post)

def isValidId(id):
    global devices
    d = devices.find_one({'tag': id})
    return (d is not None)

# MQTT
def onMessage(client, userdata, msg):
    unit = msg.topic.split("/")[1][0]
    data = msg.payload.decode().split(",")
    insertReading(data[0], data[1], unit)
```

CÓDIGO 3.20. Funciones principales del servicio PV

3.3. Servicios orientados a usuarios

3.3.1. Calibrator

El servicio Calibrator es un servidor WebSocket. Tiene la finalidad de entablar una conexión con el navegador del usuario para transmitirle mediciones en vivo. En el código 3.21 se muestra su archivo principal. Se escribió en JavaScript para la plataforma Nodejs y depende de las bibliotecas CORS, Express, MQTTjs y WS.

Inicialmente la conexión comienza con el protocolo HTTP y se realiza una actualización para pasar al protocolo WebSocket. El servidor guarda una lista de las conexiones abiertas y reenvía todos los mensajes del topic *live* a todos sus clientes. Se delega en el frontend la lógica de los datos a mostrar al usuario.

```
const mqtt = require('./services/broker');
mqtt.subscribe('live');
const express = require('express');
const cors = require('cors');
const app = express();
app.use(cors());

const server = require('http').createServer(app);

const PORT = process.env.PORT || 9999;
const WebSocket = require('ws');

const wss = new WebSocket.Server({ server });

wss.on('connection', (ws) => {
    ws.on('message', (data) => {
        wss.clients.forEach((client) => {
            client.send(data);
        });
    });
});

```

```
mqtt.on('message', (topic, payload) => {
    wss.clients.forEach(client => {
        let data = `${payload}`;
        client.send(data);
    });
});

server.listen(PORT, () => { console.log(`running on: ${PORT}`) });
```

CÓDIGO 3.21. Archivo principal del servicio Calibrator

3.3.2. Backend

El servicio Backend tiene la finalidad de proveer una *REST API* al usuario. Fue escrito en JavaScript para Nodejs y sus dependencias son Bcript, CORS, Express, JsonWebToken, Mongoose, MQTTs y Redis.

En el código 3.22 se puede observar el archivo principal de la aplicación. Quedan definidos los *endpoints* o entidades del servicio. Las entidades son auth, cmnd, devices, logs, users y readings.

```
const express = require('express');
const cors = require('cors');
const bodyParser = require('body-parser');

require('./connection/database');
require('./connection/cache');
require('./connection/broker');

const PORT = process.env.PORT || 8080;

const app = express();
app.use(cors());
app.use(bodyParser.json());
app.use('/auth',require('./routes/auth'));
app.use('/cmnd', require('./routes/cmnd'));
app.use('/devices', require('./routes/devices'));
app.use('/logs', require('./routes/log'));
app.use('/users', require('./routes/users'));
app.use('/readings', require('./routes/readings'));

app.listen(PORT, () => {
    console.log(`Server running on port ${PORT}`);
});
```

CÓDIGO 3.22. Archivo principal del servicio Backend

La entidad auth solo tiene el método *Post* en donde se le envía un usuario y contraseña. Si las credenciales enviadas son correctas, se responde con un JWT que identifica una sesión para utilizar durante la operación del cliente. La función principal se extrajo de su archivo y se muestra en el código 3.23. Se puede ver que al crear una nueva sesión, el servidor entrega el estado *Created (201)*. En simultaneo se guarda el JWT generado en la memoria de Redis con un tiempo de vida determinado. Por el contrario, se entrega el estado *Unauthorized (401)* cuando las credenciales no son válidas.

```
router.post('/', async (req, res) => {
    try {
        const body = req.body;
        let user = await User.findOne({ name: body.name });
        if (user) {
            if(bcrypt.compareSync(body.password,user.password)) {
                let payload = { subject: user._id };
                let token = jwt.sign(payload, SECRET_KEY);
                cache.SETEX(token, TIME_TO_LIVE, user.rank);
                res.status(201).send({
                    user: user.name,
                    rank: user.rank,
                    token: token
                });
            } else {
                res.sendStatus(401);
            }
        } else {
            res.sendStatus(401);
        }
    } catch (error) {
        console.log(error);
        res.sendStatus(500);
    }
});
```

CÓDIGO 3.23. Función principal de la entidad auth

La entidad cmnd tiene la misión de gestionar las órdenes que se envían a los sensores. Solo se usaron métodos *GET* ya que no fue necesario enviar un cuerpo en el pedido. Como se puede ver en las funciones principales mostradas en el código 3.24, se puede ordenar que un sensor ingrese al modo calibración o que regrese al modo normal de operación. Vale la pena mencionar que las funciones presentan el uso de *middlewares* que tienen como función verificar que el cliente tenga una sesión válida y persistir la operación en un log.

```
router.get('/reset/:tag',
    middleware.verifyToken,
    middleware.verifyRankEngineer,
    middleware.logRequest,
    (req, res) => {
        try {
            mqtt.publish(`cmnd/${req.params.tag}/reset`, "reset");
            res.sendStatus(200);
        } catch (error) {
            res.sendStatus(500);
        }
    });

router.get('/calibrate/:tag',
    middleware.verifyToken,
    middleware.verifyRankEngineer,
    middleware.logRequest,
    (req, res) => {
        try {
            mqtt.publish(`cmnd/${req.params.tag}/calibrate`, "live");
            res.sendStatus(200);
        } catch (error) {
            res.sendStatus(500);
        }
    });
```

CÓDIGO 3.24. Funciones principales de la entidad cmnd

La entidad devices es la más extensa del servicio. Tiene entre sus funciones crear, leer, modificar y borrar los dispositivos de la base de datos. Cada una de estas funciones existe con múltiples variantes debido a que se realizan para un solo sensor o varios en simultaneo. En el código 3.25 se muestra un método de creación de dispositivo que es representativa para cada una de las funciones dentro del archivo. La validación de la sesión del cliente queda delegada al *middleware*. Se puede ver que también se realizan comunicaciones con los sensores cuando se realiza un cambio en su configuración. Esto crea consistencia entre lo que figura en la base de datos y lo que realmente sucede en campo.

```
router.post('',
    middleware.verifyToken,
    middleware.verifyRankEngineer,
    middleware.logRequest,
    async (req, res) => {
        try {
            let body = req.body;
            let dupplicated = await Device.findOne(
                { $or: [{ serial: body.serial }, { tag: body.tag }] },
                {}
            );
            if (dupplicated) {
                res.sendStatus(403);
            } else {
                let device = new Device({
                    serial: body.serial,
                    tag: body.tag,
                    modbus: body.modbus,
                    frec: body.frec,
                    unit: body.unit
                });
                await device.save();
                mqtt.publish(`cmnd/${device.tag}/frec`, `${device.frec
    }`);
                res.sendStatus(201);
            }
        } catch (error) {
            console.log(error);
            res.sendStatus(500);
        }
    });
```

CÓDIGO 3.25. Creación de dispositivo de la entidad devices

La entidad logs cumple la función de persistir todas las actividades que el cliente realiza en el sistema. En particular aquellas que modifican el funcionamiento de los equipos en campo, los reportes a EBI y los permisos de usuarios. Tiene la finalidad de permitir auditar los eventos ocurridos. La función principal de la entidad puede ser vista en el código 3.27 y como en el resto de los *endpoints*, la validación de sesión se manejó a través de un *middleware*.

```
router.get('',
    middleware.verifyToken,
    middleware.verifyRankAdministrator,
    async (req, res) => {
        try {
            let logs = await Log.find();
            res.status(200).send({ logs });
        } catch (error) {
            res.sendStatus(500);
        }
    });
```

CÓDIGO 3.26. Función principal de la entidad logs

En la entidad users se definen los usuarios del sistema que persisten en la base de datos. Uno de los requerimientos cumplidos (sección 1.4) fue persistir las contraseñas de forma encriptada. El archivo del *endpoint* es particularmente extenso así que se extrajo la función que crea un nuevo usuario y se la muestra en el código **??**. Se puede ver como se genera una contraseña encriptada y como se verifican la sesión y jerarquía utilizando un *middleware*.

```
router.post('/new',
    middleware.verifyToken,
    middleware.verifyRankAdministrator,
    middleware.logRequest,
    async (req, res) => {
        try {
            const body = req.body;
            let dupplicated = await User.findOne(
                { $or: [{ name: body.name }, { email: body.email }] },
                {}
            );
            if (dupplicated) {
                res.sendStatus(403)
                return;
            } else {
                const SALT = bcrypt.genSaltSync(SALT_ROUNDS);
                const HASH = bcrypt.hashSync(body.password, SALT);
                let user = new User({
                    name: body.name,
                    email: body.email,
                    password: HASH,
                    rank: body.rank
                });
                await user.save();
                res.sendStatus(201);
            }
        } catch (error) {
            console.log(error);
            res.sendStatus(500);
        }
    });
```

CÓDIGO 3.27. Creación de usuario de la entidad users

La entidad readings se encarga de leer las mediciones que se encuentran en MongoDB y las reporta al cliente. Solo tiene la función *Get* ya que la información no es modificable y las nuevas mediciones son generadas por los dispositivos. En el código 3.28 se muestra la función más utilizada por el cliente, obtener todas las mediciones de un sensor.

```
router.get('/all/:device',
    middleware.verifyToken,
    middleware.verifyRankAssistant,
    async (req, res) => {
        try {
            let readings = await Measurement.find(
                { tag: req.params.device },
                { _id: 0, __v: 0 }
            );
            res.status(200).send({ readings });
        } catch (error) {
            console.log(error);
            res.sendStatus(500);
        }
    });
```

CÓDIGO 3.28. Función más utilizada de la entidad readings

Como se pudo ver en las entidades, el manejo de sesión y permisos fueron delegados a un *middleware*. El servicio de *middleware* se compone de una serie de funciones que interceptan el mensaje del cliente y se ejecutan antes de llegar a la entidad deseada. Se generaron tres responsabilidades para estas funciones. La primera es verificar que el token que viene en el mensaje sea válido y se encuentre en Redis. La segunda es que la jerarquía del usuario sea suficiente para realizar la acción solicitada. Finalmente la tercera se encarga de persistir la transacción para futuras auditorías.

La verificación del token se puede ver en el código 3.29 y se puede apreciar el orden lógico del control. Primero se verifica que exista un token en el mensaje del usuario y luego se controla que el formato sea correcto. Luego se verifica que el token se encuentre activo dentro de Redis y que no pertenezca a un usuario que se encuentre desactivado. Si estas condiciones se cumplen, se procede a modificar el mensaje original del cliente. Se genera un campo *rank* en donde se coloca el nivel de permisos que se le otorga al pedido del usuario. Finalmente se actualiza el tiempo de vida del token y se pasa el mensaje a la próxima función. Cualquier conflicto que surja en cada etapa interrumpe la petición y devuelve el código *Unauthorized (401)*.

```
middleware.verifyToken = (req, res, next) => {
    try {
        if (!req.headers.authorization) {
            return res.sendStatus(401);
        }
        let token = req.headers.authorization.split(' ')[1];
        if (token === 'null') {
            return res.sendStatus(401);
        }
        cache.GET(token, (error, reply) => {
            if (error) {
                return res.sendStatus(401);
            }
            if (!reply) {
                return res.sendStatus(401);
            }
            if (reply == UNAUTHORIZED) {
                return res.sendStatus(401);
            }
            let payload = jwt.verify(token, SECRET_KEY);
            if (!payload) {
                return res.sendStatus(401);
```

```
            }
            req.userId = payload.subject
            req.rank = reply;
            cache.EXPIRE(token, TIME_TO_LIVE);
            next();
        });
    } catch (error) {
        return res.sendStatus(401);
    }
}
```

CÓDIGO 3.29. Verificación de token

Antes de procesar el pedido del usuario dentro de la entidad deseada, se verifica que la jerarquía sea la suficiente para realizar la operación. En el código 3.30 se muestra la función que controla que el usuario tenga permisos de ingeniero o superiores. Si cumple la condición se avanza a la siguiente función, de lo contrario se interrumpe el pedido del cliente y se le devuelve el estado 401. Las funciones para los otros niveles de usuario son prácticamente iguales a la función de verificación de ingeniero.

```
middleware.verifyRankEngineer = (req, res, next) => {
    if (req.rank >= ENGINEER) {
        next();
    } else {
        return res.sendStatus(401);
    }
}
```

CÓDIGO 3.30. Verificación de rango

La última etapa del *middleware* se utiliza solo en las acciones a ser auditadas. En esta función se guarda en la base de datos el pedido del cliente con el agregado de información adicional para facilitar la tarea del auditor. El método se puede ver en el código 3.31.

```
middleware.logRequest = async (req, res, next) => {
    try {
        let body = JSON.stringify(req.body);
        let log = new Log({
            timestamp: new Date(),
            method: req.method,
            endpoint: req.originalUrl,
            user: req.userId,
            body: body
        });
        await log.save();
        next();
    } catch (error) {
        console.log(error);
        return res.sendStatus(500);
    }
}
```

CÓDIGO 3.31. Persistencia de la operación

3.3.3. Frontend

Capítulo 4

Ensayos y resultados

Párrafo introductorio del capítulo.

4.1. Pruebas unitarias

4.2. Simulaciones

4.3. Guiones y comandos

4.4. Pruebas del cliente

Capítulo 5

Conclusiones

Este capítulo trata sobre el valor agregado que se le dio al cliente, el aprendizaje adquirido y los siguientes pasos a seguir.

5.1. Resultados obtenidos

El trabajo logró cumplir con los requerimientos que solicitó el cliente, y fueron listados en el capítulo 1. Esta situación sirvió para entablar una relación positiva con el departamento de ingeniería de Gador, ya que el cliente manifestó su conformidad con los resultados obtenidos.

La planificación original del trabajo se pudo cumplir pero solo incrementado la cantidad de horas dedicadas. El principal motivo de retraso que demandó una mayor dedicación horaria fue la poca información sobre el sistema propietario en planta. Además, durante la cursada de la especialización se vieron temas de testeo de software que hicieron visibles ciertas fallas del trabajo. Se dedicó un gran esfuerzo para depurar el código y lograr así una calidad de producción.

La imposibilidad de realizar pruebas dentro de la infraestructura de Gador fue un riesgo que lamentablemente se manifestó. Solo pudo ser sorteado utilizando una máquina virtual con una licencia de uso único de seis horas de duración para verificar la comunicación. El riesgo que por fortuna no se hizo realidad fue que alguna de las personas necesarias para realizar el trabajo se enfermara de *covid-19*, o que se tomaran decisiones de prevención que afectaran la normalidad del desarrollo.

Las técnicas que mejor resultado dieron durante la creación del trabajo fue la automatización y despliegue de contenedores usando *Docker* y *Docker Compose* y el desarrollo orientado a pruebas. La combinación de estos conocimientos genera software de calidad de producción que puede ser desplegado con gran facilidad en múltiples plataformas.

5.2. Trabajo futuro

La principal mejora a realizar es en la capa de negocios, que si bien cumple con los requerimientos del cliente, tendría un salto de valor incorporar *Checkmk* al sistema. Finalmente quedaría incorporar el trabajo a la infraestructura de Gador, para lograrlo se debe crear el hardware necesario. El siguiente paso natural es iniciar un proyecto para diseñar los sensores y puntos de agregación para tener una solución completa.

Bibliografía

[1] Dave Evans. «Hacia adonde se dirige la tecnología». En: *Cisco Internet Business Solutions Group (IBSG)* (2011).

[2] Gador. *Misión Gador*. https://www.gador.com.ar/corporacion/mision-gador/. Abr. de 2021. (Visitado 26-04-2021).

[3] U.S. Department of Health y Human Services. «Guidance for Industry». En: *Center for Drug Evaluation and Research (CDER)* (2003).

[4] Jean Bédard. «Temperature mapping of storage areas». En: *Technical supplement to WHO Technical Report Series, No. 961, 2011* (2014).

[5] Google. *Announcing the Material Design Award Winners for 2020*. https://material.io/blog/mda-2020-winners. Dic. de 2020. (Visitado 04-02-2021).

[6] AWS. *Bayer Crop Science Drives Innovation in Precision Agriculture Using AWS IoT*. https://aws.amazon.com/es/solutions/case-studies/bayer-cropscience/?c=i&sec=cs3. Mar. de 2021. (Visitado 06-03-2021).

[7] mongoDB. *Sharding*. https://docs.mongodb.com/manual/sharding. Mar. de 2021. (Visitado 06-03-2021).

[8] docker docs. *Overview fo Docker Compose*. https://docs.docker.com/compose/. Mar. de 2021. (Visitado 07-03-2021).

[9] Docker. *What is a container*. https://www.docker.com/resources/what-container. Mar. de 2021. (Visitado 07-03-2021).

[10] mongoDB. *La base de datos líder para aplicaciones modernas*. https://www.mongodb.com/es. Mar. de 2021. (Visitado 07-03-2021).

[11] DB-Engines. *DB-Engines Ranking*. https://db-engines.com/en/ranking. Mar. de 2021. (Visitado 07-03-2021).

[12] Paul Serby. *How and why to build a consumer app with Node.js*. https://venturebeat.com/2012/01/07/building-consumer-apps-with-node/. Ene. de 2012. (Visitado 08-03-2021).

[13] Michael Oberdorf. *oitc/modbus-server*. https://hub.docker.com/r/oitc/modbus-server. Mar. de 2021. (Visitado 08-03-2021).

[14] the Docker Community. *redis*. Disponible: 2021-03-13. URL: https://hub.docker.com/_/redis.