



**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, en la que se adaptó automatización de edificios y gestión empresarial marca Honeywell. La finalidad es integrar una red de sensores que utilizan un protocolo de comunicación que el sistema de Honeywell 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

Resumen	I
1. Introducción general	1
1.1. Motivación	1
1.2. Modelo de capas de la Internet de las Cosas	3
1.2.1. Capa de negocios	3
1.2.2. Capa de aplicación	4
1.2.3. Capa de procesamiento	5
1.2.4. Capa de red	6
1.2.5. Capa de percepción	7
1.2.6. Despliegue de una solución	7
1.3. Estado del arte	7
1.4. Objetivos y alcance	9
2. Introducción específica	11
2.1. Tecnologías utilizadas	11
2.1.1. Orquestación	11
2.1.2. Tecnologías de red	12
2.1.3. Bases de datos	13
2.2. Bibliotecas y paquetes de terceros	14
2.2.1. Bibliotecas y paquetes web	14
2.2.2. Bibliotecas y paquetes de transporte	14
2.2.3. Bibliotecas de seguridad y pruebas	15
2.3. Sistema propietario del cliente	16
3. Diseño e implementación	19
3.1. Arquitectura y orquestación	19
3.1.1. Servidor Modbus	20
3.1.2. Broker Mosquitto	20
3.1.3. Validación de registros Modbus	21
3.1.4. MongoDB	22
3.1.5. Validación de mediciones	23
3.1.6. Backend	23
3.1.7. Servicio de calibración	24
3.1.8. Redis	24
3.1.9. Frontend	25
3.1.10. Despliegue automático	25
3.2. Servicios orientados a dispositivos	26
3.2.1. Broker Mosquitto	26
3.2.2. Validación de registros Modbus	27
3.2.3. Validación de mediciones	27
3.3. Servicios orientados a usuarios	28
3.3.1. Calibrator	28

3.3.2. Backend	29
3.3.3. Frontend	34
4. Ensayos y resultados	41
4.1. Recursos utilizados	41
4.2. Guiones y comandos	42
4.2.1. Base de datos	42
4.2.2. Mosquitto	42
4.2.3. Redis	43
4.2.4. Nodejs	44
4.3. Pruebas unitarias	45
4.4. Simulaciones	46
4.5. Pruebas del cliente	48
5. Conclusiones	49
5.1. Resultados obtenidos	49
5.2. Trabajo futuro	49
Bibliografía	51

Índice de figuras

1.1. Red industrial Gador.	2
1.2. Software <i>Checkmk</i> , ejemplo de capa de negocio. [5]	3
1.3. Ejemplo de interfaz de diseño material. [6]	4
1.4. Ejemplo de comunicación MQTT.	6
1.5. Arquitectura de datos de alta disponibilidad.	8
1.6. Red industrial Gador.	9
2.1. Arquitectura de Docker. [10]	12
2.2. Arquitectura de Nodejs. [11]	13
2.3. Control de temperatura de sólidos.	17
2.4. Unidad de tratamiento de aire de sólidos.	17
3.1. Esquema de conexión de los servicios.	19
3.2. Lista de dispositivos.	39
3.3. Lecturas del sensor.	39
3.4. Detalles del dispositivo.	40
4.1. Pedido de credenciales con Postman.	46
4.2. Respuesta de credenciales con Postman.	46
4.3. Pedido de dispositivos con Postman.	47
4.4. Respuesta de dispositivos con Postman.	47

Índice de tablas

1.1. Modelo de capas IoT.	3
2.1. Dependencias del trabajo.	16

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 los objetos (denominados en la jerga 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.

En Argentina muchas empresas están retrasadas en su progreso tecnológico y muchas no incorporaron sistemas electrónicos en sus procesos o productos. Por lo tanto, es necesario crear un sistema que logre adaptar la tecnología en uso con el fin de integrarlas a las nuevas prácticas de negocios.

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.

En este contexto, con el objetivo de mostrar un caso de éxito en la adecuación tecnológica, 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 debe satisfacer los requerimientos del *Código federal de regulaciones - Título 21 - Capítulo de alimentos y drogas (21CFR11)* [3]. La norma establece que los registros de las mediciones ambientales de los depósitos y cuartos productivos se deben almacenar de forma electrónica pero, a su vez, se debe demostrar que los registros tienen la misma validez y seguridad que aquellos hechos en papel. 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 y en uso 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, no es económicamente viable reemplazarlo por un producto moderno que esté aprobado por la FDA. En consecuencia, 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 estudios periódicos 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. El modelo lógico de la arquitectura se puede visualizar en la figura 1.1, donde se puede ver una estructura del tipo árbol donde la información es suministrada al sistema EBI.

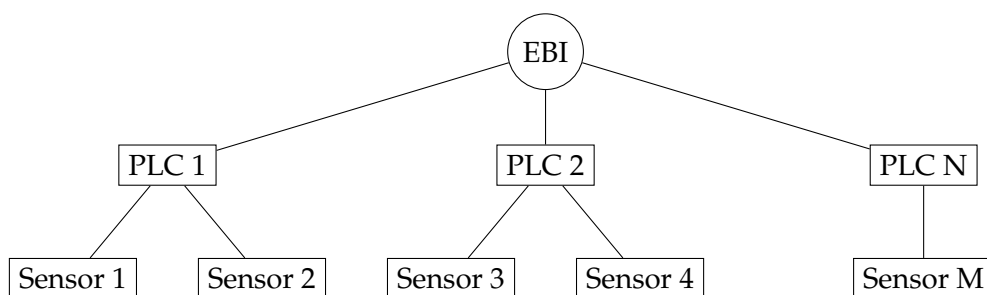


FIGURA 1.1. Red industrial Gador.

1.2. Modelo de capas de la Internet de las Cosas

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 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 de capas seleccionado separa los conocimientos en cinco categorías, las cuales son las capas de negocio, aplicación, procesamiento, red y percepción. En la tabla 1.1 se presenta un resumen de las funciones de cada capa.

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

1.2.1. Capa de negocios

La capa de negocio suele tener una funcionalidad contable. Esta capa puede ser la encargada de determinar y generar la facturación para cobrarle a los usuarios de la aplicación, como así también, 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. Un ejemplo de capa de negocio se puede observar en la figura 1.2.

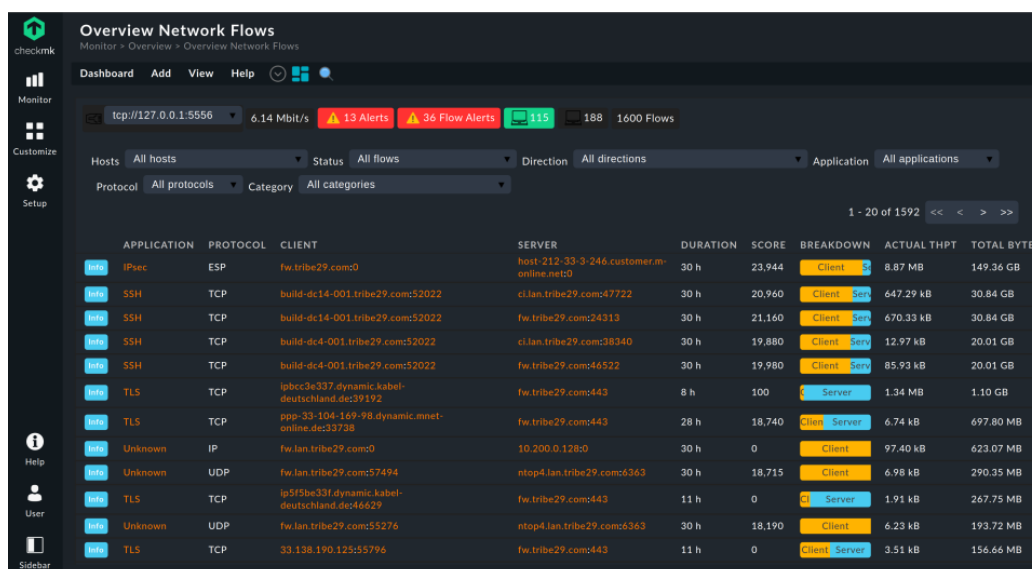


FIGURA 1.2. Software Checkmk, ejemplo de capa de negocio. [5]

1.2.2. Capa de aplicación

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 y 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 1.3.

Para llevar adelante 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). De esta manera se logra que el servidor otorgue todo el código necesario para que el dispositivo del usuario genere por sí mismo los componentes gráficos a mostrar. Es importante que el código entregado pueda ser visualizado en múltiples tamaños de pantallas, ya que 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 la define como *responsiva*.

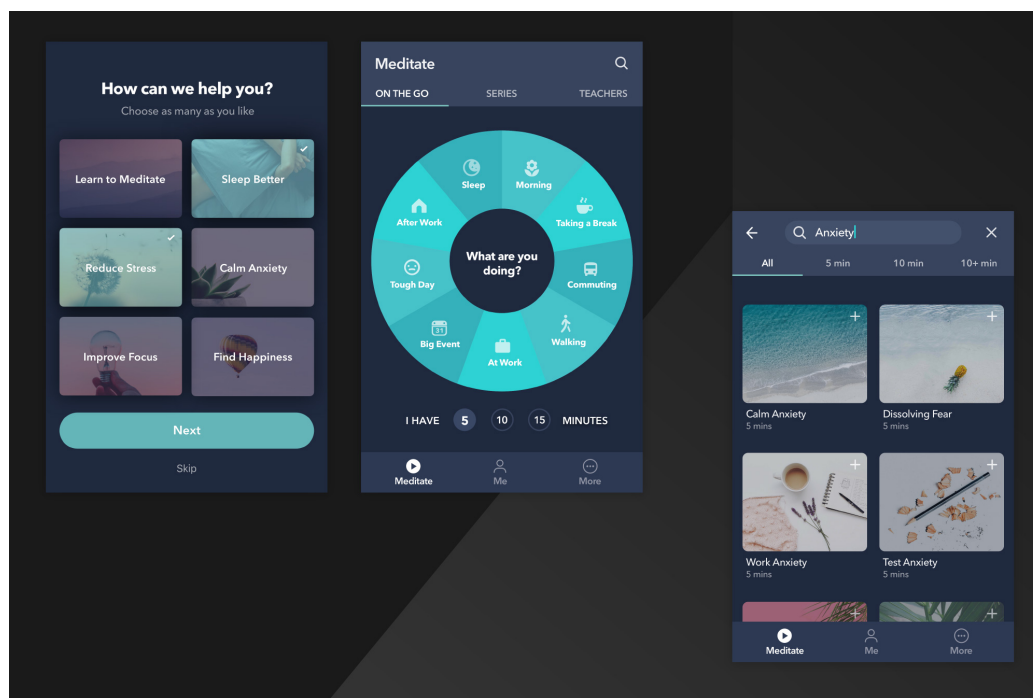


FIGURA 1.3. Ejemplo de interfaz de diseño material. [6]

1.2.3. Capa de procesamiento

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. Una de las principales es la tecnología de 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 contiene a los siguientes tipos de bases de datos:

- Clave-valor
- Orientada a documentos
- Orientada a columnas
- Orientada a 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 formatos de codificación estándar, los cuales son:

- XML
- YAML
- JSON
- BSON

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. Además 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.

1.2.4. Capa de red

Se dispone de un repertorio de protocolos pertenecientes a la capa de red para lograr que los dispositivos se comuniquen entre sí. Entre los mencionados a lo largo de esta memoria se encuentran 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 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. 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 no es 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 encuentren suscritos, como se puede ver en el ejemplo de la figura 1.4, donde el cliente 2 no obtiene los datos del sensor porque no se encuentra suscrito.

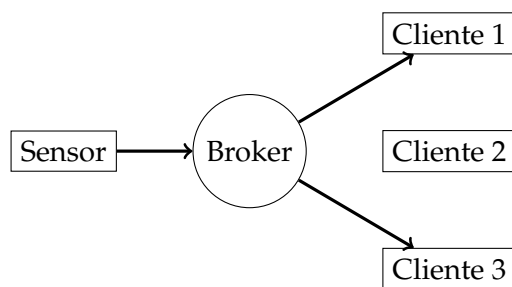


FIGURA 1.4. Ejemplo de comunicación MQTT.

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.

WebSocket es un protocolo similar a HTTP pero con la diferencia que la conexión es bidireccional. Esto quiere decir que cuando se logra la conexión 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*.

1.2.5. Capa de percepción

Los sensores utilizados en una solución de IoT están incluidos en la capa de percepción. 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 en este tema.

1.2.6. Despliegue de una solución

Teniendo definidos 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 utilizan corren un firmware específico. Por ejemplo, si se decidió utilizar AWS 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 [7].

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 suele transmitir 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, que suelen conformar una réplica donde solo se almacenan los datos de configuración. Esta arquitectura de alta disponibilidad se conoce como granja de datos, se puede construir con mongoDB [8] y se encuentra representada en la figura 1.5.

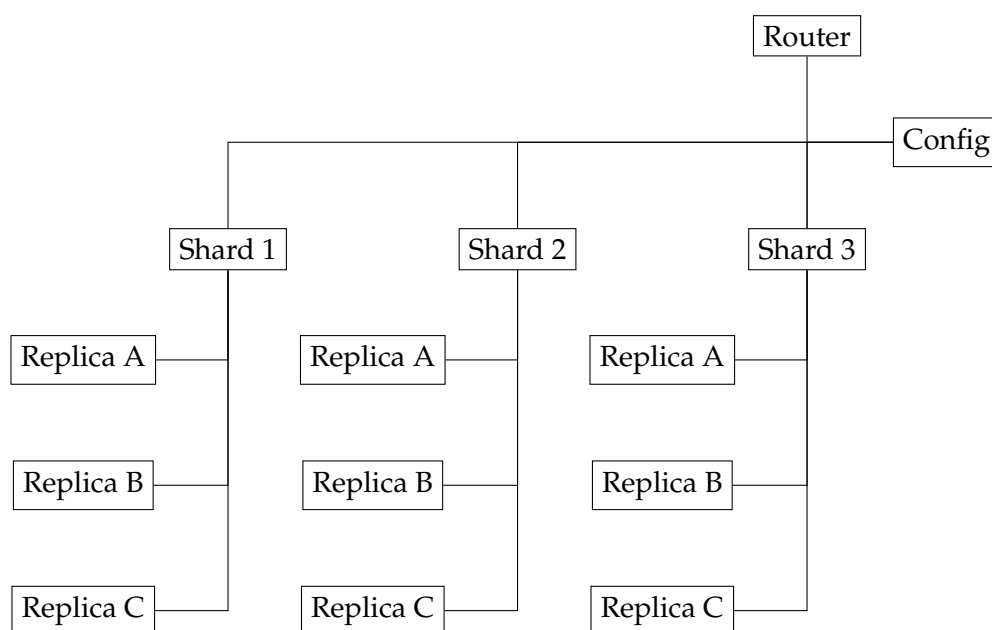


FIGURA 1.5. Arquitectura de datos de alta disponibilidad.

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 trabajo 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 asentar los conocimientos a través de la práctica.

El trabajo se limitó a desarrollar el software a desplegar en un servidor. Esto significa que no se contempló el desarrollo del hardware, en particular los sensores y los puntos de agregación. El esquema del servidor en la red de Gador puede ser visualizado en la figura 1.6, donde el servidor denominado Nodos corresponde a la solución desarrollada. 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.

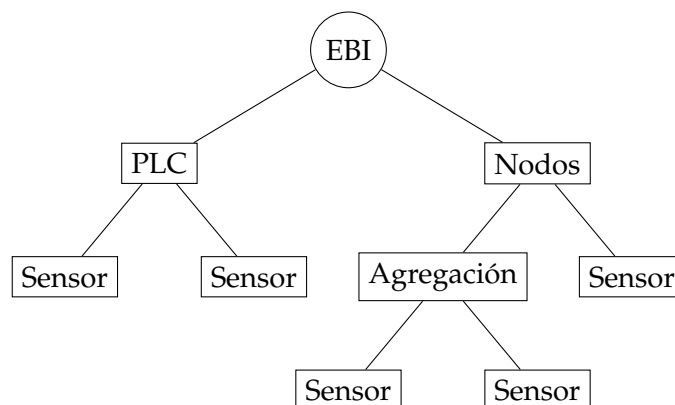


FIGURA 1.6. Red industrial Gador.

Una vez definidos los objetivos y el alcance del proyecto, se inició un proceso de negociación con el cliente. Se buscó determinar cuáles 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

2.1.1. Orquestación

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 cumplen la función de ser 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 sí.

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. Es decir, 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 Dockerfile, 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 descargada directamente sobre el entorno de producción. De esta manera se facilita el despliegue de la aplicación, por lo que sólo resta orquestar los contenedores para que trabajen en conjunto.

La orquestación necesaria para la etapa de despliegue se logra utilizando Docker Compose. Según se indica en su documentación [9], 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 tener orquestada la solución.

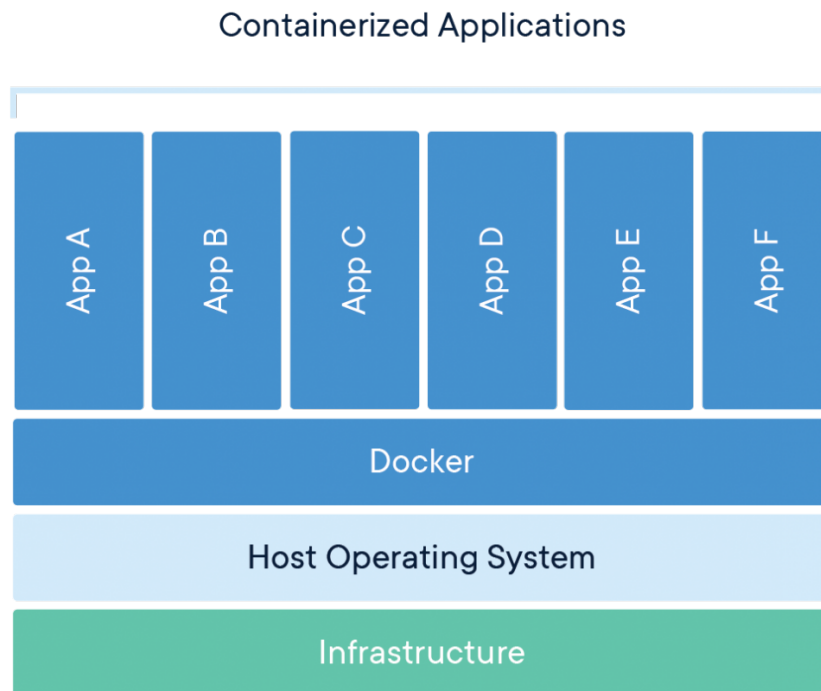


FIGURA 2.1. Arquitectura de Docker. [10]

2.1.2. Tecnologías de red

Este trabajo utiliza tecnologías web y la plataforma utilizada para implementarlas fue Nodejs, que es un servidor asincrónico y orientado a eventos que ejecuta JavaScript. Su arquitectura se puede visualizar en la figura 2.2. 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 Nodejs 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

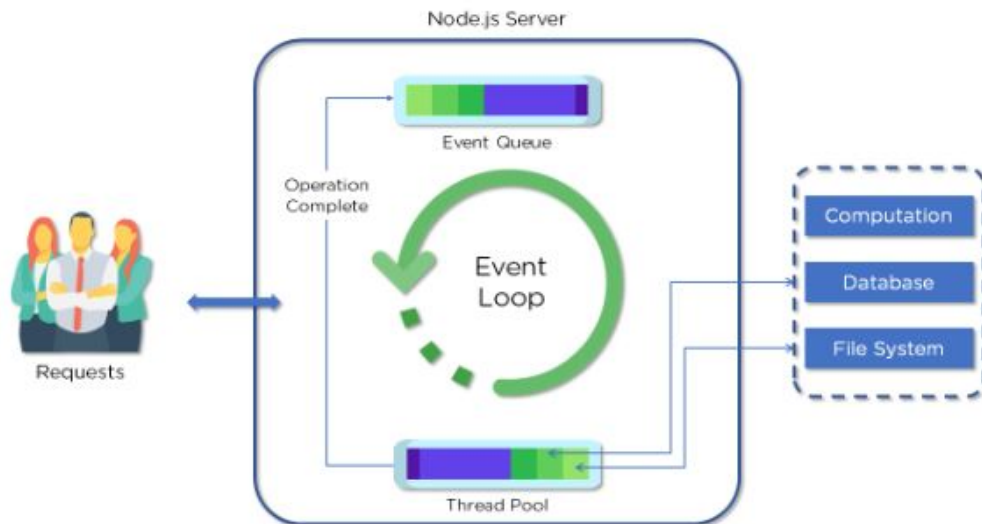


FIGURA 2.2. Arquitectura de Nodejs. [11]

profundidad en la cursada, fue suficiente para entender su potencial, ya que muchas de sus bibliotecas fueron útiles para desarrollar algunos de los servicios más pequeños del trabajo, 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. Además 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 suscripciones para correr pruebas y además se pueden crear contraseñas encriptadas para los usuarios.

2.1.3. Bases de datos

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"[12]. 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 [13]. Por lo tanto se evaluó como aspecto relevante ganar experiencia con esta tecnología.

Los contenedores que forman la aplicación deben compartir una memoria volátil para que el sistema 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

2.2.1. Bibliotecas y paquetes web

Para implementar la capa de aplicación se utilizó un framework de diseño de interfaces gráficas llamado Angular y se creó una SWA. Se desarrolló 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. Además 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. [14]

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.

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.

2.2.2. Bibliotecas y paquetes de transporte

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 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 [15]. El sistema de orquestación corrige el puerto al conectar el 5020 del contenedor con el 502 del ordenador.

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.

2.2.3. Bibliotecas de seguridad y pruebas

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 biblioteca utilizada para cumplir con los requerimientos de seguridad fue Jsonwebtoken. 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.

Todos los recursos externos del trabajo se pueden visualizar en la tabla 2.1.

TABLA 2.1. Dependencias del trabajo.

Recurso externo	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
Bcrypt	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 tiene como objetivo integrarse al sistema que Gador tiene en ejecución. Esta sección explica con mayor detalle dicho sistema.

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. Su objetivo es 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 2.3, se utiliza el software para visualizar los sensores de los distintos cuartos de producción y depósitos. En la figura 2.4 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.

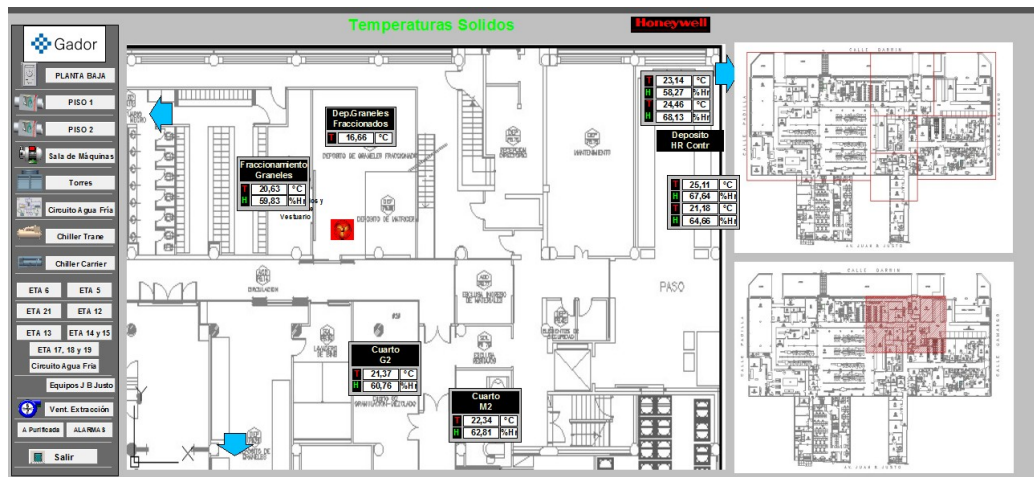


FIGURA 2.3. Control de temperatura de sólidos.

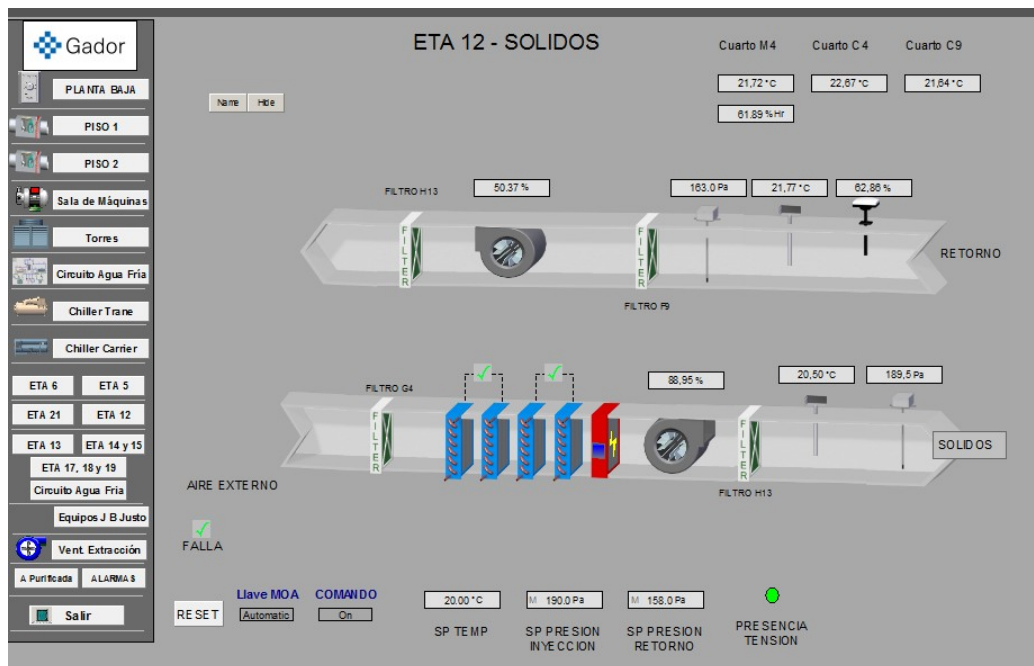


FIGURA 2.4. Unidad de tratamiento de aire de sólidos.

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. Las entidades externas que interactúan con el servidor Nodos se encuentran destacadas en color rojo. 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.

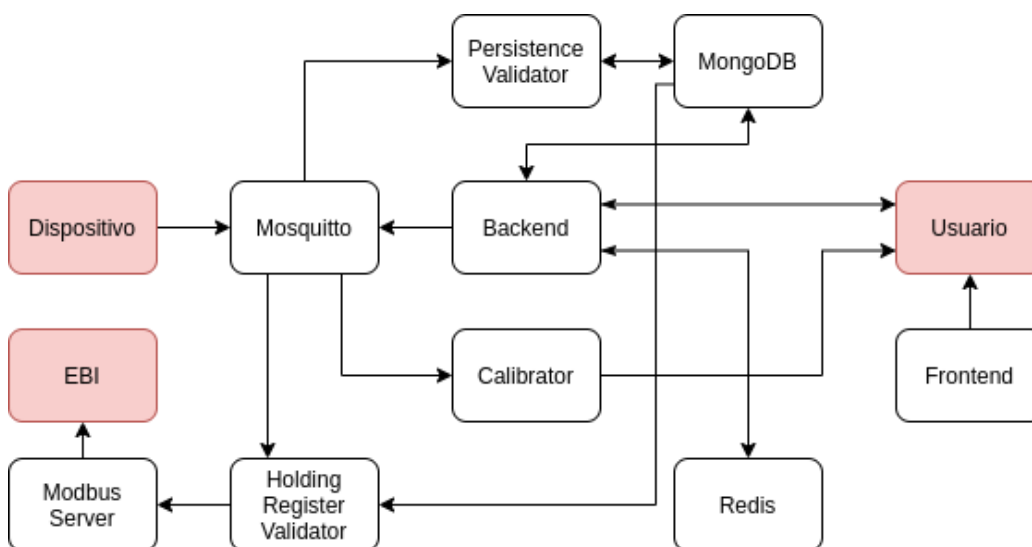


FIGURA 3.1. Esquema de conexión de los servicios.

```

1 networks:
2   iot:
3     driver: bridge

```

CÓDIGO 3.1. Red de interconexión Docker Compose.

3.1.1. Servidor Modbus

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.

```
1 modbus-server:
2   image: oitc/modbus-server
3   container_name: modbus-server
4   hostname: modbus-server
5   restart: always
6   ports:
7     - '502:5020'
8   expose:
9     - '5020'
10  networks:
11    - iot
```

CÓDIGO 3.2. Orquestación del servidor Modbus.

3.1.2. Broker Mosquitto

El servicio Mosquitto se configuró con tres volúmenes que conectan al contenedor con archivos no efímeros que persisten la información. De esta manera se asegura 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 se puedan utilizar los *topics* designados. Adicionalmente, los distintos usuarios tienen diferentes permisos según el *topic*. Así se logra una mayor confiabilidad y seguridad en el manejo de los mensajes.

```
1 mosquitto:
2   image: eclipse-mosquitto
3   container_name: mosquitto
4   hostname: mosquitto
5   restart: always
6   volumes:
7     - ./mosquitto/mosquitto.conf:/mosquitto/config/mosquitto.conf
8     - ./mosquitto/users.txt:/mosquitto/config/users.txt
9     - ./mosquitto/acl.txt:/mosquitto/config/acl.txt
10  expose:
11    - '1883'
12    - '9001'
13  ports:
14    - '1883:1883'
15    - '9001:9001'
16  networks:
17    - iot
```

CÓDIGO 3.3. Orquestación del broker Mosquitto.

3.1.3. Validación de registros Modbus

El servicio Holding Registers Validator (hrv) tiene la particularidad de depender de otros servicios, como se puede ver en el código 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 de 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.

```

1 hrv:
2   build: ./holdingRegistersValidator/
3   image: vaca/hrv
4   container_name: hrv
5   hostname: hrv
6   restart: always
7   expose:
8     - '1883'
9     - '5020'
10  depends_on:
11    - 'mosquitto'
12    - 'mongo'
13    - 'modbus-server'
14  networks:
15    - 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.

```

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

```

CÓDIGO 3.5. Dockerfile del servicio hrv.

3.1.4. MongoDB

El servicio MongoDB no tiene salida al exterior del servidor y 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. Así se le indica al motor de MongoDB cual 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 utilizados para su construcción son:

- devices.js
- measurements.js
- seed.js
- 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 autenticar 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.

```
1 mongo:
2   image: mongo
3   container_name: mongo
4   hostname: mongo
5   command: mongod --bind_ip_all --port 27017
6   expose:
7     - '27017'
8   volumes:
9     - ./mongodb/scripts:/scripts
10  networks:
11    - iot
```

CÓDIGO 3.6. Orquestación de MongoDB.

```
1 use gador;
2 load("scripts/devices.js")
3 load("scripts/users.js")
4 load("scripts/measurements.js")
5 load("scripts/measurements.js")
6 load("scripts/measurements.js")
7 load("scripts/measurements.js")
8 load("scripts/measurements.js")
9 load("scripts/measurements.js")
10 load("scripts/measurements.js")
11 load("scripts/measurements.js")
12 load("scripts/measurements.js")
```

CÓDIGO 3.7. Seed de la base de datos.

3.1.5. Validación de mediciones

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

```
1  pv:
2    build: ./persistenceValidator
3    image: vaca/pv
4    container_name: pv
5    hostname: pv
6    restart: always
7    expose:
8      - '1883'
9      - '27017'
10   depends_on:
11     - 'mosquitto'
12     - 'mongo'
13   networks:
14     - iot
```

CÓDIGO 3.8. Orquestación del servicio pv.

3.1.6. Backend

Para orquestar el servicio backend, según el código 3.9, se utilizó el puerto 8080 del ordenador para permitir la comunicación con una entidad externa. 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. El Dockerfile parte de la imagen oficial de Nodejs y copia los archivos de dependencias dentro del contenedor auxiliar. Con este contenedor 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.

```
1  backend:
2    build: ./backend
3    image: vaca/backend
4    container_name: backend
5    hostname: backend
6    expose:
7      - '1883'
8      - '6379'
9      - '27017'
10   ports:
11     - '8080:8080'
12   depends_on:
13     - 'mosquitto'
14     - 'mongo'
15     - 'redis'
16   networks:
17     - iot
```

CÓDIGO 3.9. Orquestación del servicio Backend.

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

CÓDIGO 3.10. Dockerfile del servicio Backend.

3.1.7. Servicio de calibración

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.

```
1 calibrator:
2   build: ./calibrator
3   image: vaca/calibrator
4   container_name: calibrator
5   hostname: calibrator
6   expose:
7     - '1883'
8     - '9999'
9   ports:
10    - '9999:9999'
11   depends_on:
12     - 'mosquitto'
13   networks:
14     - iot
```

CÓDIGO 3.11. Orquestación del servicio Calibrator.

3.1.8. Redis

El servicio Redis fue creado a partir de la imagen oficial de Redis que se encuentra disponible en Dockerhub [16]. 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.

```
1 redis:
2   image: redis
3   container_name: redis
4   hostname: redis
5   expose:
6     - '6379'
7   networks:
8     - iot
```

CÓDIGO 3.12. Orquestación del servicio Redis.

3.1.9. Frontend

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.

```
1 frontend:
2   build: ./frontend/
3   image: vaca/frontend
4   container_name: frontend
5   hostname: frontend
6   restart: always
7   ports:
8     - '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.

```
1 FROM node as builder
2 WORKDIR /src/app
3 COPY . ./
4 RUN npm install
5 RUN npm run ng build --prod
6 FROM nginx
7 COPY --from=builder /src/app/dist/frontend /usr/share/nginx/html
```

CÓDIGO 3.14. Dockerfile del servicio Frontend.

3.1.10. Despliegue automático

Con los Dockerfiles definidos 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.

```
1 #!/bin/bash
2 chmod +x clean.sh
3 docker-compose up -d
4 sleep 10
5 docker-compose exec mongo sh -c "mongo < /scripts/seed.js"
6 docker-compose exec mongo sh -c "mongo < /scripts/seed.test.js"
```

CÓDIGO 3.15. Guión de inicialización.

```
1 #!/bin/bash
2 docker-compose down
3 docker rmi vaca/backend
4 docker rmi vaca/hrv
5 docker rmi vaca/pv
6 docker rmi vaca/auth-api
7 docker rmi vaca/calibrator
8 docker rmi vaca/frontend
9 clear
```

CÓDIGO 3.16. Guión de limpieza.

3.2. Servicios orientados a dispositivos

3.2.1. Broker Mosquitto

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. Por esta razón 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.

```
1 allow_anonymous false
2 password_file /mosquitto/config/users.txt
3 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 se usó con los contenedores. El segundo usuario se nombró device y se empleó para identificar a los dispositivos. 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 *cmd* 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.

```

1 user docker
2 topic readwrite cmnd/#
3 topic read data/#
4
5 user device
6 topic readwrite cmnd/#
7 topic write data/#
8 topic write live/#

```

CÓDIGO 3.18. Lista de control de acceso

3.2.2. Validación de registros Modbus

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 suscribe 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.

```

1 # MQTT
2 def onMessage(client, userdata, msg):
3     data = msg.payload.decode().split(",")
4     addr = getAddr(data[0])
5     if addr != -1:
6         val = int(data[1])
7         write_slave(addr, val)
8
9 # DATABASE
10 def getAddr(id):
11     global devices
12     d = devices.find_one({"tag": id})
13     if d is None:
14         return -1
15     if 'modbus' in d:
16         return int(d['modbus'])
17     return -1
18
19 # MODBUS
20 def write_slave(addr, value):
21     global master
22     if master.write_single_register(addr, value):
23         print('writing successful')
24     else:
25         print('writing error')

```

CÓDIGO 3.19. Funciones principales del servicio HRV

3.2.3. Validación de mediciones

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 suscrito al topic data. Cuando recibe una medición verifica que provenga de un sensor válido en la base de datos. Si se cumple esta condición se procede a impactar en la colección Readings en MongoDB. El servicio fue escrito en Python y se extrajeron las funciones principales, se las pueden ver en el código 3.20.

```

1 # DATABASE
2 def insertReading(id, value, unit):
3     post = {
4         "date": datetime.datetime.utcnow(),
5         "tag": id,
6         "val": value,
7         "unit": unit
8     }
9     global measurements
10    measurements.insert_one(post)
11
12 def isValidId(id):
13     global devices
14     d = devices.find_one({'tag': id})
15     return (d is not None)
16
17 # MQTT
18 def onMessage(client, userdata, msg):
19     unit = msg.topic.split("/")[1][0]
20     data = msg.payload.decode().split(",")
21     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.

```

1 const mqtt = require('./services/broker');
2 mqtt.subscribe('live');
3 const express = require('express');
4 const cors = require('cors');
5 const app = express();
6 app.use(cors());
7
8 const server = require('http').createServer(app);
9
10 const PORT = process.env.PORT || 9999;
11 const WebSocket = require('ws');
12
13 const wss = new WebSocket.Server({ server });
14
15 wss.on('connection', (ws) => {
16     ws.on('message', (data) => {
17         wss.clients.forEach((client) => {
18             client.send(data);
19         });
20     });
21 });
22

```



```

23 mqtt.on('message', (topic, payload) => {
24     wss.clients.forEach(client => {
25         let data = `${payload}`;
26         client.send(data);
27     });
28 });
29
30 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 Bcrypt, 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.

```

1 const express = require('express');
2 const cors = require('cors');
3 const bodyParser = require('body-parser');
4
5 require('./connection/database');
6 require('./connection/cache');
7 require('./connection/broker');
8
9 const PORT = process.env.PORT || 8080;
10
11 const app = express();
12 app.use(cors());
13 app.use(bodyParser.json());
14 app.use('/auth', require('./routes/auth'));
15 app.use('/cmnd', require('./routes/cmnd'));
16 app.use('/devices', require('./routes/devices'));
17 app.use('/logs', require('./routes/log'));
18 app.use('/users', require('./routes/users'));
19 app.use('/readings', require('./routes/readings'));
20
21 app.listen(PORT, () => {
22     console.log('Server running on port ${PORT}');
23 });

```

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.

```

1 router.post('/', async (req, res) => {
2   try {
3     const body = req.body;
4     let user = await User.findOne({ name: body.name });
5     if (user) {
6       if (bcrypt.compareSync(body.password, user.password)) {
7         let payload = { subject: user._id };
8         let token = jwt.sign(payload, SECRET_KEY);
9         cache.SETEX(token, TIME_TO_LIVE, user.rank);
10        res.status(201).send({
11          user: user.name,
12          rank: user.rank,
13          token: token
14        });
15      } else {
16        res.sendStatus(401);
17      }
18    } else {
19      res.sendStatus(401);
20    }
21  } catch (error) {
22    console.log(error);
23    res.sendStatus(500);
24  }
25 });

```

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.

```

1 router.get('/reset/:tag',
2   middleware.verifyToken,
3   middleware.verifyRankEngineer,
4   middleware.logRequest,
5   (req, res) => {
6     try {
7       mqtt.publish('cmnd/${req.params.tag}/reset', "reset");
8       res.sendStatus(200);
9     } catch (error) {
10      res.sendStatus(500);
11    }
12  });
13
14 router.get('/calibrate/:tag',
15   middleware.verifyToken,
16   middleware.verifyRankEngineer,
17   middleware.logRequest,
18   (req, res) => {
19     try {
20       mqtt.publish('cmnd/${req.params.tag}/calibrate', "live");
21       res.sendStatus(200);
22     } catch (error) {
23       res.sendStatus(500);
24     }
25  });

```

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.

```

1 router.post('/',
2   middleware.verifyToken,
3   middleware.verifyRankEngineer,
4   middleware.logRequest,
5   async (req, res) => {
6     try {
7       let body = req.body;
8       let duplicated = await Device.findOne(
9         { $or: [{ serial: body.serial }, { tag: body.tag } ] },
10        {}
11      );
12      if (duplicated) {
13        res.sendStatus(403);
14      } else {
15        let device = new Device({
16          serial: body.serial,
17          tag: body.tag,
18          modbus: body.modbus,
19          frec: body.frec,
20          unit: body.unit
21        });
22        await device.save();
23        mqtt.publish('cmdn/${device.tag}/frec', `${device.frec}
24      });
25      res.sendStatus(201);
26    }
27    } catch (error) {
28      console.log(error);
29      res.sendStatus(500);
30    }
31  });

```

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 3.26 y como en el resto de los *endpoints*, la validación de sesión se manejó a través de un *middleware*.

```

1 router.get( '',
2   middleware.verifyToken ,
3   middleware.verifyRankAdministrator ,
4   async (req, res) => {
5     try {
6       let logs = await Log.find();
7       res.status(200).send({ logs });
8     } catch (error) {
9       res.sendStatus(500);
10    }
11  });

```

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 3.27. Se puede ver como se genera una contraseña encriptada y como se verifican la sesión y jerarquía utilizando un *middleware*.

```

1 router.post( '/new' ,
2   middleware.verifyToken ,
3   middleware.verifyRankAdministrator ,
4   middleware.logRequest ,
5   async (req, res) => {
6     try {
7       const body = req.body;
8       let duplicated = await User.findOne(
9         { $or: [{ name: body.name }, { email: body.email }] },
10        {}
11      );
12      if (duplicated) {
13        res.sendStatus(403)
14        return;
15      } else {
16        const SALT = bcrypt.genSaltSync(SALT_ROUNDS);
17        const HASH = bcrypt.hashSync(body.password, SALT);
18        let user = new User({
19          name: body.name,
20          email: body.email,
21          password: HASH,
22          rank: body.rank
23        });
24        await user.save();
25        res.sendStatus(201);
26      }
27    } catch (error) {
28      console.log(error);
29      res.sendStatus(500);
30    }
31  });

```

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 que es obtener todas las mediciones de un sensor.

```

1 router.get('/all/:device',
2   middleware.verifyToken,
3   middleware.verifyRankAssistant,
4   async (req, res) => {
5     try {
6       let readings = await Measurement.find(
7         { tag: req.params.device },
8         { _id: 0, __v: 0 }
9       );
10      res.status(200).send({ readings });
11    } catch (error) {
12      console.log(error);
13      res.sendStatus(500);
14    }
15  });

```

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).

```

1 middleware.verifyToken = (req, res, next) => {
2   try {
3     if (!req.headers.authorization) {
4       return res.sendStatus(401);
5     }
6     let token = req.headers.authorization.split(' ')[1];
7     if (token === 'null') {
8       return res.sendStatus(401);
9     }
10    cache.GET(token, (error, reply) => {
11      if (error) {
12        return res.sendStatus(401);
13      }
14      if (!reply) {
15        return res.sendStatus(401);
16      }
17      if (reply == UNAUTHORIZED) {
18        return res.sendStatus(401);
19      }
20      let payload = jwt.verify(token, SECRET_KEY);
21      if (!payload) {
22        return res.sendStatus(401);

```

```

23         }
24         req.userId = payload.subject
25         req.rank = reply;
26         cache.EXPIRE(token, TIME_TO_LIVE);
27         next();
28     });
29     } catch (error) {
30         return res.sendStatus(401);
31     }
32 }

```

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 o 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.

```

1 middleware.verifyRankEngineer = (req, res, next) => {
2     if (req.rank >= ENGINEER) {
3         next();
4     } else {
5         return res.sendStatus(401);
6     }
7 }

```

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.

```

1 middleware.logRequest = async (req, res, next) => {
2     try {
3         let body = JSON.stringify(req.body);
4         let log = new Log({
5             timestamp: new Date(),
6             method: req.method,
7             endpoint: req.originalUrl,
8             user: req.userId,
9             body: body
10        });
11        await log.save();
12        next();
13    } catch (error) {
14        console.log(error);
15        return res.sendStatus(500);
16    }
17 }

```

CÓDIGO 3.31. Persistencia de la operación

3.3.3. Frontend

El servicio Frontend tiene la misión de generar la interfaz gráfica que experimenta el usuario en su navegador. Este servicio fue el más extenso en la cantidad de

líneas de código y es el más sofisticado desde el punto de vista tecnológico. Es la única aplicación en donde se utilizó el lenguaje TypeScript para escribir los archivos fuente. Además se utilizó el framework Angular junto con la biblioteca de componentes Angular Material.

Se crearon una serie de servicios que tienen la finalidad de dotar a la interfaz gráfica de la capacidad de obtener la información que necesita. Particularmente, la habilidad de comunicarse con otros servicios de la solución.

El servicio AuthService se encarga de gestionar las credenciales de sesión. Para lograr este objetivo se comunica con el Backend y realiza un *Post* indicando usuario y contraseña. Cuando la respuesta es satisfactoria, se guarda el token, el usuario y su rango en la memoria local del navegador. Este servicio provee la capacidad de leer estos datos que se encuentran en el navegador para que los componentes se comporten de la forma esperada. En el código 3.32 se puede ver la función mas importante del servicio, la obtención de credenciales.

```
1 signIn(name: string , password: string): Promise<boolean> {
2   return this.http.post(
3     this.url ,
4     {
5       name: name,
6       password: password
7     },
8     {
9       headers: { 'Content-Type': 'application/json; charset=utf-8' },
10      observe: 'response'
11    })
12  .toPromise()
13  .then((res) => {
14    if (res.status == 201) {
15      let credentials = <Credentials>res.body;
16      localStorage.setItem('user', credentials.user);
17      localStorage.setItem('rank', credentials.rank.toString());
18      localStorage.setItem('token', credentials.token);
19      return true;
20    } else {
21      localStorage.clear();
22      return false;
23    }
24  })
25  .catch((error) => {
26    localStorage.clear();
27    return false;
28  })
29 }
```

CÓDIGO 3.32. Obtención de credenciales

El servicio CredentialsInterceptorService tiene la misión de interceptar todos los mensajes que emita la interfaz gráfica y agrega el token a la cabecera del mensaje HTTP. Tiene una única función y se la puede ver en el código 3.33.

```

1 intercept(req, next) {
2   let credentials = req.clone({
3     setHeaders: {
4       'Content-Type': "application/json",
5       Authorization: 'Bearer ${this.auth.getToken()}'
6     }
7   });
8   return next.handle(credentials);
9 }

```

CÓDIGO 3.33. Intercepción de mensajes

El servicio DevicesService cumple la función de unir la entidad que representa los dispositivos en el Backend. Este servicio posee múltiples funciones para satisfacer las operaciones de lectura, creación, modificación y eliminación de recursos. Por esta razón solo se muestra el método encargado de obtener la lista de dispositivos que se puede observar en el código 3.34.

```

1 getAll(): Promise<Devices> {
2   return this.http.get(this.url, { observe: 'response' }).toPromise()
3     .then(res => {
4       let devices = <Devices>res.body;
5       return devices;
6     })
7     .catch(error => {
8       this.auth.logout();
9       return <Devices>{};
10    });
11 }

```

CÓDIGO 3.34. Obtención de la lista de dispositivos

El servicio LogsService se encarga de obtener el registro de actividades para realizar auditorías. Solo es posible obtener la lista y no existen métodos para borrar, alterar ni crear logs. La función se puede ver en el código 3.35.

```

1 getAll(): Promise<Logs> {
2   return this.http.get(this.url, { observe: 'response' }).toPromise()
3     .then(res => {
4       let logs: Logs = <Logs>res.body;
5       return logs;
6     })
7     .catch(error => {
8       this.auth.logout();
9       return <Logs>{};
10    });
11 }

```

CÓDIGO 3.35. Obtención de la lista de logs

El servicio ReadingsService tiene el objetivo de interactuar con el *endpoint* readings del Backend. La única funcionalidad es la de leer las lecturas ya que estas son creadas por los dispositivos. El método más utilizado de este servicio se puede observar en el código 3.36.


```

1 getAllReadings(tag: string): Promise<Readings> {
2   return this.http.get(`${this.url}/all/${tag}`, { observe: 'response'
3     })
4     .toPromise()
5     .then(res => {
6       let readings: Readings = <Readings>res.body;
7       return readings;
8     })
9     .catch(err => {
10      console.log(err);
11      return <Readings>{}
12    })
13 }

```

CÓDIGO 3.36. Obtención de mediciones

El servicio `UserService` tiene la tarea de conectarse a la entidad `users` del Backend. Se satisfacen dos necesidades, la primera es alimentar el *pipe* encargado de mostrar los nombres de usuarios en la pantalla de logs. La segunda es que los componentes relacionados con la gestión de usuarios puedan consumir los datos que necesitan. Estas dos funcionalidades se pueden ver en el código 3.37.

```

1 getPipeData(): Promise<PipeData> {
2   return this.http.get(
3     `${this.url}/pipe-data`,
4     { observe: 'response' }
5   ).toPromise()
6   .then(res => {
7     console.log('then')
8     let pipeData: PipeData = <PipeData>res.body;
9     return pipeData;
10  })
11  .catch(error => {
12    this.auth.logout();
13    return <PipeData>{};
14  })
15 }
16
17 getAll(): Promise<Users> {
18   return this.http.get(`${this.url}/all`, { observe: 'response' }).
19     toPromise()
20     .then(res => {
21       let users: Users = <Users>res.body;
22       return users;
23     })
24     .catch(error => {
25       this.auth.logout();
26       return <Users>{};
27     })
28 }

```

CÓDIGO 3.37. Funciones de `UserService`

El último de los servicios es `WebsocketService` y fue creado para conectar el navegador del cliente con el servicio `Calibrator` del servidor `Nodos`. Se genera una conexión del tipo `WebSocket` para obtener los datos de calibración en vivo de un sensor. La función principal se puede ver en el código 3.38.

```

1 public connect(): WebSocketSubject<any> {
2   return websocket({
3     url: WS_ENDPOINT,
4     deserializer: ({data}) => data
5   });
6 }

```

CÓDIGO 3.38. Función principal de WebsocketService

Cuando el cliente no posee las credenciales necesarias o las que tiene expiraron en el Backend, se debe direccionar la interfaz gráfica a la pantalla del componente de login. Para lograr esa función se implementó una guarda de identificación y se muestra en el código 3.39.

```

1 canActivate(): boolean {
2   if (this.authService.loggedIn()) {
3     return true;
4   } else {
5     this.router.navigate(['/login']);
6     return false;
7   }
8 }

```

CÓDIGO 3.39. Función de guarda de identificación

Algunos datos llegan del Backend en un formato no amigable para el usuario. Para mejorar el aspecto de la información o para que sea interpretable por un humano, se utilizan *pipes*. En el trabajo se crearon dos *pipes* propios. El primero transforma el código de identificación de usuario (código *HASH*) y lo reemplaza por su nombre y apellido, como se puede ver en el código 3.40. El segundo cambia el código de magnitud de los dispositivos y los expresa en unidades de medición que son de uso común. La transformación se puede ver en el código 3.41.

```

1 transform(value: string, arg: PipeData): unknown {
2   let data: PipeDataRow[] = arg.pipeData;
3   let name: string = "No encontrado";
4   data.forEach(row => {
5     if(row._id == value) {
6       name = row.name;
7       return;
8     }
9   });
10  return name;
11 }

```

CÓDIGO 3.40. Función de pipe de usuario

```

1 transform(value: string): string {
2   return value[0] == 't'? "C": value[0] == 'h'? "HR" : "";
3 }

```

CÓDIGO 3.41. Función de pipe de unidad

Para unir la información proveniente del Backend con una estructura de datos de TypeScript, se crearon una serie de interfaces que cumplen la función de ser modelos de datos. En el código 3.42 se muestra la interfaz correspondiente a un dispositivo.

```

1 export interface Device {
2     serial: number,
3     tag: string,
4     modbus: number,
5     freq: number,
6     unit: string
7 }

```

CÓDIGO 3.42. Modelo de datos de dispositivo

La etapa final del desarrollo del Frontend fue crear los componentes gráficos a mostrar en pantalla. La tarea se llevó a cabo utilizando la biblioteca Angular Material y en las figuras 3.2, 3.2 y 3.2 se puede observar la apariencia lograda. En los ejemplos se aprecian formularios, tablas con paginación y listas desarrolladas.

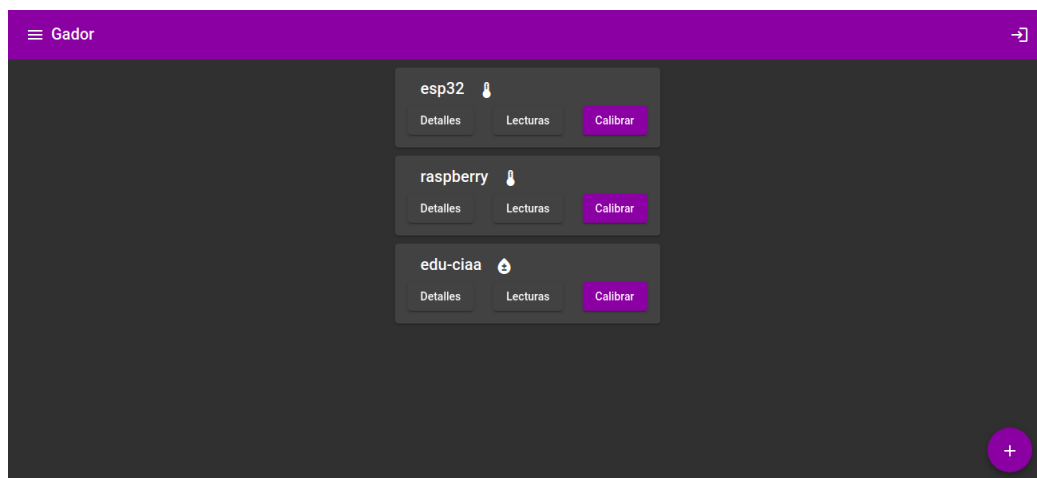


FIGURA 3.2. Lista de dispositivos.

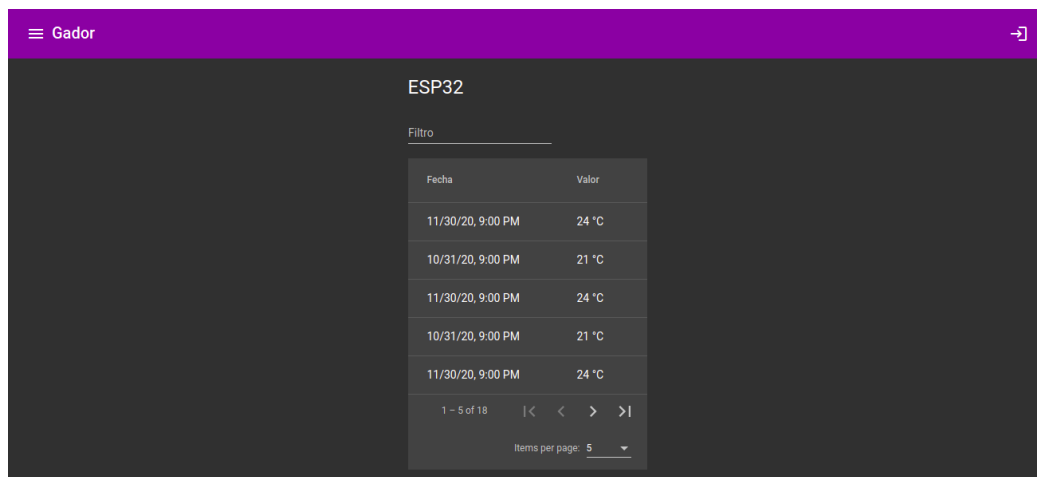
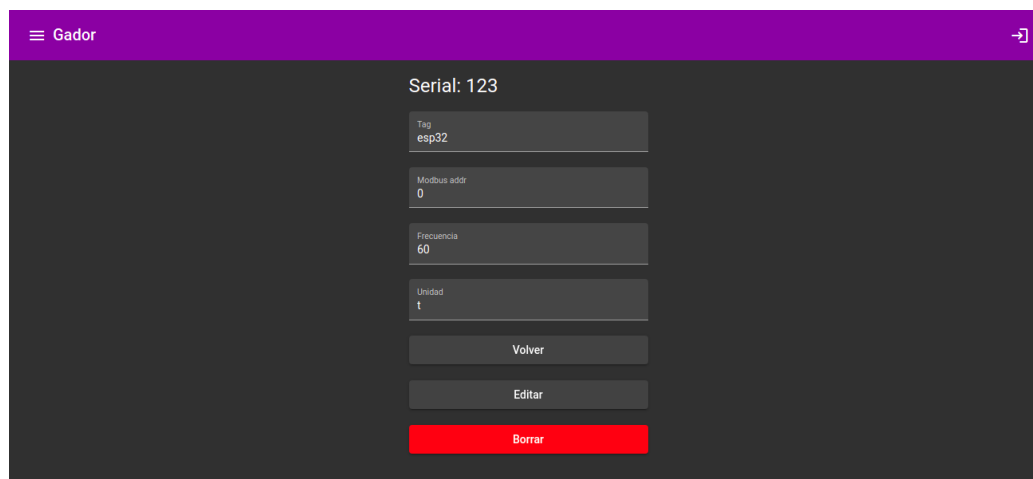


FIGURA 3.3. Lecturas del sensor.



Serial: 123

Tag	esp32
Modbus addr	0
Frecuencia	60
Unidad	t

Volver

Editar

Borrar

FIGURA 3.4. Detalles del dispositivo.

Capítulo 4

Ensayos y resultados

Este capítulo tiene la finalidad de explicar el proceso de aceptación del trabajo y como se determinó que los requerimientos se cumplieron. Además se expone la metodología utilizada para validar el código a medida que se fue escribiendo.

4.1. Recursos utilizados

Para desarrollar el trabajo y realizar las pruebas se utilizaron una serie de equipos que se detallan a continuación:

- Ordenador portátil Banghó
- Ordenador monoplaca Raspberry Pi 4 B
- Módulo Nodemcu Esp32 Wi-Fi
- Smartphone LG K20 Aurora Black
- Router Sagemcom F@ST 3890 v2 TLC

Los detalles del ordenador portátil son:

- Sistema operativo: Ubuntu 20.04 focal
- Kernel: x68_64 Linux 5.4.0-67-generic
- Shell: bash
- Resolución: 2732x768
- Entorno de escritorio: GNOME 3.36.5
- CPU: Intel Core i7-4710MQ @ 8x 3.5GHz
- GPU: Intel Corporation 4th Gen Core Processor Integrated Graphics Controller (rev 06)
- RAM: 11891MiB

Se utilizaron además una serie de programas para hacer el desarrollo y las pruebas y se los enumera a continuación:

- Visual Studio Code versión 1.54.3
- Navegador Chromium versión 89.0.4389.90 (Build oficial) snap (64 bits)
- Terminal de Gnome versión 3.36.2
- Postman versión 8.0.7

- Wireshark versión 3.2.3
- Nmap versión 7.80
- Mosquitto versión 1.6.9

4.2. Guiones y comandos

En el capítulo 3 se detalló el trabajo realizado, como funcionan todas sus partes y la interdependencia que existe entre ellas. Sin embargo, antes de llegar a tener un sistema completo y funcionando se tuvieron partes incompletas y servicios inexistentes. Esta situación demandó crear una serie de guiones y comandos que pudieran recrear de forma limitada alguna de las funcionalidades de las dependencias de cada componente.

4.2.1. Base de datos

Varios de los servicios necesitaron tener acceso a una conexión de base de datos en MongoDB durante su desarrollo. Para crear una instancia efímera del motor se utilizó un guión de Bash que se puede ver en el código 4.1. En él se puede observar que se utilizó Docker para esa etapa del trabajo. El código se escribió para facilitar las modificaciones en su configuración y se designó una carpeta para almacenar una serie de archivos en JavaScript. Estos archivos cumplieron la función de poblar con datos a MongoDB.

```

1 #!/bin/bash
2 IMAGE_NAME=mongo
3 CONTAINER_NAME=mongo
4 CONTAINER_PORT=27017
5 CONTAINER_DIRECTORY=/scripts
6 MACHINE_PORT=27017
7 MACHINE_DIRECTORY=$PWD/mockDB
8
9 docker run \
10 --rm \
11 --name $CONTAINER_NAME \
12 -p $MACHINE_PORT:$CONTAINER_PORT \
13 -v $MACHINE_DIRECTORY:$CONTAINER_DIRECTORY \
14 -d \
15 $IMAGE_NAME
16
17 sleep 5
18 docker exec $CONTAINER_NAME sh -c "mongo < /scripts/mockData.js"
```

CÓDIGO 4.1. Guión de base de datos.

4.2.2. Mosquitto

Los servicios que necesitaron de un *broker* MQTT para validar su desarrollo se valieron de un guión de Bash. Ese guión, que se puede ver en el código 4.2, se encargó de proveer un *broker* completamente promiscuo y sin ninguna medida de seguridad. La razón para generar esta configuración fue eliminar cualquier tipo de falla producto de las medidas de seguridad. De esta manera cualquier comportamiento no deseado se origina en el código escrito para cada servicio en particular.

```

1 #!/bin/bash
2 IMAGE_NAME=eclipse-mosquitto
3 CONTAINER_NAME=mosquitto
4 CONTAINER_PORT=1883
5 MACHINE_PORT=1883
6
7 docker run \
8 --rm \
9 --name $CONTAINER_NAME \
10 -p $MACHINE_PORT:$CONTAINER_PORT \
11 -d \
12 $IMAGE_NAME

```

CÓDIGO 4.2. Guión de Mosquitto.

Se utilizaron los servicios que provee Mosquitto para realizar publicaciones y subscripciones. Estas acciones fueron hechas desde la terminal del ordenador portátil y del ordenador monoplaca. Los comandos se pueden ver de forma genérica en el código 4.3.

```

1 mosquitto_sub -h 'localhost' -u 'docker' -P 'container' -t 'data/#'
2 mosquitto_pub -h 'localhost' -u 'device' -P 'thing' -t 'data' -m 'educiaa,25'

```

CÓDIGO 4.3. Comandos de Mosquitto.

4.2.3. Redis

Los componentes del sistema que utilizaron un mecanismo de identificación de clientes usaron Redis para lograrlo. Por ese motivo se creó un guión de Bash que creara un contenedor de Redis con su configuración por defecto. El código 4.4 muestra las instrucciones necesarias para lograr el objetivo. Se puede ver en su última línea que se creó un token de prueba para verificar la conexión con el servicio.

```

1 #!/bin/bash
2 IMAGE_NAME=redis
3 CONTAINER_NAME=redis
4 CONTAINER_PORT=6379
5 MACHINE_PORT=6379
6
7 docker run \
8 --rm \
9 --name $CONTAINER_NAME \
10 -p $MACHINE_PORT:$CONTAINER_PORT \
11 -d \
12 $IMAGE_NAME
13
14 sleep 5
15 docker exec redis sh -c "redis-cli SET xxxx.yyyy.zzzz 3"

```

CÓDIGO 4.4. Guión de Redis.

4.2.4. Nodejs

Se necesitó crear un ambiente de desarrollo y pruebas para poder escribir el código de los servicios basados en Nodejs. Esto se logró al construir los componentes que se detallan a continuación:

- Archivo de variables de entorno
- Archivo de configuración del framework de pruebas Mocha
- Creación de guiones en el archivo de paquetes de Nodejs

El archivo de variables de entorno se utilizó para no colocar en el código las contraseñas ni las direcciones de los recursos externos. Un ejemplo de este tipo de archivo se puede ver en el código 4.5. Su configuración apunta a servicios creados con los comandos y guiones de pruebas. La ventaja de esta metodología es que se puede usar el mismo código en producción y durante las pruebas.

```
1 PORT=8080
2 DB_URL=mongodb://localhost:27017/gador
3 REDIS_HOST=127.0.0.1
4 REDIS_PORT=6379
5 SECRET_KEY=secret
6 TIME_TO_LIVE=120
7 MQTT_HOST=mqtt://localhost
```

CÓDIGO 4.5. Archivo de variables de entorno.

Para configurar el framework de pruebas de Mocha se utilizó la configuración del código 4.6. Se observa que se determinó un tiempo de ejecución máximo para evitar bloquear el ensayo. La contingencia se puede dar en el caso que una función no pueda converger a un resultado.

```
1 module.exports = {
2   recursive: true,
3   slow: 75,
4   timeout: 5000,
5   spec: ['test/**/*.test.js']
6 }
```

CÓDIGO 4.6. Configuración de Mocha.

Finalmente se crearon guiones para correr con el gestor de paquetes de Nodejs. Se los puede ver en el código 4.7 y los guiones a analizar son los siguientes:

- dev
- test

El guión dev cumple la función de correr los guiones de creación de MongoDB, Redis y Mosquitto. Además carga las variables de entorno en la sesión de terminal para finalmente correr la aplicación en modo desarrollo. Con el entorno de desarrollo activo se pueden realizar las pruebas al ejecutar el guión test. Este guión limpia la terminal y reinicia la base de datos. Luego procede a cargar las variables de entorno y ejecutar Mocha.

```
1 "build": "npm install",
2 "dev": "./mockDB/mongo.sh && ./mockDB/redis.sh && ./mockDB/mosquitto.sh
   && eval $(cat ./env) nodemon ./src/app.js",
3 "production": "eval $(cat ./environment/.env) node ./src/app.js",
4 "test": "clear && ./mockDB/restart.sh && eval $(cat ./env) mocha",
```



```
5 "clean": "docker stop mongo && docker stop redis && docker stop
  mosquito"
```

CÓDIGO 4.7. Guiones de Nodejs.

4.3. Pruebas unitarias

Durante las primeras etapas del desarrollo del trabajo se utilizó la metodología *Test Driven Development (TDD)*. Luego se la abandonó y se pasó a utilizar distintos programas para realizar las pruebas. Se decidió realizar el cambio en la forma de trabajar debido a que la complejidad de las pruebas fue creciendo de manera exponencial. Se llegó a un punto del trabajo donde *TDD* se volvió improductivo y su abandono fue la única manera de continuar a un ritmo de avance aceptable.

En la primera etapa del trabajo se escribieron pruebas unitarias que fueron creciendo a medida que se agregaban funcionalidades al código fuente. En el caso del *endpoint* que representa los dispositivos en el Backend el archivo de pruebas unitarias alcanzó las 285 líneas de código. Mientras que el archivo del *endpoint* solo demandó 148 líneas. Se alcanzó el punto donde la extensión de las pruebas superaron al código fuente.

Algunos ejemplos de pruebas de la entidad dispositivos del Backend se puede ver en el código 4.8. El ejemplo se divide en cuatro pruebas realizadas al método *GET* de la entidad cuando se solicita un dispositivo en particular. Los casos analizados son:

- El pedido no tiene credenciales
- El pedido tiene credenciales inválidas
- El pedido tiene credenciales válidas y el dispositivo existe
- El pedido tiene credenciales válidas y el dispositivo no existe

```
1 describe('GET on "${url}/<device tag>" without a token', () => {
2   it('should return STATUS 401', async () => {
3     let response = await fetch(`${url}/esp32`, voidGET);
4     expect(response.status).to.be.equal(401);
5   });
6 });
7
8 describe('GET on "${url}/<device tag>" with an INVALID token', () => {
9   it('should return STATUS 401', async () => {
10    let response = await fetch(`${url}/esp32`, invalidGET);
11    expect(response.status).to.be.equal(401);
12  });
13 });
14
15 describe('GET on "${url}/<device tag>" with a VALID token', () => {
16   let data = { serial: 123, tag: "esp32", modbus: 0, freq: 60, unit: "
17   t" };
18   data = JSON.stringify(data);
19   it('should return "status 200" and data', async () => {
20     let response = await fetch(`${url}/esp32`, validGET);
21     let status = response.status;
22     let text = await response.text();
23     expect(status).to.be.equal(200);
24     expect(text).to.be.equal(data);
25   });
26 });
```

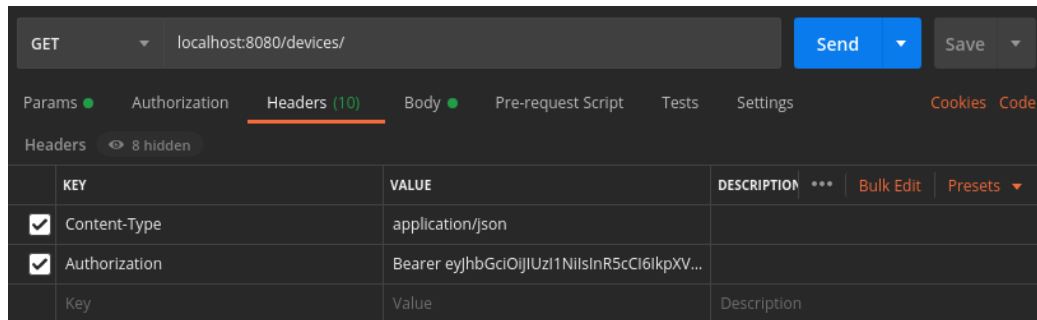



FIGURA 4.3. Pedido de dispositivos con Postman.

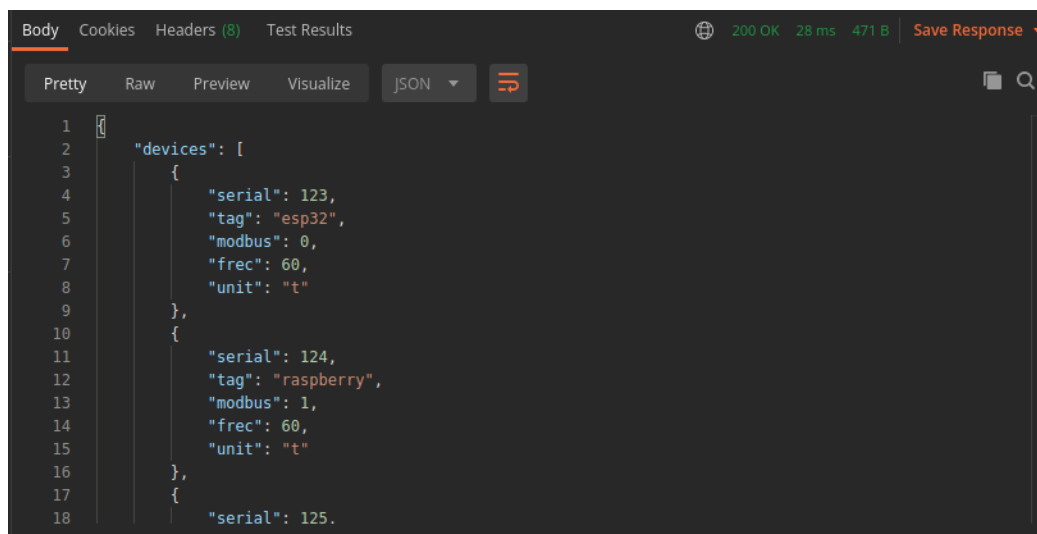


FIGURA 4.4. Respuesta de dispositivos con Postman.

Las funcionalidades de EBI se simularon con el código 4.9 escrito en Python. Cuando se invoca el programa se realizan lecturas y escrituras de los registros del servidor Modbus dentro de Nodos.

```

1 from pyModbusTCP.client import ModbusClient
2 import random
3
4 def r():
5     return random.randint(0, 65535)
6
7 client = ModbusClient(host='0.0.0.0', port=5020,
8                       auto_open=True, auto_close=True)
9
10 regs = client.read_holding_registers(0, 10)
11
12 if regs:
13     print('Holding Registers: ' + str(regs))
14 else:
15     print('reading error')
16
17 if client.write_multiple_registers(0, [r(), r(), r(), r(), r(), r(), r()
18   , r(), r(), r()]):
19     print('writing successful')
20 else:
21     print('writing error')

```

CÓDIGO 4.9. Simulación de EBI.

El usuario fue simulado utilizando un smartphone y se le solicitó a terceros que interactuaran con el sistema con el objetivo de probar la experiencia de uso y la facilidad para manejar el flujo de la aplicación.

Finalmente el dispositivo fue simulado con un modulo Nodemcu Esp32 Wi-Fi que entrega valores aleatorios en intervalos regulares. En el código 4.10 se detalla la función de envío de datos. Se tiene la posibilidad de usar los *topics* DATA y LIVE, para persistir en la base de datos o realizar calibraciones respectivamente.

```
1 void send_data ()
2 {
3   String temperature = String(random(MIN_TEMPERATURE, MAX_TEMPERATURE));
4   String my_message = String(DEVICE + temperature);
5   if (direction == 0)
6   {
7     mqtt.publish(DATA, my_message.c_str());
8   }
9   else
10  {
11    mqtt.publish(LIVE, my_message.c_str());
12  }
13
14  delay(frec);
15 }
```

CÓDIGO 4.10. Función de envío de datos.

4.5. Pruebas del cliente

El cliente realizó las pruebas que validan la comunicación del sistema con EBI. Para ese fin utilizó una máquina virtual con una licencia de prueba de seis horas que otorgó generosamente la empresa Honeywell. No fue posible probarlo en el ambiente productivo real debido a las políticas de seguridad de Gador.

Finalmente se realizó una demostración final a un representante del departamento de ingeniería de Gador. En ese evento se mostró todo el código y las funcionalidades. Se logró en esa instancia que el cliente diera su aprobación y conformidad. Con este último paso se finalizó el desarrollo del trabajo y se aceptó que se cumplieron con todos los requerimientos.

Capítulo 5

Conclusiones

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

5.1. Resultados obtenidos

El trabajo logró cumplir con los requerimientos que solicitó el cliente, que 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 a costa de un incremento en 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 testeado 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 ocurrió 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] Capterra. *Herramientas de monitorización de redes*.
<https://www.capterra.co/software/181692/check-mk>. Mar. de 2021. (Visitado 15-03-2021).
- [6] Google. *Announcing the Material Design Award Winners for 2020*.
<https://material.io/blog/mda-2020-winners>. Dic. de 2020. (Visitado 04-02-2021).
- [7] AWS. *Bayer Crop Science Drives Innovation in Precision Agriculture Using AWS IoT*. <https://aws.amazon.com/es/solutions/case-studies/bayer-cropsience/?c=i&sec=cs3>. Mar. de 2021. (Visitado 06-03-2021).
- [8] MongoDB. *Sharding*. <https://docs.mongodb.com/manual/sharding>. Mar. de 2021. (Visitado 06-03-2021).
- [9] docker docs. *Overview for Docker Compose*.
<https://docs.docker.com/compose/>. Mar. de 2021. (Visitado 07-03-2021).
- [10] Docker. *What is a container*.
<https://www.docker.com/resources/what-container>. Mar. de 2021. (Visitado 07-03-2021).
- [11] Eshna Verma. *The A-Z of Node.js*. <https://www.simplilearn.com/understanding-node-js-architecture-article>. Mar. de 2021. (Visitado 15-03-2021).
- [12] MongoDB. *La base de datos líder para aplicaciones modernas*.
<https://www.mongodb.com/es>. Mar. de 2021. (Visitado 07-03-2021).
- [13] DB-Engines. *DB-Engines Ranking*. <https://db-engines.com/en/ranking>. Mar. de 2021. (Visitado 07-03-2021).
- [14] 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).
- [15] Michael Oberdorf. *oits/modbus-server*.
<https://hub.docker.com/r/oits/modbus-server>. Mar. de 2021. (Visitado 08-03-2021).
- [16] the Docker Community. *redis*. Disponible: 2021-03-13. URL:
https://hub.docker.com/_/redis.