TRABAJO FIN DE MÁSTER

MÁSTER EN INGENIERÍA INFORMÁTICA

Lazarillo - Robot guía

Plataforma robótica de código abierto para uso general

Autor
Adrián Morente Gabaldón

Director
Juan José Ramos Muñoz

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Granada, septiembre de 2022

Lazarillo - Robot guía: Plataforma robótica de código abierto para uso general

Adrián Morente Gabaldón

Palabras clave: robot, embebido, *IoT*, *Linux*, *Yocto*, web, *C++*, *PubSub*, *React*, *Docker*, *Python*, *Redis*

Resumen

Lazarillo se trata de una plataforma libre de código abierto pensada para provisionar un robot que actúa como guía de caminos para los visitantes que acuden a la ETSIIT.

Este dispositivo embebido consta de un sistema operativo hecho a medida además de diversas aplicaciones para su uso, e internamente una arquitectura software que permite extender sus funcionalidades a todos los desarrolladores interesados.

En cuanto a *IoT*, la plataforma consta de métodos de conectividad que permiten al dispositivo ser administrado por un técnico desde un portal web, pudiendo así realizar actualizaciones o gestiones varias.

Lazarillo - Robot guide: Open-source multipurpose robotic platform

Adrián Morente Gabaldón

Keywords: robot, embedded, *IoT*, *Linux*, *Yocto*, web, *C++*, *PubSub*, *React*, *Docker*, *Python*, *Redis*

Abstract

Lazarillo is a free & open-source platform for provisioning a robot that shall behave as a path guide for the ETSIIT's visitors.

This embedded device contains a custom-made operating system in addition to some assorted applications. Internally, it contains a software architecture that allows any interested developers to extend its functionalities.

Regarding *IoT*, the platform includes connectivity methods that make a technician able to manage or upgrade the device through a web portal.

Yo, **Adrián Morente Gabaldón**, alumno de la titulación Máster en Ingeniería Informática de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI 77139229N, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: Adrián Morente Gabaldón

Granada a 9 de septiembre de 2022.

Agradecimientos

A Mari Carmen, Miguel y Lorena por animarme a cerrar esta etapa después de años posponiéndolo prestando atención a cosas más prioritarias. Porque “el tiempo pone cada cosa en su lugar”.

A mis ex-compis de uni por estar siempre ahí para apoyarnos en los caminos tan diversos que vamos tomando.

A Paola, que pese a no soportar la frialdad y perfeccionismo con que me tomo las cosas, siempre me obliga a que termine lo que me da pereza hacer.

Índice general

Capítulo 1

Introducción

Lazarillo se trata de una plataforma robótica abierta cuyo propósito es proporcionar una arquitectura solvente y extensible que permita añadir nuevas características y utilidades, además de facilitar el acceso a su gestión y mantenimiento.

Contiene un sistema operativo abierto basado en GNU/Linux y *The Yocto Project*, además de distintos servicios implementados con lenguajes de programación diferentes, mostrando así la interoperabilidad del software. Goza de conectividad inalámbrica, la cual facilita la conexión con herramientas externas para su gestión. Asímismo, el proyecto también consta de un panel web de administración desde el cual se pueden enviar acciones remotas al robot.

Aunque se trata de una plataforma extensible y de propósito múltiple, el uso inicial para el que fue ideado es el de actuar como **asistente** y **guía** dentro de un espacio cerrado (ya podemos ver que el título asignado al proyecto es un pequeño guiño a la literatura española). Sin embargo, el *stack* de herramientas y arquitectura que se han ido confeccionando durante su desarrollo, se podrían utilizar fácilmente para cualquier proyecto de propósito general que aúne dispositivos embebidos con IoT y administración remota de éstos.

Capítulo 2

Estado del arte

Dado que a lo largo de este documento estaremos hablando sobre dispositivos embebidos, *Internet de las cosas* y plataformas inteligentes (como robots), antes deberemos situarnos un poco en el contexto actual para revisar qué hay en el horizonte, teniendo en cuenta tanto software como hardware y limitaciones técnicas.

¿Qué es un "robot"?

Comencemos analizando la definición de la palabra ***robot***, que según el *IEEE* (*Instituto de Ingenieros Eléctricos y Electrónicos*, asociación mundial de expertos dedicada a la normalización del desarrollo en diversas áreas técnicas [1]), no *es algo fácil de definir*, pero una buena aproximación sería "una máquina autónoma capaz de percibir el entorno, realizando cálculos para la toma de decisiones y acciones que aplica en el mundo real" [2].

Un ejemplo de estos es la *Roomba* de la marca *iRobot*, la conocida aspiradora robótica que recorre las habitaciones de la casa de forma totalmente autónoma realizando una limpieza (más o menos a fondo) de ésta [3]. Los cálculos que realiza este dispositivo han de serle suficientes para recorrer la casa de forma eficiente, esquivando obstáculos y cubriendo la mayor superficie posible de la vivienda. Para este tipo de computaciones, un dispositivo robótico ha de valerse de sensores de proximidad (u otros), cámaras, etc.

Otro ejemplo mucho más claro de robot lo encontramos en ***Pepper*** [5] (obra de *SoftBank Robotics*), un computador con apariencia *humanoide* con hasta 20 grados de libertad que acepta interacción a través de una pantalla táctil, además de incorporar **reconomiento vocal** en 15 lenguajes diferentes. Por si fuera poco, a través del **reconocimiento facial** que hace de los usuarios es capaz de inferir su estado emocional.

Si bien el conjunto de avances de inteligencia artifical que realizan los robots de este tipo son espléndidos, no hace falta irse a un plano tan alto para hablar de robótica; ni aunar medidas tan avanzadas de detección y reacción ante el entorno, que también pueden ser usadas por elementos inteligentes que sin em-

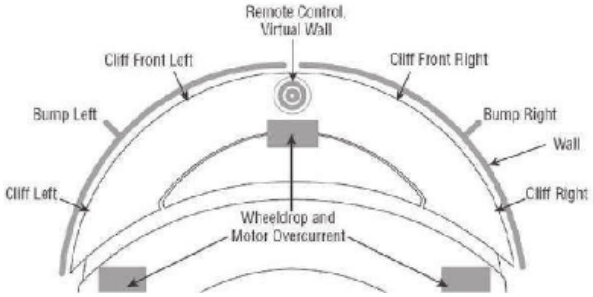

Figura 2.1: Ejemplo de sensores en una *Roomba* - Fuente: *ResearchGate* [4]

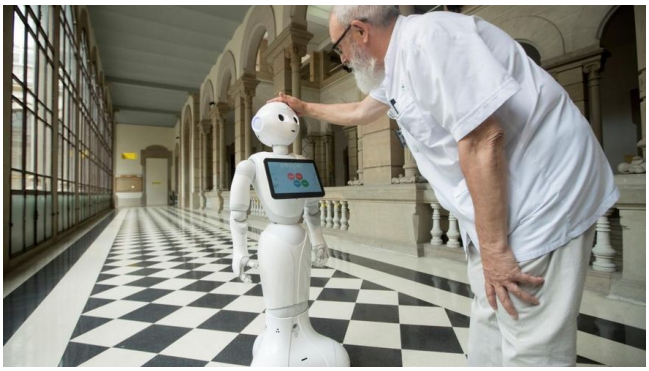

Figura 2.2: El robot humanoide Pepper proporciona atención a los pacientes en hospitales de Barcelona, brindándoles apoyo moral y entretenimiento - Fuente: *Noticias de Navarra* [6]

bargo no llegan a ser considerados como "robots". Podemos encontrar un ejemplo en el *Autopilot*, que es como llama la compañía automovilística de vehículos eléctricos *Tesla* a su sistema de conducción automática [7]. Los complejísimos algoritmos de **visión computacional** que tienen lugar en el vehículo a la hora de observar el entorno en busca de otros coches, peatones, señales y obstáculos; también pasan por un entrenamiento de modelos de inteligencia artificial intenso. La diferencia en el uso de estos, es que (al menos actualmente), aún no existe la conducción plenamente autónoma, y por seguridad aún se requiere que haya un conductor al volante. ¿Podríamos entonces establecer el límite entre lo que es un "robot" y lo que no en función de si es necesaria la interacción humana?

Responder a ésto es siempre complejo pero interesante, así que me gustaría concluir esa pregunta volviendo al artículo de "What is a Robot?" del *IEEE* [2] con la divertida respuesta de *Joseph Engelberger*:

"No sé cómo definir un robot, ¡pero sé reconocer uno cuando lo veo!"

¿Qué necesito para hacer un robot?

Si quisiéramos crear un nuevo robot, son varios los campos de conocimiento que deberíamos tener en cuenta. Para empezar, la **electrónica** es algo esencial, ya que su base será la que permitirá controlar el movimiento del dispositivo a través de pulsos de corriente, además de dotarlo de los periféricos de detección y comunicación que hemos mencionado previamente. Por otro lado, el campo de la **informática** permitirá extender las funcionalidades de este robot dotándole de un sistema operativo y añadiendo aplicaciones sobre la capa abstracta tejida por la electrónica.

Si hablamos de hardware, podemos empezar por adquirir una plataforma móvil ya preparada para su uso. Lo primero que puede pensar cualquiera es tomar una como la *Roomba*, cuya movilidad y detección de obstáculos ya está implementada, y sobre ella añadir los componentes y prestaciones que queramos. El problema en cuanto a ésto es que se trata de una **plataforma cerrada**. No disponer del código fuente hace que no podamos extender el robot como queramos, y así el fabricante se reserva los derechos de su uso.

Si no podemos asumir el presupuesto de una plataforma ya montada o simplemente queremos hacer pruebas y divertirnos con la experiencia, hoy en día es fácil y asequible acceder a componentes electrónicos con los que crear pequeños robots y artefactos caseros. Un ejemplo de *núcleo* para ésto sería una placa de *Arduino*, que por unos 20 o 30 euros contiene las conexiones para añadirle módulos de cámara, altavoces, servomotores y cualquier cosa que se nos ocurra [8].

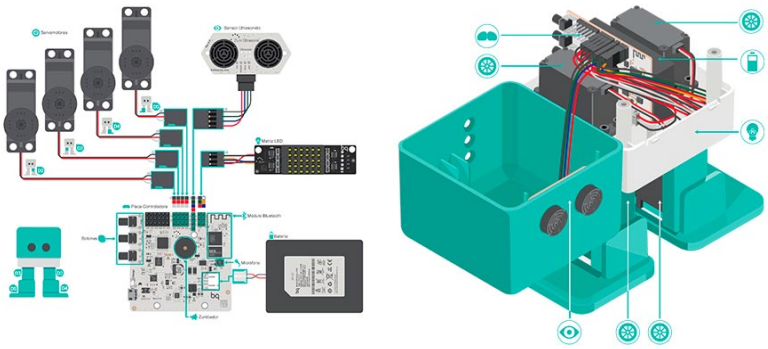

Figura 2.3: *Zowi*, ejemplo de robot basado en Arduino mas sensores - Fuente: *Balara* [9]

Esta placa de desarrollo contiene un microcontrolador que permite ser programado en código C muy fácilmente, pero está limitado a eso. Si queremos utilizar un sistema que nos permita gestionar más cosas que la simple ejecución de un código, como un sistema operativo completo, deberíamos dar el salto a algo más parecido a un ordenador común, para lo que puede servirnos la conocida

Raspberry Pi [10]. Existen otras alternativas de la misma gama provenientes de *Asus* o de *NXP* [11], pero nos ceñiremos a la primera ya que está muy enfocada al sector educativo y al público joven y goza de una comunidad muy numerosa. Lo que todos estos dispositivos comparten son diversas conexiones a través de las cuales podemos añadir los elementos que mencionábamos previamente para nuestro proyecto de robot, además de permitirnos acceso a la configuración de su sistema operativo (o incluso podemos confeccionar uno a nuestra medida, como veremos en el siguiente apartado).

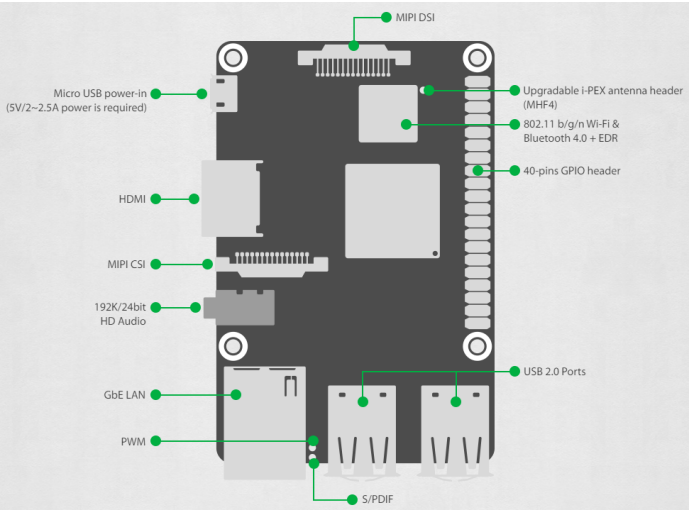

Figura 2.4: *Tinker board*, la alternativa de *Asus* - Fuente: *Asus.com* [12]

Sistema operativo, el cerebro del robot

Continuando con la *Raspberry*, el ordenador monoplaca en torno al cual podríamos construir nuestro robot, pasemos ahora a hablar del sistema operativo. Para esta placa en cuestión existe uno muy aclamado por la gente llamado ***Raspbian*** [13], basado en *Debian*, una conocida distribución de *GNU/Linux* que viene de los propios creadores de la *RPi* y dispone de todo lo necesario para formar un sistema de operativo completo, incluyendo hasta interfaz gráfica y aplicaciones de escritorio.

Contando con instalar en la tarjeta de memoria un sistema operativo ya montado como éste, podríamos seguir adelante con la conexión de periféricos que conformarán nuestro robot, o quizás nos interese más detenernos un momento a decidir si realmente queremos usar opciones como *Raspbian* o preferimos tener algo personalizado y a nuestra medida.

Para los más aventureros, existe la posibilidad de diseñar y compilar un sistema operativo propio (aunque también basado en *GNU/Linux*) con herramientas como ***Buildroot*** [14] y ***The Yocto Project*** [15]. Se tratan de frameworks que permiten compilar una distribución *Linux* a medida según las necesidades del

dispositivo embebido en cuestión seleccionando las piezas que lo conforman como si de un puzzle se tratase. Por ejemplo, un robot que no tenga pantalla pero sí informe al usuario mediante efectos sonoros puede prescindir de todos los paquetes gráficos; igual que otro que no necesite conectarse a un router de forma inalámbrica no necesita tener instalados todos los paquetes correspondientes a la gestión de *WiFi*.

Por un lado, tenemos la decisión entre utilizar un sistema operativo ya compilado o diseñarlo como hemos comentado. Si miramos el estudio que realizó *Mads Doré Hansen* en su paper *Yocto or Debian for Embedded Systems* [16] podemos matizar nuestra respuesta en consideración.

> "*Debian* es bueno para pruebas rápidas y entornos de escritorio con memorias grandes y requisitos bajos de mantenimiento. *Yocto* es bueno para entornos personalizados con soporte a distintas plataformas hardware de poca memoria y que requieren trazabilidad y reusabilidad.

Sobre *Debian*, también menciona que al ser conveniente para pruebas rápidas, muchos equipos de desarrollo comienzan su trabajo en la plataforma embebida utilizándolo, posponiendo tanto el paso a utilizar *Yocto* para diseñar un sistema propio, que terminan postérgandolo infinitamente y no lo realizan nunca. Un argumento a favor de ellos es que la **curva de aprendizaje** de *Yocto* es grande, dado que requiere compilar todo el sistema para la arquitectura destino, mientras que opciones como *Debian* seguramente ya dispongan de paquetes precompilados.

Por ahora, nos conformaremos con saber de la existencia de estas alternativas. Más adelante comentaremos cuál nos resulta más conveniente para el caso que nos ocupa.

Programación del robot

Pasemos ahora a hablar de **software**. ¿Qué herramientas necesita un desarrollador para programar su robot? Pues bien, indagando un poco podremos ver que realmente no dista mucho de codificar cualquier otro software convencional.

En el artículo de *Codete* citado [17], listan por orden de volumen de uso los lenguajes de programación más utilizados para robótica. Podemos ver que todos los lenguajes ahí presentes son de uso general, y que cualquier experiencia que ya podamos tener por ejemplo con *Python* para desarrollo web, nos puede servir para iniciarnos en el mundo de la robótica. Otro ejemplo es *Java*, que si bien también es usado extensamente para aplicaciones de escritorio entre otras, aporta herramientas y librerías que pueden ser muy útiles para recepción y procesamiento de imágenes.

Sin embargo, de acuerdo con dicho artículo, el más ampliamente usado es *C* y *C++*, ya que permiten un manejo más eficiente de la memoria y un procesamiento a bajo nivel más cercano al hardware. Además, estos lenguajes colaboran

directamente con la API del sistema operativo, utilizando las propias librerías ya presentes en el dispositivo.

Por otro lado, cuando se habla de robótica suele salir a colación ***ROS*** *(Robot Operating System)* [18], un proyecto de **código abierto** que incluye un conjunto de librerías y herramientas con las que no es necesario *reinventar la rueda* para adentrarse en la programación de robótica. En diversos artículos podemos encontrar razones por las que utilizar *ROS* (como en [19]):

- Se trata de un sistema agnóstico para lenguajes. Permite comunicar un robot programado en *C++* con otro hecho en *Python*, por ejemplo.
- Es ligero y de propósito general para cualquier finalidad que tenga el robot en cuestión.
- Contiene herramientas de simulación que facilitan el desarrollo del robot antes de requerir el uso físico de electrónica real (lo que puede suponer un ahorro en costes de material).

Para un uso modesto, quizá el aspecto más interesante de utilizar *ROS* puede resultar la comunicación entre distintos **nodos**. Podemos entender como *nodo* cada una de las partes independientes del robot que colabora con el resto. Para un robot hipotético que se desliza sobre el suelo y que mueve los brazos para agarrar cosas, podríamos distinguir entre sus nodos la plataforma móvil, las manos, las cámaras que observan el entorno y un largo etcétera. En cuanto a cómo se comunican estos nodos, podemos plantearlo como un problema de comunicación de distintos servicios cualesquiera. Veremos esto en el siguiente apartado.

Para finalizar con el apartado de programación, un apunte: deberemos tener siempre presentes los **principios *SOLID* de la programación orientada a objetos**, que muy eficazmente se describen en la documentación de ***Digital Ocean*** [20]:

- **S** *(Single)* - *Responsabilidad única*: cualquier clase o módulo debe cumplir un solo propósito, por lo que ese debería ser su único trabajo y razón para cambiar su implementación.
- **O** *(Open)* - *Abierto-cerrado*: los objetos deben estar abiertos para extensión y cerrados para modificación.
- **L** *(Liskov)* - *Sustitución de Liskov*: "cada clase derivada debe ser sustituible por su clase base".
- **I** *(Interface)* - *Segregación de interfaz*: una clase no debe forzarse a implementar una interfaz que no usa o depender de métodos que no va a utilizar.
- **D** *(Dependency)* - *Inversión de dependencia*: las entidades deben depender de interfaces y no clases concretas (ya que éstas pueden cambiar sin que lo esperemos).

Teniendo esto en mente y cumpliendo estas directivas, podremos implementar un código más escalable y limpio. Además, el uso de interfaces y clases abstractas permitirá reemplazar piezas de software sin afectar a las clases derivadas.

¿Cómo se comunican los nodos de un robot?

Como anticipábamos, podemos entender cada **nodo** como un servicio independiente que se comunica con el resto a través de algún mecanismo integrado en el sistema operativo.

Imaginemos que tenemos un nodo **principal** que representa el cerebro del robot, y que por tanto es el encargado de ordenar al resto de nodos (*secundarios*) la siguiente acción que deben realizar. Si cada uno de estos elementos es un servicio independiente como hemos dicho, es necesario un método de comunicación para que el principal notifique a los demás mediante un mensaje. Si buscamos métodos de **comunicación entre procesos**, seguramente el primer resultado que obtengamos sea ***IPC***, cuyas siglas (*Inter-Process Communication*) en castellano significan exactamente "comunicación entre procesos" [21]. En la siguiente figura podemos ver una comparativa de las dos formas de comunicación posibles en este ámbito: *Shared memory* y *Message Passing*

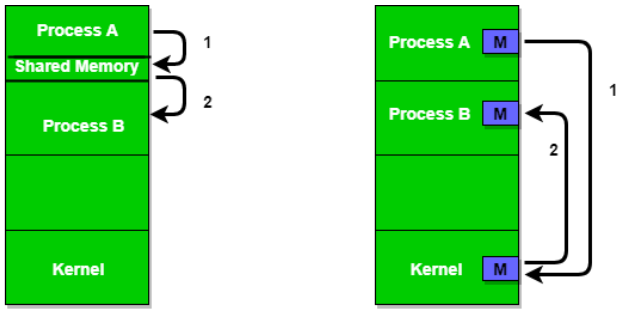

Figura 2.5: Formas de comunicación *IPC* - Fuente: *GeeksforGeeks.org* [21]

El método de comunicación mediante *IPC* más inmediato es el uso de *Shared memory* o **memoria compartida**, que consiste literalmente en reservar un espacio de la memoria lógica para que todos los procesos cooperantes hagan uso de un espacio común donde leer y escribir variables compartidas. Si bien ésto es ágil y rápido, mi experiencia trabajando con esta metodología en proyectos grandes es que si no se tiene especial cuidado a la hora de diseñar qué datos intervienen y su procedencia, a largo plazo ésto produce un gran **acoplamiento** en el software involucrado.

Por otro lado, *Message Passing* (**paso de mensajes**) consiste en habilitar un búfer compartido a nivel de kernel en el que pueden escribir y leer los procesos interesados. La diferencia con la **memoria compartida** es que en ésta se pueden utilizar directamente las variables del lenguaje de programación escogido, mientras que con **paso de mensajes** lo que se hace es escribir objetos completos con un identificador de mensaje. Así, lo que hacen los servicios para leer información es iterar sobre dicho búfer buscando si los identificadores de mensaje presentes son de su interés o no. Con esta aproximación, es sencillo enviar mensajes *1:1* (de un servicio a otro) o *1:n* (*multicast* o *broadcast*) entre servicios concretos.

Dejando *IPC* a un lado y volviendo a *ROS*, que antes mencionábamos que es un set de librerías y herramientas bien amplio, destaquemos ahora que también contiene formas de comunicación entre nodos, e incluso su documentación contiene tutoriales sobre cómo empezar a trabajar en ello (como encontramos en [22], donde enseñan cómo partir con un modelo ***publicación-suscripción*** en lenguaje *C++*).

Este patrón también conocido como *PubSub* se trata de una forma de **mensajería asíncrona** para integrar microservicios en una arquitectura software. O como detallan en *Amazon AWS* [23]: "para usar en arquitecturas *serverless* o basadas en **microservicios**".

El funcionamiento de éste es simple y permite implementar **arquitecturas basadas en eventos**:

- Un **servicio** se suscribe a un **tópico**, como lo haría un usuario de la televisión por cable a los canales cuyo contenido le interese.
- Otro servicio **publicará mensajes** en dicho tópico, de forma que el primero los recibirá inmediatamente.
- Esta mecánica se generaliza con tantos servicios y tópicos como sea necesario.

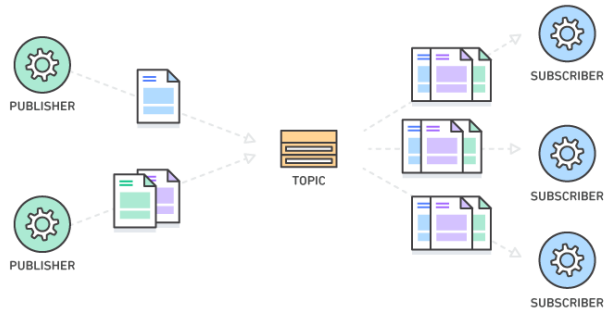

Figura 2.6: Mecánica *PubSub*. Dos servicios publican sobre un mismo tópico para que otros tres reciban su contenido inmediatamente - Fuente: *Amazon AWS* [23]

Sobre *PubSub*, encontramos también el **servicio de mensajería *MQTT***, que funciona de forma similar. En la imagen siguiente. vemos que ahora se sitúan al **publicador** y al **suscriptor** a ambos lados de un **bróker**, que es el responsable de notificar a cada uno de los servicios suscritos cuando se ha añadido un mensaje a su tópico de interés.

En mi experiencia, el modelo *PubSub* es el más cómodo de utilizar una vez que el proyecto ya está preparado para trabajar en él y añadir funcionalidades, además de gozar de una gran **escalabilidad**, ya que añadir nuevos mensajes, tópicos y servicios es totalmente transparente y práctico. Bien es cierto que para

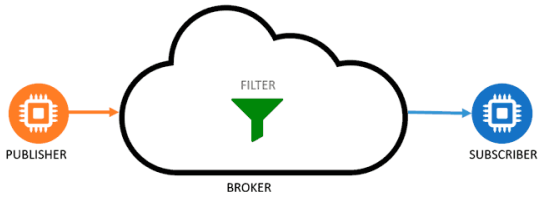

Figura 2.7: Mensajería *MQTT* con patrón **publicador/suscriptor**. El bróker intermedio es quien recibe los mensajes que van a cada tópico y notifica a cada servicio suscrito. - Fuente: *Luis Llamas* [24]

no reinventar la rueda, es conveniente utilizar librerías que ya implementen este paradigma, como *Redis* o *ROS*, que mencionábamos antes.

Si bien hemos visto que *ROS* puede servir para cualquiera que quiera iniciarse en la robótica, *Redis* es una herramienta muy interesante a tener en cuenta para intercomunicar los distintos nodos o procesos que tendrán lugar en el dispositivo. Se trata de un **almacén de estructuras de datos** que puede actuar como base de datos, caché o bróker de mensajes; que se mantiene en memoria principal, por lo que es volátil a la par que muy rápida [25]. Además, implementa el paradigma *PubSub*, permitiendo realizar suscripciones a tópicos y/o publicar mensajes sobre ellos. Automáticamente, los programas que utilicen alguna librería de *Redis* son notificados cuando un mensaje es publicado en un tópico al que están suscritos [26].

Dejemos aquí la enumeración y descripción de los distintos módulos software que nos pueden servir para programar el susodicho robot. En apartados posteriores estudiaremos cada una de las decisiones tomadas al respecto.

Conectividad del robot

Para terminar con el *Estado del Arte*, veamos por último qué opciones hay a la hora de comunicar nuestro robot con la red exterior, en caso de que necesitemos enviar telemetría, recibir órdenes remotas o incluso recabar datos para un servidor.

La primera alternativa que nos encontramos se trata del conocido protocolo ***HTTP*** (*HyperText Transfer Protocol*), que utilizamos a diario cuando consultamos cualquier web a través de nuestro ordenador o teléfono móvil. De acuerdo con la documentación de *Mozilla* [27], es un protocolo de escritura cliente-servidor donde la comunicación es iniciada por el **cliente**, es decir, el interesado en recibir los datos (normalmente una página web). La particularidad de este protocolo es que cada conexión termina en cuanto el servidor provee una respuesta completa al cliente. Veámoslo en la siguiente figura:

Una alternativa que encontramos hoy en día al clásico *HTTP* es el protocolo ***WebSockets***, que de acuerdo con la documentación para desarrolladores de *Mozilla*: *"es una tecnología que permite abrir una sesión de comunicación* ***interactiva*** *entre el cliente y el servidor"* [29].

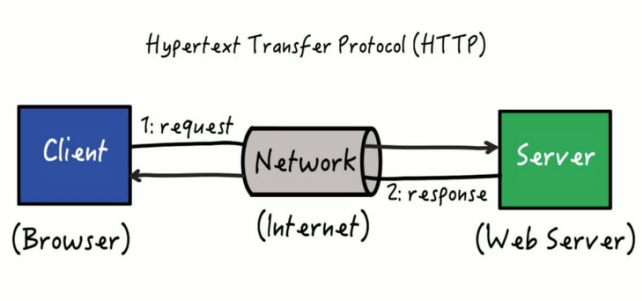

Figura 2.8: Funcionamiento del protocolo *HTTP*. Un cliente realiza una petición a través de la red, el servidor recolecta los datos necesarios, los envía al cliente y finaliza la conexión. - Fuente: *Research hubs* [28]

Las diferencias entre éste y el anterior son:

- La conexión en *HTTP* finaliza con cada respuesta exitosa, mientras que en *WebSockets* es interactiva, permaneciendo abierta entre el cliente y el servidor, sin necesidad de reabrirla para cada nueva petición.
- *WebSockets* es una tecnología **basada en eventos**, por lo que no es necesario realizar lecturas periódicas de cara al otro extremo ya que los mensajes enviados por éste *notifican* al servicio que la utiliza.
- La comunicación es **bidireccional**: una vez que se establece la conexión, tanto el cliente como el servidor pueden mandarse mensajes entre sí transparentemente sin necesidad de renegociar el canal de conexión.

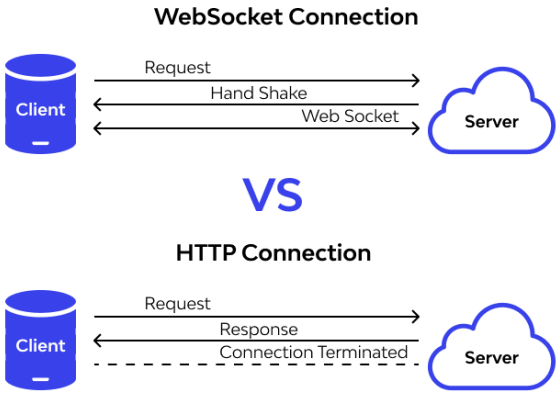

Figura 2.9: La diferencia entre la conexión en *WebSocket* y *HTTP* la vemos en la última línea de cada diagrama. En el primer caso el canal permanece abierto y en el segundo se cierra tras la respuesta del servidor. - Fuente: *Wallarm.com* [30]

Para terminar con esta sección, cabe recordar también otra vez ***MQTT***, que pese a no ser un protocolo como tal, permitirá orquestar los robots en torno a un servidor de forma que puedan compartir un canal de **mensajería *push*** sobre un servidor acordado. Este elemento actuaría como **nodo principal** que podría publicar en distintos tópicos, y los robots actuarían como nodos **secundarios** y suscritos.

Si bien siempre es posible indagar un poco más en todas las alternativas y tecnologías que tenemos disponibles a día de hoy, pasemos de una vez a enumerar los requisitos que regirán el desarrollo de este proyecto.

Capítulo 3

Especificación y requisitos

Pasemos ahora a describir de la forma más detallada posible cada uno de los requerimientos que conforman la idea del proyecto, planteando del mismo modo alternativas o aspectos que sería de agrado incluir, si bien no forman parte de la especificación inicialmente. Las decisiones tomadas, así como las soluciones implementadas, serán detalladas en el capítulo siguiente, si bien a lo largo de éste mismo pueden surgir necesidades cuya solución se detalle directamente.

Se desea disponer de una plataforma robótica extensible, libre y abierta; que permita una buena ampliación de nuevas características mediante software. Además, se propone la implementación de un uso concreto para esta plataforma, y es que dicho robot sirva como **guía de caminos** para sus usuarios finales en un **entorno controlado**. En este capítulo listaremos los distintos requisitos y puntualizaremos sobre cada una de las decisiones tomadas para satisfacerlos.

3.1. Requisitos generales

Los requerimientos aquí listados comprenderán cosas tanto de procedimientos para el desempeño del proyecto (como pueden ser la visibilidad y su legislación) hasta las funcionalidades más concretas que se esperan del producto final.

3.1.1. Licencias

Se tratará de un proyecto de software **libre** y de **código abierto**. Para ello, se publicará bajo una licencia *GPLv3* en un repositorio público en *Github*.

Además de *Github*, existen otras alternativas de repositorios públicos que permiten utilizar *git* para control de versiones (como *Bitbucket* y *Gitlab* entre otras). El porqué de utilizar *Github* es meramente por aprovechar la licencia *premium* que se provee a los estudiantes de la UGR simplemente por matricularse; permitiendo así tener algunos repositorios privados a su disposición [31].

Con el código público y la licencia elegida, cualquier usuario podrá descargar el software, compilarlo, ejecutarlo e incluso añadir modificaciones al código para ser probadas e integradas en la plataforma final. Para ello, en el propio repositorio se pondrá a disposición de los interesados una documentación que explique cómo replicar el entorno tanto de desarrollo como de compilación.

3.1.2. Especificaciones

El término *plataforma robótica* puede ser demasiado amplio para su manejo, por lo que en esta sección detallaremos más a fondo algunos de los puntos más interesantes, así como los factores de éxito que harían de ***Lazarillo*** un producto útil y diferencial con respecto a las alternativas ya existentes.

Extensibilidad

Ya que se desea disponer de una plataforma extensible cuyo comportamiento y características puedan ampliarse a través de software, se debe dotar al robot de una arquitectura que permita este crecimiento, conteniendo en ella servicios (o módulos) independientes que puedan incluirse o no en función de la aplicación específica que vaya a cumplir el robot.

Para ello, sería interesante disponer de una arquitectura basada en **microservicios** donde cada uno de ellos cumple un propósito muy concreto y se comunica con el resto sin generar acoplamiento. Para esto es de imperativa necesidad utilizar un paradigma de comunicación en el cual sea transparente añadir datos y servicios.

Por otro lado, ha de asegurarse que existe la **interoperabilidad**, la cual representa que una arquitectura que incluye distintas plataformas de hardware y diferentes sistemas software (implementados con lenguajes de programación variados) cooperan bien entre sí.

Veamos un ejemplo rápido para ilustrar el párrafo anterior: si el programa que recibe los datos de un servidor web está programado en *Python* y debe transmitirlos al servicio que toma las decisiones de movimiento (programado en *C++*); el paradigma de comunicación que los conecta ha de ser **agnóstico en el lenguaje** y que dicha operación sea efectiva de forma transparente.

Conectividad

Como plataforma inteligente y conectada que utiliza el paradigma del ***edge computing***, sabemos que el robot ha de ser un dispositivo embebido con conexiones al exterior como *Bluetooth* y/o *WiFi*. Que éstas vengan implícitas en la plataforma hardware utilizada facilitará mucho el trabajo, ya que nos ahorramos el hecho de tener que soldar componentes. En cuanto a plataformas hardware, en el apartado del **Estado del Arte** ya comentamos algunas alternativas y opciones. Utilizaremos para ésto una *Raspberry Pi Model 3 B*, que pese a no ser

el último modelo de la marca *Raspberry*, es la que tengo a disposición en casa. Además, satisface las necesidades de conectividad que comentábamos.

Por otro lado, en cuanto al requisito de computación en el borde, la plataforma deberá contener uno o más servicios que permitan la comunicación con el exterior, de una forma u otra, además de enviar y recibir mensajes. El robot deberá habilitar un canal de comunicación **persistente** y **bidireccional** que le permitan tanto a él como al servidor enviarse eventos entre sí.

Interfaz y experiencia de usuario

El robot contará con una pantalla táctil con la que proveerá la información necesaria al usuario (en función de las aplicaciones que necesite o decidan integrarse en el robot). Cualquier pantalla táctil que permita su conexión con la *Raspberry* debería servir, por lo que no entraremos a detallar limitaciones hardware. Para la interacción del usuario, el sistema contará con una **aplicación embebida de entorno gráfico** que permita el uso del robot.

Dado que ***Lazarillo*** se pretende que actúe como **guía**, otra característica que sería de agradecer en la plataforma tiene que ver con la **reproducción de sonidos** que faciliten la comunicación con el usuario, así como la **accesibilidad**. No todo el mundo goza de capacidad visual o simplemente no están habituados a interfaces táctiles, por lo que emitir alertas y sonidos descriptivos facilitaría llegar a más usuarios de forma plena y satisfactoria.

Gestión experta y mantenimiento

Como hemos comentado en secciones anteriores, el robot gozará de hardware provisto de conectividad inalámbrica. En este punto haremos uso de esta característica para ofrecer un método de mantenimiento, supervisión y gestión del robot, por parte de alguna "mano experta". Necesitaremos un método de administración de los distintos robots existentes desde un portal web externo a ellos. Un técnico encargado de gestionar los robots, accederá a una web alojada en un servidor a través del protocolo común de *HTTP*.

Inicialmente, este portal web servirá para listar los dispositivos conectados (es decir, los robots que han sido provisionados con el software de *Lazarillo* y se encuentran en funcionamiento), pero posteriormente permitirá enviar acciones remotas desde el servidor al robot (como reinicios, actualizaciones de software, acciones concretas a realizar por el robot, etc.).

Es deseable que la interfaz web sea sencilla y usable. Además, sería de agradecer que ésta pueda visualizarse correctamente en **dispositivos móviles**, ya que ampliaría el rango de posibilidades de gestión de los dispositivos robóticos.

Movilidad

El factor determinante que diferenciará a nuestro robot de un sistema empotrado inmóvil será la capacidad de desplazarse por el entorno. Ya sea para un uso u otro, el robot deberá venir dotado de un sistema hardware que le permita avanzar por el plano, conteniendo elementos como **motores** y **ruedas** o **cintas móviles**.

Además, si se desea que el robot sea **inteligente** y reconozca el entorno por el que se está moviendo, deberá dotarse de algún sistema de reconocimiento como **sensores de proximidad** o **cámaras**. Respecto a esto, si queremos seguir el paradigma de *edge computing* como venimos comentando, el procesamiento de estas señales podría realizarse en el servidor en lugar de en el propio robot, lo que también liberaría a la plataforma hardware del robot de una buena parte de la computación.

Para acotar el alcance del proyecto y que sea asumible para un trabajo de este calibre, inicialmente la movilidad podrá estar basada en hacer **seguimiento de líneas** sobre el suelo.

Figura 3.1: Ejemplo de robot móvil autónomo - Fuente: *Robotnik*

3.2. Objetivos opcionales

En esta sección enumeraremos y describiremos sucíntamente qué otras ideas surgieron durante el momento de *brainstorming* del proyecto, y que si bien son opcionales para su desempeño, realmente aportarían algún valor al producto final.

Movilidad autónoma

Un factor que haría de ***Lazarillo*** un producto totalmente independiente y útil sería que no necesitase de caminos guiados para desplazarse. Se valoraría la instalación en sí mismo de los mapas cerrados en que se ubicaría durante su desempeño, así como un mecanismo de **geolocalización** en el espacio. Contando con esto, el robot mantendría una comunicación persistente con el servidor

informando en *tiempo real* de la ubicación actual.

Capítulo 4

Implementación

En este capítulo comentaremos a fondo las decisiones tomadas que hemos comentado previamente en base a la especificación, además de describir las implementaciones propuestas con las distintas tecnologías utilizadas. Pondremos el foco no solo en el **software implícito en el robot** sino también a los componentes desarrollados externos a él, pero que también conforman la arquitectura completa de la plataforma, como lo son el **sistema operativo**, el **panel web de administración** y otras herramientas involucradas. Las ideas que por una razón u otra no hayan llegado a implementarse a tiempo, serán definidas en profundidad en apartados posteriores.

Para elaborar este capítulo de una forma más legible y entendible, seguiremos una estructura muy similar a la propuesta en el capítulo 3 de **Especificación de requisitos**.

4.1. Requisitos generales

4.1.1. Licencias y compartición del software

En cuanto a licencias y permisos del proyecto, como se anticipó se usarán diversos **repositorios** en *Github*, que aparecen listados a continuación y descritos en función del contenido (que se elaborará más tarde):

- ***lazarillo-embedded*** [32]: se trata del software embebido programado en ***C++*** que da lógica al robot.
- ***lazarillo-admin-frontend*** [33]: es la página web que utilizan los técnicos para supervisar los robots y enviarles determinadas acciones. Se encuentra hecho en ***React*** y utiliza *WebSockets* para la conexión con el dispositivo.
- ***lazarillo-admin-backend*** [34]: es un servicio web que actúa como intermediario entre el robot y el servicio de ***frontend*** para la negociación de la conexión. Está programado en ***Python*** usando la librería de ***Flask*** principalmente.

- ***meta-lazarillo*** [35]: aquí se aloja el *layer* (o "capa") personalizada para *The Yocto Project* con la que se especifican las dependencias de ***Lazarillo*** en la compilación del sistema operativo del robot.

- ***documentacion-tfm*** [36]: en este repositorio se alojan los fuentes de LaTeX para generar esta misma documentación que se está leyendo.

Todos los repositorios listados se publican bajo la licencia ***GPL*** en su *versión 3* (cuya última declaración es del 29 de junio de 2007 [37]) y que permite a cualquier interesado utilizar el software e incluso ampliar sus funcionalidades mientras que posteriormente se distribuya con la misma licencia.

Habiendo establecido ésto, entremos en materia y veamos todos los detalles de la implementación.

4.2. Sistema operativo

Como comentábamos en el capítulo del *Estado del arte* en el apartado de *Sistema operativo, el cerebro del robot*, existen diversas alternativas para componerlo. Partiendo de la base de que contamos con una *Raspberry* como núcleo del computador, las opciones más sonadas son utilizar ***Raspbian*** (sistema ya compilado e instalable en la plataforma) o crear nuestra propia distribución utilizando **el proyecto *Yocto***.

¿Por qué omitimos un sistema operativo que ya existe? Pues bien, la respuesta es fácil. *Raspbian* es un sistema operativo de uso general que permite utilizar la *Raspberry* como cualquier ordenador normal, ya sea para ofimática, desarrollo de software, consumo de recursos multimedia o incluso videojuegos. Esto provoca que el sistema operativo en cuestión venga con demasiado *bloatware* (software innecesario que no se utiliza) preinstalado de base; y llevaría más tiempo modificar la imagen de *Raspbian* para que no contenga todo el software no deseado que confeccionar un sistema operativo a medida. Además, deseamos que el robot solo muestre una aplicación embebida en pantalla en lugar de un entorno de escritorio normal, por lo que podemos prescindir de este entorno completo y configurar nuestro nuevo sistema para que solo muestre una aplicación y ahorre tiempo en el arranque.

La ventaja de crear nuestra propia imagen de sistema permitirá que incluyamos solo las librerías y dependencias que realmente necesitemos, obviando aspectos que no nos sea necesario incluir y que tomarían un espacio útil en la máquina. Además, siendo un proyecto de código abierto permitirá que interesados aprovechen el sistema operativo y lo extiendan con las funcionalidades y programas que les apetezca. Pasemos a continuación a comentar la implementación realizada.

Nota: este documento no se pretende que sea una guía de cómo utilizar *Yocto* y entender todo su ecosistema, por lo que solo destacaremos las configuraciones personalizadas hechas para darle forma a ***lazarillo-image***.

Para empezar a trabajar con *Yocto*, lo primero es elegir **la versión de *Yocto*** a utilizar y seguir su documentación oficial sobre *Quick Build* [38], en la que se detalla cómo clonar el proyecto y realizar una primera compilación de prueba. Esta ejecución inicial llevará al menos un par de horas, ya que necesitará descargar todas las librerías necesarias y compilarlas.

Release Activity

Codename	Yocto Project Version	Release Date	Current Version	Support Level	Poky Version	BitBake branch	Maintainer
Langdale	4.1	October 2022		Future - Support for 7 months (until May 2023)	N/A		Richard Purdie <richard.purdie@linuxfoundation.org>
Kirkstone	4.0	May 2022	4.0.3 (August 2022)	Long Term Support (minimum Apr. 2024)	N/A	2.0	Steve Sakoman <steve@sakoman.com>
Honister	3.4	October 2021	3.4.4 (May 2022)	EOL	N/A	1.52	Anuj Mittal <anuj.mittal@intel.com>
Hardknott	3.3	April 2021	3.3.6 (April 2022)	EOL	N/A	1.50	Anuj Mittal <anuj.mittal@intel.com>
Gatesgarth	3.2	Oct 2020	3.2.4 (May 2021)	EOL	N/A	1.48	Anuj Mittal <anuj.mittal@intel.com>
Dunfell	3.1	April 2020	3.1.19 (August 2022)	Long Term Support (until Apr. 2024)	23.0	1.46	Steve Sakoman <steve@sakoman.com>
Zeus	3.0	October 2019	3.0.4 (August 2020)	EOL	22.0.3	1.44	Anuj/Armin

Figura 4.1: La versión elegida es ***Dunfell 3.1*** ya que lleva el suficiente tiempo en el mercado como para ser estable, y además es versión *Long Term Support* y asegura soporte oficial hasta abril de 2024. Versiones más novedosas o no *LTS* pueden no asegurar tanta estabilidad. - Fuente: *Yocto Releases* [39]

Acto seguido, lo que hacemos para continuar con la creación de nuestra propia distribución son varias cosas:

- Crear nuestro propio *layer*, llamado ***meta-lazarillo*** donde estableceremos nuestras directivas.
- Crear una **imagen** de sistema, que se llamará ***lazarillo-image*** y que incluirá el software programado por nosotros y que se instalará automáticamente en el robot.
- Añadir a la imagen creada previamente las dependencias para que sus aplicaciones se ejecuten correctamente.

El contenido de este *layer* puede verse en la web [35] pero igualmente listaremos aquí los archivos más representativos.

conf/layer.conf

Este primer archivo contiene las variables más genéricas para la configuración inicial del *layer* y suele tener un formato parecido en todos los *metas* que se crean, ya que principalmente fija el nombre característico de la colección además de la rama de **compatibilidad** (que es *dunfell*) como hemos comentado arriba.

```
1 BBPATH .= ":${LAYERDIR}"
2
3 BBFILES += "${LAYERDIR}/recipes-*/*/*.bb \
4 ${LAYERDIR}/recipes-*/*/*.bbappend"
5
6 BBFILE_COLLECTIONS += "lazarillo"
```

```
7  BBFILE_PATTERN_lazarillo = "^${LAYERDIR}/"
8  BBFILE_PRIORITY_lazarillo = "1"
9
10 LAYERSERIES_COMPAT_lazarillo = "dunfell"
11 LAYERVERSION_lazarillo = "1"
```

recipes-core/images/lazarillo-image.bb

A continuación, veamos el archivo que realmente define **la imagen** del sistema operativo, cuyo nombre utilizaremos cuando queramos compilarlo.

```
1  inherit core-image lazarillo-class
2
3  GLIBC_GENERATE_LOCALES = "es_ES.UTF-8"
4
5  ###################################################
6  # Groups of needed packages
7
8  CORE_PKGS = " \
9          kernel-devicetree \
10         kernel-image \
11         kernel-modules \
12         openssl \
13 "
14
15 MISC_PKGS = " \
16         curl \
17         nano \
18         psplash \
19 "
20
21 QT_PKGS = " \
22         qtbase qtbase-plugins qtbase-tools \
23         qtdeclarative qtdeclarative-qmlplugins \
24         qtgraphicaleffects qtgraphicaleffects-qmlplugins \
25         qtmultimedia qtmultimedia qtmultimedia-plugins qtmultimedia-
               qmlplugins \
26         qtquickcontrols qtquickcontrols-qmlplugins \
27         qtquickcontrols2 qtquickcontrols2-qmlplugins \
28         qtvirtualkeyboard qtvirtualkeyboard-plugins qtvirtualkeyboard-
               qmlplugins \
29 "
30
31 LAZARILLO_PKGS = " \
32         lazarillo-embedded \
33 "
34
35 ###################################################
36 # Installation of grouped packages
37
38 IMAGE_FEATURES += " \
39         autologin \
40 "
41
42 IMAGE_INSTALL += " \
43         ${CORE_PKGS} \
44         ${MISC_PKGS} \
45         ${QT_PKGS} \
46         ${LAZARILLO_PKGS} \
47 "
```

Todas las constantes del tipo *X_PKGS* tan solo son valores auxiliares que se concatenan a *IMAGE_INSTALL* para que en la compilación se incluyan todas estas dependencias en el sistema de ficheros final.

Si nos fijamos en el valor de *LAZARILLO_PKGS* podemos ver que ahí se lista **lazarillo-embedded**, que se trata de la aplicación embebida que comentába-

mos que implementará toda nuestra lógica del robot. Vemos el contenido de su **receta** (como se llaman en *Yocto* los archivos que definen cómo se debe obtener una pieza de software, compilarla e instalarla en el destino) en el siguiente apartado.

recipes-core/lazarillo-embedded/lazarillo-embedded.bb

```
# Compiles Qt project with cmake after downloading
inherit cmake_qt5

# AUTOREV references to newest commit
SRCREV = "${AUTOREV}"
LICENSE = "GPLv3"

EXTRA_OECMAKE = "-DCOMPILE_MODE=d"

# Reference to repo where to download the software from
SRC_URI = "git://git@github.com/adrianmorente/lazarillo_hmi.git;protocol=ssh"

DEPENDS += " \
        qtbase \
        qtdeclarative \
        qtgraphicaleffects \
        qtmultimedia \
        qtquickcontrols \
        qtquickcontrols2 \
        qtvirtualkeyboard \
"

do_install() {
        install -d ${D}${bindir}
        install -m 0700 lazarillo-hmi ${D}${bindir}
        install -m 0700 motor-manager ${D}${bindir}
        install -m 0700 web-gateway ${D}${bindir}
}

FILES_${PN} = "${bindir}"
```

En el archivo mostrado se configura la *URL* del repositorio del que *Yocto* descargará el proyecto (usando las claves *SSH* configuradas en el dispositivo de compilación).

Además, encontramos listadas las dependencias necesarias para la compilación exitosa del proyecto. Como utilizamos el framework *Qt* en su versión **open source** (disponible en [40]) para construir la aplicación embebida, se enumeran las librerías que *Yocto* deberá descargar e instalar también.

Para finalizar, con *do_install* establecemos que se copien al directorio normal de binarios del sistema operativo final (normalmente */usr/bin*) los tres programas generados por *lazarillo-embedded*, con permisos de escritura, lectura y ejecución solo para el usuario actual. Por ahora nos ceñiremos a lo estrictamente relacionado con *Yocto* y estos servicios los definiremos en detalle más adelante.

Configuración local y compilación

Una vez que tenemos nuestro *layer* preparado para que ***bitbake***, la herramienta en *Python* que gestiona las compilaciones de *Yocto*, pueda encontrar nuestros paquetes y dependencias, pasamos a configurar el archivo *build/conf/-local.conf*, que es el realmente imprescindible. En este archivo podríamos haber añadido todo el contenido de configuración necesaria para compilar nuestra imagen, si bien es mucho más limpio utilizar *layers* separados que poder versionar y controlar de forma independiente según la *release* de *Yocto* que queramos usar.

El contenido usado para el archivo es el siguiente (se han obviado algunas líneas por defecto de *Yocto* que solo son para desarrollo y que no aportan mucho para el alcance de este documento):

```
1  MACHINE = "raspberrypi3"
2
3  DISTRO = "poky"
4  DISTRO_FEATURES_append = " alsa alsa-plugins gles2 opengl pulseaudio systemd
       wifi"
5  DISTRO_FEATURES_REMOVE_append = " x11 wayland"
6  PACKAGECONFIG_pulseaudio += " systemd"
7  VIRTUAL-RUNTIME_init_manager = "systemd"
8  DISTRO_FEATURES_BACKFILL_CONSIDERED = "sysvinit"
9
10 PACKAGE_CLASSES = "package_deb"
11
12 BB_NUMBER_THREADS = "4"
13 PARALLEL_MAKE = "-j 4"
14
15 EXTRA_USERS_PARAMS += " usermod -a -G audio root;"
16 EXTRA_USERS_PARAMS += " usermod -P lazarillo123 root;"
17
18 EXTRA_IMAGE_FEATURES ?= "debug-tweaks"
```

En primer lugar, fijamos la *MACHINE* de destino para la que queremos compilar, en este caso la *Raspberry Pi 3* como hemos comentado anteriormente. Además, decimos a *Yocto* que genere nuestra imagen de sistema operativo utilizando *poky* como distribución base (que es la *de facto* en la herramienta), además de añadir paquetes como *Alsa*, *Pulseaudio*, *OpenGL* y *Systemd*.

- *Alsa* y *Pulseaudio* son dependencias para los sistemas de sonido, que aunque no se utilizan actualmente, se incluyen en el sistema para cuando se quieran emitir alertas sonoras a través del servicio **lazarillo-hmi** y así proporcionar una interfaz más cercana y accesible para los usuarios del robot.

- *OpenGL* (y su complemento *GLES2*) se incluyen ya que son requisitos indispensables para que el framework de *Qt* con aplicaciones de interfaz gráfica funcione como esperamos en dispositivos embebidos. Gracias a esto, utilizamos el lenguaje *QML* del framework (basado en *Javascript*) para diseñar dicha aplicación con interfaz táctil.

- *Systemd* se establece en el sistema como gestor de servicios y **demonios** (o *daemons*, pequeños servicios que se ejecutan en segundo plano en el sistema). Este es el gestor de arranque de servicios por defecto en la mayoría de distribuciones modernas que utilizamos hoy en día para escritorio.

Para terminar, fijamos a *Bitbake* que utilice 4 hilos en la compilación. Cuantos más se utilicen, más tareas de descarga/compilación/instalación podrán ejecutarse en paralelo. Obviamente dependerá de la máquina utilizada.

Para terminar, añadimos una contraseña al usuario *root* por defecto para que no se pueda manipular libremente el robot por consola si se accede por *ssh*.

Una vez tenemos listo nuestro *layer* y el archivo *local.conf*, podemos lanzar la orden ***bitbake lazarillo-image*** en el directorio de desarrollo de *Yocto* y ver cómo comienza a descargar dependencias y compilarlas en la arquitectura de la *RPi*:

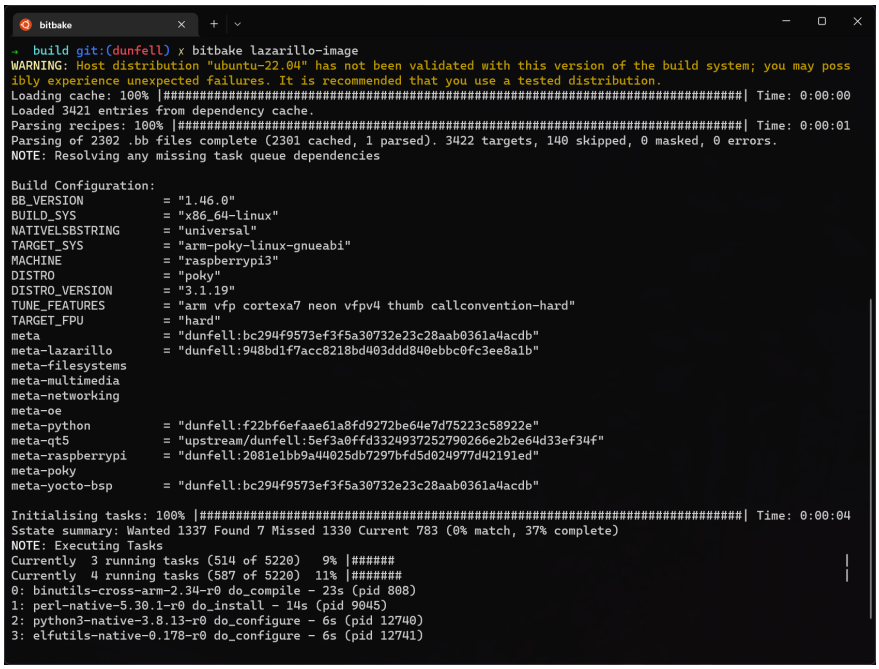

Figura 4.2: Captura de pantalla de *Ubuntu 22.04* sobre el Subsistema Linux de Windows, usado para la compilación del nuevo sistema operativo.

Llegados aquí, solo quedará esperar unas horas a que termine todo el proceso. Una vez finalizado, grabaremos el archivo ***lazarillo-image-raspberrypi3.sdimg*** generado bajo el directorio *build/tmp/deploy/* en la tarjeta de memoria que queramos usar en el robot, la insertamos en la *Raspberry* correspondiente y probamos nuestro nuevo y flamante sistema operativo personalizado.

Personalización del sistema operativo

Para cerrar con el sistema operativo, comentemos el caso de ***psplash*** [41], un programa que se instala en **lazarillo-image.bb** dentro del grupo *MISC_PKGS* como dependencia externa y que permite mostrar una imagen estática durante el arranque del dispositivo final.

Detalles como éste hacen que el dispositivo tenga un aspecto más profesional, dado que en lugar de mostrarse una imagen genérica de *GNU/Linux* o, peor si cabe, una terminal de arranque; el robot mostraría en su pantalla una imagen customizada (que podría ser un logo personalizado del proyecto, si lo hubiera).

Basta con integrar esta sencilla aplicación dentro del sistema operativo, cambiar la imagen que se desea mostrar y (re)compilar el sistema operativo. Sin embargo, por falta de tiempo, no he podido generar un logo personalizado para el proyecto que mostrar, por lo que el logo por defecto de *OpenEmbedded* es el mostrado al iniciar.

Figura 4.3: Imagen mostrada por *psplash* al arranque de la máquina: logo de *OpenEmbedded*, organización contribuidora activamente en el Proyecto *Yocto* - Fuente: *OpenEmbedded.org* [42]

4.3. Arquitectura de Lazarillo

Una vez que hemos visto la integración completa del sistema operativo, en esta sección enumeraremos los distintos servicios y módulos específicos creados para *Lazarillo* y describiremos el propósito para el que han sido programados. Como venimos comentando, habrá ciertas partes que serán enteramente embebidas en el robot, y por otro lado también existirán otros componentes software que estarán situados "fuera".

Una idea que se ha tenido muy presente durante el desarrollo del proyecto es la de cumplir los famosos **principios *SOLID*** de la programación orientada a objetos. Véase en el capítulo de *Estado del arte* cuáles son cada uno de los puntos.

Para cumplir esto, se implementan módulos separados de **responsabilidad única**, lo que resulta en clases no muy grandes y fácilmente mantenibles con un propósito muy claro y que colaboran entre sí. Además, se utilizan interfaces generales que permiten luego a las clases derivadas gestionar su comportamiento más concreto.

Se persigue una arquitectura que carezca de **acoplamiento** y el uso de interfaces ayuda a que ciertos componentes software sean luego intercambiables y no estrictamente dependientes. Por ejemplo: si se desea no utilizar *Redis* para

la mensajería interna y se intercambia por *MQTT*, este cambio solo se realiza en el módulo ***messaging-broker*** y resulta totalmente transparente al resto de servicios, que directamente heredan ese comportamiento.

Para hacernos una idea general de todos los entes involucrados en el software, prestemos atención al diagrama de la figura 4.4.

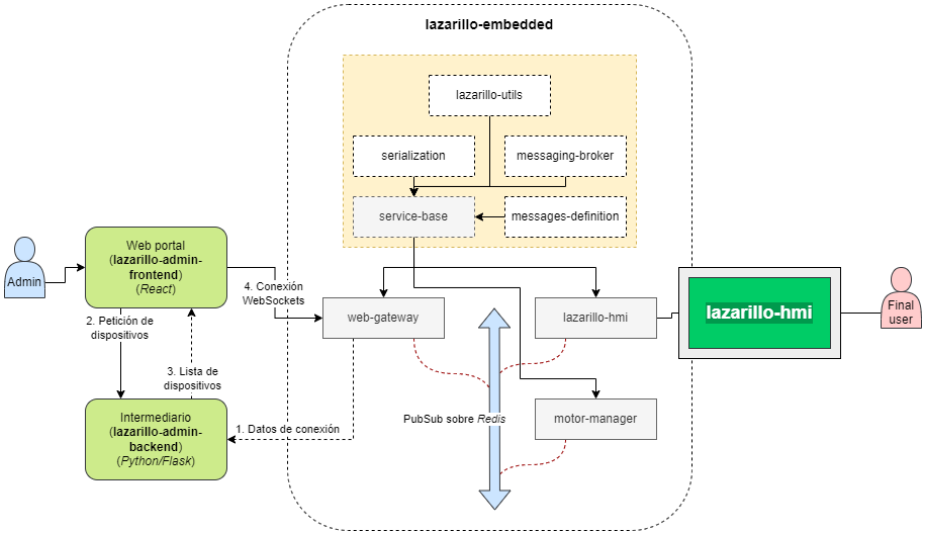

Figura 4.4: Diagrama mostrando la arquitectura del proyecto, incluyendo conexiones entre servicios y relaciones internas.

A la izquierda del todo, junto al agente denominado *Admin* se ubican los dos servicios web involucrados que ya hemos mencionado (**lazarillo-admin**, arriba **frontend** y **backend** abajo). A su derecha encontramos un rectángulo grande que engloba todo el software enclaustrado (**lazarillo-embedded**) y para finalizar, la interfaz de usuario (gráfica táctil y sonora) con la que interactuarán los que necesiten de la atención de *Lazarillo*.

Para explicar la implementación, separaremos todo el software en tres módulos: uno para el software embebido y otro por cada servicio web (que son dos). Aunque ya venimos anunciando cosas, entraremos a continuación en detalles más profundos de lenguajes de programación y librerías utilizadas.

4.3.1. lazarillo-embedded

Si volvemos a mirar el diagrama 4.4, distinguimos el cuadrado blanco que incluye los módulos presentes en *lazarillo-embedded*. Por un lado tenemos los comprendidos en el rectángulo blanco que se tratan de **librerías**, las cuales se enlazan al compilar a los **servicios**, representados por rectángulos grises (y comunicados sobre el búfer de comunicación en azul). En esta sección describiremos la razón de existencia de cada uno y el propósito que persiguen, empezando

por las librerías y acabando por los programas que se ejecutan en el robot.

Hagamos un pequeño inciso aquí para comentar que la compilación está gestionada con la herramienta *CMake* [43], una opción escalable y muy útil para proyectos grandes. Permite establecer funciones y macros comunes para todos los componentes involucrados dentro de un proyecto, utilizando variables para customizar la compilación. Además, con el uso de *cpack* también permite generar archivos *.deb* entendibles por el sistema operativo para su posterior instalación.

Antes de nada, veamos la estructura del proyecto hasta el primer nivel de directorios, sin entrar en cada uno, definiendo cada uno de ellos así como los ficheros presentes y la utilidad de cada uno, aunque algunos con más enjundia se desarrollarán más tarde:

NOTA: los directorios se muestran en negrita cursiva y los archivos solo en cursiva.

- ***cmake/****ProjectSetup.cmake*: contiene el código común de *CMake* para la correcta compilación del proyecto. Define las banderas de compilación, rutas donde localizar las librerías necesarias e implementa funciones y macros que utilizarán los servicios para enlazarse con las librerías de las que dependen (*Qt*, *Json*, *Redis* y *Systemd*).
- ***lazarillo-hmi/***: subproyecto que implementa la aplicación gráfica mostrada en la pantalla táctil del robot.
- ***lazarillo-utils/***: librería que añade objetos estáticos accesibles para todos los subproyectos. En esta fase solo contiene métodos estáticos para obtener la fecha y hora del sistema y así proveerla de forma estándar a los servicios.
- ***messages-definition/***: este módulo no es en sí un servicio que se ejecute directamente en el sistema operativo sino una **librería** de la que se nutren los demás donde se definen los mensajes que utilizan internamente para comunicarse. Ésta a su vez se enlaza con el servicio que describiremos más tarde y cuyo fin es **serializar** los mensajes.
- ***messaging-broker/***: esta librería es la que establece toda la comunicación con el servidor de *Redis* para el manejo de **mensajería sobre *PubSub***. Contiene clases para gestionar la conexión y utiliza el patrón de diseño *Factory* para la creación de los objetos que publican mensajes y se suscriben a tópicos. Al gestionar la conexión en solo este módulo, si se decidiera cambiar *Redis* por otra alternativa, al ser un comportamiento heredado para los servicios, el cambio debería ser solo un trámite.
- ***motor-manager/***: idealmente, este directorio contiene un nuevo servicio llamado *motor-manager* cuya idea es la de **controlar los distintos motores y elementos electrónicos que permiten que el robot se mueva**. Contendría clases distintas por cada elemento conectado e intercambiaría mensajes con todos ellos. Desafortunadamente, **no ha llegado a implementarse** ninguna funcionalidad en esta fase del proyecto.

- ***serialization/***: esta librería contiene los tipos de datos para **serialización** (y deserialización) utilizando el formato *Json*. Realmente se podría prescindir de este módulo y almacenar todos los mensajes como texto plano sobre *Redis* y enviarlos así a los servicios. Lo que permite la serialización es transformar los datos para almacenarlos en un búfer (o cualquier recipiente) y que luego puedan ser reconstruidos en el mismo o en otro dispositivo. Se sigue esta aproximación (*por si acaso*) simplemente por dotar de algo más de escalabilidad al proyecto por si sigue creciendo y se quieren hacer más cosas con los mensajes.
- ***service-base/***: aunque el nombre pueda inducir a pensar que se trate de un *servicio*, realmente es una librería que incluye interfaces y clases que reutilizarán todos los servicios que quieran crearse posteriormente. Realiza acciones generales interesantes como **instanciar el bróker de mensajería** (mencionado arriba) y proveer de métodos que se ejecutarán tanto al inicio como al final del resto de programas directamente.
- ***test_env/***: si bien esta parte no ha llegado a probarse profusamente, el objetivo que persigue es el de crear un entorno de pruebas para el desarrollo en forma de contenedores de *Docker*. Con un archivo *docker-compose.yml* se levantan dos instancias de éstos, uno que simula los servicios instalados en el robot y otro que mantiene el servidor de *Redis* donde se realiza la comunicación. Usando esto, es posible **probar el código del robot en nuestra máquina local**, ahorrando tiempo.
- ***web-gateway/***: esta sencilla aplicación hace de *gateway* entre el robot y el panel externo de administración. Mantiene un servidor *WebSocket* que recibe los comandos de la web, los procesa y traduce a mensajes (de *messages-definition*) que luego publica en *Redis* para que las aplicaciones interesadas y suscritas los reciban. De esta forma, un administrador que usa el panel web puede enviar distintas órdenes.
- *.clang-format*: ésto no es más que una recomendación a desarrolladores. Si bien este archivo no aporta funcionalidad como tal al proyecto, sí que lo hace al desarrollo. *ClangFormat* [44] se trata de una herramienta que permite dar formato automático al código conforme se escribe. Si se configura este archivo con el editor de texto y las variables siguiendo las **guías de estilo** que se acuerden con el equipo de desarrollo, todos los programadores involucrados podrán trabajar sobre un estilo común. Ésto evitará conflictos con el control de versiones cuando alguien inserta espacios sin percatarse o más saltos de línea de la cuenta.
- *build.sh*: este archivo es un ***script*** realmente útil para el desarrollo. Internamente ejecuta los comandos de *CMake* con sus parámetros necesarios, permitiendo customizar el proceso con argumentos simples como *-j8* para utilizar 8 hilos en la compilación, *-c d* para compilar en modo de depuración o *–cpack* para generar los binarios de instalación si la compilación es exitosa. Véase la imagen 4.5.
- *CMakeLists.txt*: finalmente, este es el archivo central del proyecto. Simplemente utiliza el código de la carpeta ***cmake/*** para incluir el comportamiento general allí descrito y añade a la compilación cada uno de los módulos (exactamente las librerías y los servicios que ya hemos listado).

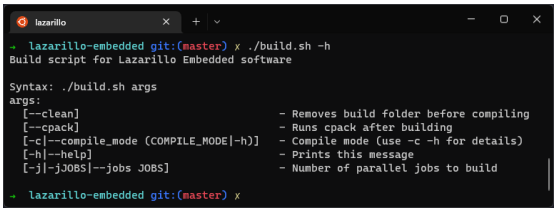

Figura 4.5: Captura de pantalla del uso del script *build.sh* para compilación de *lazarillo-embedded*.

A partir de ahora, veremos el código más relevante para entender la lógica del proyecto. Por temas de espacio, se obviarán del código las guardas, los *namespaces* y los comentarios, además de incluir la implementación de los métodos (ubicados en los archivos **.cpp* en lugar de **.h* como de costumbre). Si se quieren ver los archivos completos, véase el enlace del título de cada fragmento.

messages-definition

En el punto de implementación alcanzado a la hora de la entrega del proyecto no existen muchos mensajes, pero la librería se ha pensado de forma que sea sencillo añadir mensajes nuevos para los nuevos desarrolladores que quieran participar en ello y utilizar *Lazarillo*.

Para ilustrar este comportamiento, veremos dos clases: una **interfaz genérica** para todos los mensajes y una **clase objeto** de ejemplo que la implementa. Empecemos por la interfaz:

i_base_message.h

```
1  class IBaseMessage : public serialization::Serializable
2  {
3  public:
4          ~IBaseMessage() {}
5
6          virtual std::string name() const = 0;
7
8          void setHeader(MessageHeader const &p_header)
9          {
10                 m_header = std::make_shared<MessageHeader>(p_header);
11         }
12
13         MessageHeader getHeader() const
14         {
15                 return (m_header) ? *m_header : getDefaultHeader();
16         }
17
18         MessageHeader getDefaultHeader() const;
19         {
20                 MessageHeader header;
21                 header.name = name();
22                 return header;
23         }
24
25         virtual serialization::Serializer & serializePayload(serialization::
               Serializer &p_serializer) const = 0;
26
27         serialization::Serializer & serialize(serialization::Serializer &
               p_serializer) const override
```

```
28          {
29                  return serializePayload(p_serializer);
30          }
31
32  private:
33          std::shared_ptr<MessageHeader> m_header;
34  };
```

Vemos que esta clase abstracta hereda de *serialization::Serializable*, presente en el módulo de **serialization** y que provee los métodos que han de sobreescribirse para añadir los campos al objeto serializado que se guardará en memoria.

El mensaje abstracto utiliza una *MessageHeader* que compone la **cabecera** que heredarán todos los mensajes. Tiene tres campos:

- **name**: nombre del mensaje en cuestión, a cumplimentar por cada clase que herede de *IBaseMessage*.
- **source**: nombre del servicio que lo envía.
- **timestamp**: fecha y hora en la que se compone el mensaje para su envío.

Además de proveer métodos de acceso para la cabecera, declara los métodos de serialización y deserialización que implementarán cada una de las clases hijas. Un detalle a considerar en éstos es que reciben y devuelven referencias al objeto de serialización; esto se hace para poder concatenar los campos en una sola línea. Ejemplo:

```
1  # Sin usar referencias como argumento y retorno:
2  p_serializer.add("campo1");
3  p_serializer.add("campo2");
4  p_serializer.add("campo3");
5  return p_serializer;
6
7  # Usando referencias:
8  return p_serializer.add("campo1").add("campo2").add("campo3");
```

Veamos ahora un ejemplo de mensaje que implemente dicha interfaz. Usaremos la clase *EventConnectionStatus*, que representa el mensaje que se envía del servicio **web-gateway** al resto de servicios cuando cambia la conexión por *WebSocket* con el portal de administración:

event_connection_status.h

```
1  class EventConnectionStatus : public IBaseMessage
2  {
3  public:
4          static constexpr char const *NAME{"EVENT_CONNECTION_STATUS"};
5
6          explicit EventConnectionStatus(models::ConnectionStatus p_status)
7          : m_status{p_status} {}
8
9          explicit EventConnectionStatus(serialization::Deserializer &
               p_deserializer)
10         {
11                 p_deserializer.extract("Status", m_status);
12         }
13
14         serialization::Serializer & serializePayload(serialization::
               Serializer &p_serializer) const final
15         {
```

```
16                  IBaseMessage::serializePayload(p_serializer);
17                  return p_serializer.add("Status", m_status);
18          }
19
20          std::string name() const final { return NAME; }
21
22  private:
23          models::ConnectionStatus m_status{models::ConnectionStatus::UNKNOWN};
24  };
```

Podemos ver que utilizando esta aproximación, el código necesario para crear un nuevo mensaje es bastante sencillo. Simplemente sobreescribe el nombre con el que se identificará y especifica qué campos han de añadirse o leerse. La variable que se utiliza en cuestión (*m_status*) no es más que un enumerado con los posibles estados de la conexión (*UNKNOWN*, *CONNECTED*, *NOT_CONNECTED*).

service-base

Este módulo define el esqueleto abstracto de cada uno de los servicios del robot. Heredando el comportamiento de esta librería, cada uno de los nuevos servicios se ahorran procedimientos rutinarios como instanciar la conexión con la base de datos y el bróker de mensajería.

Solamente contiene dos interfaces que aprovecharán el resto de módulos. Comentémoslas a continuación:

service_messager.h

```
1   class ServiceMessager
2   {
3   public:
4           ServiceMessager(std::shared_ptr<lzr::msg::Broker> p_broker, std::
                shared_ptr<messages::MessageFactory> p_message_factory, std::
                string const &p_service_name)
5           {
6                   auto serialization = std::make_shared<lzr::msg::serialization
                        ::SerializationFactory>();
7                   m_pub = lzr::msg::createMsgPublisher(p_broker, serialization,
                         p_service_name);
8                   m_sub = lzr::msg::createMsgSubscriber(p_broker, serialization
                        , p_message_factory);
9           }
10
11          bool publish(std::string const &p_topic, messages::IBaseMessage const
                 &p_message)
12          {
13                  return m_pub->publish(p_topic, p_message);
14          }
15
16          void subscribe(std::string const &p_topic, std::string const &
                p_message, std::shared_ptr<lzr::msg::IBaseMessageReceiver>
                p_receiver)
17          {
18                  m_sub->subscribe(p_topic);
19                  m_sub->add_receiver(p_message, p_receiver);
20          }
21
22          auto get_pub() const { return m_pub; }
23          auto get_sub() const { return m_sub; }
24
25  private:
26          std::shared_ptr<lzr::msg::MsgPublisher> m_pub;
27          std::shared_ptr<lzr::msg::MsgSubscriber> m_sub;
28  };
```

Esta es una clase auxiliar para *ServiceBase* que envuelve las utilidades para publicar y suscribirse a mensajes fácilmente desde cualquier servicio. Utiliza dos objetos: *MsgPublisher* y *MsgSubscriber* que enlaza del módulo **messaging-broker** y que permite gestionar de forma sencilla la mensajería.

El método para **publicar** (*publish*) hace lo que se espera: simplemente enviar el mensaje. Sin embargo, el método ***subscribe***, además de suscribir el servicio correspondiente al tópico y mensaje de *PubSub* especificado, asocia un *callback* que será llamado cuando se publique dicho mensaje. Este ***IBaseMessageReceiver*** es otra interfaz de *messaging-broker* que define el comportamiento general de lo que se ha bautizado como *message receivers*, y que simplemente implementan la lógica ejecutada cuando se recibe el mensaje esperado.

service_base.h

```
class ServiceBase
{
public:
        virtual std::string get_name() = 0;
        virtual void init() = 0;
        virtual int run_internal() = 0;
        virtual void finish() = 0;

        int run()
        {
                // Abstract initialisation for communication
                init_messaging();

                // Initialisation steps given by child class
                init();

                std::cout << "Running service " << get_name() << std::endl;
                sd_notify(0, "READY=1");

                auto exit_code = run_internal();

                std::cout << "Stopping service " << get_name() << std::endl;
                sd_notify(0, "STOPPED=1");

                return exit_code;
        }

        bool publish(std::string const &p_topic, messages::IBaseMessage const
             &p_message)
        {
                return m_messager->publish(p_topic, p_message);
        }

        void subscribe(std::string const &p_topic, std::string const &
             p_message, std::shared_ptr<msg::IBaseMessageReceiver> p_receiver
             )
        {
                m_messager->subscribe(p_topic, p_message, p_receiver);
        }

private:
        void init_messaging()
        {
                auto broker = std::shared_ptr<lzr::msg::Broker>();

                constexpr auto CONNECTION_TIMEOUT{5s};
                auto const connected = broker->connect_async(
                     CONNECTION_TIMEOUT).get();
                if (!connected)
                {
                        throw std::runtime_error("Unable to connect to broker
```

```
                                .");
48                  }
49
50                  auto message_factory = std::make_shared<messages::
                        MessageFactory>();
51
52                  m_messager = std::make_shared<lzr::service::ServiceMessager>(
                        broker, message_factory, get_name());
53          }
54
55          std::shared_ptr<ServiceMessager> m_messager;
56  };
```

En esta clase abstracta *ServiceBase* ocurre todo el comportamiento genérico que utilizamos en los servicios que realmente se ejecutan en el robot. Si miramos el método ***run()*** distinguimos los siguientes pasos:

- **init_messaging()**: es un método privado que intenta realizar la conexión con el bróker de mensajería y compone los objetos de publicación/suscripción utilizando una **fábrica**, siguiendo el patrón de diseño ***Factory Method*** [45]. Éste facilita la creación de objetos concretos a partir de una interfaz común.
- **init()**: se trata de un método virtual que implementan las clases hijas para definir todo el comportamiento de inicialización que necesiten sus servicios.
- Llamadas a **sd_notify(...)**: es una función de la librería de *Systemd* y se utiliza para avisar al sistema operativo del estado actual del servicio en cuestión.
- **run_internal**: otro método virtual para que los servicios herederos implementen el *loop* de su funcionamiento real.

Llegados aquí, pasemos al fin a ver uno de los servicios que se ejecutan en el robot, concretamente el que se muestra en la pantalla táctil con la que puede interactuar el usuario.

lazarillo-hmi

Se trata de la aplicación que muestra la interfaz táctil al usuario. La tecnología usada por esta aplicación ha sido el framework de ***Qt***, que pese a tener modalidades de licencias de pago, permite un uso *open source* [40]. Además, se trata de una tecnología con una amplia comunidad, una documentación muy rica y un gran soporte para dispositivos embebidos. De hecho, *Qt* distribuye sus propios *layers* para poder utilizarlos en *Yocto* y así descargar automáticamente las librerías necesarias para nuestra aplicación.

Por otro lado, dispongo de amplia experiencia con dicha herramienta, lo que agiliza enormemente el tiempo de desarrollo. Utilizamos la versión **5.15.5** de la herramienta, versión *LTS* y última de la versión *major* número **5**. Si bien me habría gustado utilizar la **6**, que es la más novedosa y ya está en el mercado, me habría supuesto tener que ponerme a estudiar sobre ella y analizar todas las diferencias, lo cual requería un tiempo que no tenía.

El *framework* se compone a su vez de las librerías para *C++* y un lenguaje para implementación de interfaces gráficas basado en *JavaScript*, llamado ***QML***. Se puede obtener más información en su página web [46].

Empecemos viendo el archivo *CMakeLists.txt* con el que se describe lo necesario para poder compilar el servicio:

CMakeLists.txt

```
project(lazarillo-hmi VERSION 0.1 LANGUAGES CXX)

set(_include_dir inc/lazarillo-hmi)
set(TS_FILES i18n/lazarillo-hmi_es_ES.ts)

set(PROJECT_SOURCES
        ${_include_dir}/utils/style.h
        ${_include_dir}/service.h
        src/message_receivers/event_reboot_receiver.cpp
        src/message_receivers/event_reboot_receiver.h
        src/utils/style.cpp
        src/main.cpp
        src/service.cpp
        gui/qml.qrc
        gui/img.qrc
        ${TS_FILES}
)

add_executable(${PROJECT_NAME} ${PROJECT_SOURCES})

lzr_link_qt(${PROJECT_NAME})
lzr_link_systemd(${PROJECT_NAME})

qt5_create_translation(QM_FILES ${CMAKE_SOURCE_DIR} ${TS_FILES})

target_include_directories(${PROJECT_NAME} PUBLIC inc)
target_include_directories(${PROJECT_NAME} PRIVATE src)

target_link_libraries(${PROJECT_NAME} PRIVATE
        messages-definition
        messaging-broker
        serialization
        service-base
)
```

En la variable *PROJECT_SOURCES* se enumeran todos los ficheros fuente necesarios para el proyecto, tanto de *C++* como de *QML* (incluidos en ficheros del tipo *.qrc*), que luego se pasan como parámetro a *add_executable* para generar el binario de ejecución.

Las funciones *lzr_link_qt* y *lzr_link_systemd* utilizan el archivo cmake/ProjectSetup.cmake para enlazar con las librerías de *Qt* y *Systemd*, de forma similar a lo que se hace en *target_link_libraries* pero con los módulos internos del proyecto.

Pasemos ahora a ver código de la parte lógica en *C++* y la interfaz gráfica en *QML*. Empezaremos por ver el código que inicializa el servicio y lanza la interfaz, aprovechando para ilustrar sobre el uso de la clase abstracta ***ServiceBase*** explicado previamente:

service.h

```
class Service : public lzr::service::ServiceBase
{
```

```
3  public:
4          Service(int argc, char *argv[])
5          {
6                  m_app = new QGuiApplication(argc, argv);
7                  m_app->setOrganizationName("Adrian Morente");
8
9                  QTranslator translator;
10                 const QStringList uiLanguages = QLocale::system().uiLanguages
                       ();
11                 for (const QString &locale : uiLanguages)
12                 {
13                         if (translator.load(":/i18n/" + "lazarillo-hmi_" +
                               QLocale(locale).name()))
14                         {
15                                 m_app->installTranslator(&translator);
16                                 break;
17                         }
18                 }
19
20                 qmlRegisterUncreatableType<Style>("lazarillo.utils", 1, 0, "
                       Style", "Style enum type not creatable.");
21                 qRegisterMetaType<const Style *>("const Style");
22         }
23
24         ~Service() { delete m_app; }
25
26         std::string get_name() override { return "lazarillo-hmi"; }
27
28         void init() override
29         {
30                 auto event_reboot_receiver = std::make_shared<lzr::hmi::
                       EventRebootReceiver>();
31                 subscribe(WEB_GATEWAY_EVENT_TOPIC, messages::EventReboot::
                       NAME, event_reboot_receiver);
32         }
33
34         int run_internal() override
35         {
36                 QQmlApplicationEngine engine;
37                 engine.addImportPath(":/");
38
39                 Style style;
40                 style.changeSkin();
41                 engine.rootContext()->setContextProperty("style", &style);
42
43                 // Finally create QML window
44                 const QUrl url(QStringLiteral("qrc:/main.qml"));
45                 QObject::connect( &engine, &QQmlApplicationEngine::
                       objectCreated, m_app,
46                         [url](QObject *obj, const QUrl &objUrl)
47                         {
48                                 if (!obj && url == objUrl)
49                                 QCoreApplication::exit(-1);
50                         }, Qt::QueuedConnection);
51                 engine.load(url);
52
53                 return m_app->exec();
54         }
55
56 private:
57         QGuiApplication *m_app;
58 };
```

Esta clase *Service* instancia un objeto del tipo *QGuiApplication*, que es uno de los que provee *Qt* para utilizar aplicaciones mostradas en una ventana. En el constructor inicializa la aplicación, instala los archivos de traducciones (para poder mostrar la aplicación embebida en distintos idiomas), y expone el tipo *Style* de C++ a QML para poder ser usado allí.

El método *get_name()* informa al servicio abstracto sobre el nombre de esta

aplicación (que se usa en la interfaz para mostrar *logs* así como para el envío de mensajes a *PubSub*).

En *init_internal()*, método sobreescrito de la interfaz, se inicializa el ***receptor de mensaje*** para *EventReboot*. Como hemos explicado antes, estos receptores implementan un comportamiento específico que se ejecuta cuando se recibe el mensaje esperado en el tópico suscrito. En este caso, este receptor simplemente reinicia el dispositivo, lo que puede ser interesante realizar desde el portal web.

Y para terminar, *run_internal()* implementa el método abstracto que vimos en ***service-base*** que se llama en el *loop* del servicio. Lo que hace es cargar un motor de QML con el código situado en *main.qml* (que implementa la interfaz), asignarlo al objeto *m_app* visto antes y lanzar la aplicación.

Utilizando así el servicio base, el programa *main* de todos los servicios queda tan limpio como se ve a continuación:

main.cpp

```
#include "lazarillo-hmi/service.h"

int main(int argc, char *argv[])
{
        lzr::hmi::Service service(argc, argv);
        return service.run();
}
```

Simplemente carga el objeto *Service* visto arriba y llama al método *run()* de la clase heredada para aprovechar la inicialización de la mensajería.

Para terminar con los servicios embebidos en el robot, veamos ahora el que provee el servidor *WebSocket* al que se conecta el portal web.

web-gateway

Este servicio es el *puerto de entrada* a *Lazarillo* para la comunicación con el exterior. Como hemos dicho reiteradamente, realiza la comunicación mediante *WebSockets* con ***lazarillo-admin-frontend*** habilitando un servidor al efecto, el portal web a través del cual un técnico puede administrar los distintos robots. Actúa como intérprete de los mensajes provenientes del *socket* y los publica en la mensajería interna del robot (basada en *Redis*) para informar al resto de servicios de cualquier acción emitida por el servidor.

Además, como la comunicación basada en *WebSockets* es bidireccional y el canal siempre se mantiene activo, puede enviar mensajes al servidor en cualquier momento.

Dado que el fichero ***CMakeLists.txt*** de este módulo sigue una estructura igual al de ***lazarillo-hmi*** cambiando solo el nombre del proyecto y los ficheros fuente incluidos, nos ahorraremos la explicación, ya que no veremos nada nuevo.

En cuanto al archivo ***main.cpp*** que se ejecuta por defecto en el servicio, el código también es igual al de *lazarillo-hmi*, con la salvedad de que esta vez la clase *Service* (utilizada para sobreescribir la de *IBaseService*) es la implementación realizada por *web-gateway*. La única diferencia con respecto al servicio anterior reside en la inicialización, y es que al ser una aplicación sin interfaz gráfica que solo se ejecuta en segundo plano esperando solicitudes, solamente instancia un objeto del tipo propio ***WebsocketServer***. Veamos dicha clase:

websocket_server.h

```
1  class WebsocketServer : public QObject
2  {
3          Q_OBJECT
4
5  public:
6          explicit WebsocketServer(const int p_port  = 8080, QObject *p_parent
               = nullptr)
7                  : QObject{p_parent}, m_socket(new QWebSocketServer(
                       QStringLiteral("Lazarillo WS Server"),
8                  QWebSocketServer::NonSecureMode, this))
9          {
10                 if (m_socket->listen(QHostAddress::Any, p_port))
11                 {
12                         qDebug() << "Listening on port: " << p_port;
13
14                         connect(m_socket, &QWebSocketServer::newConnection,
                               this, &WebsocketServer::onNewConnection);
15                         connect(m_socket, &QWebSocketServer::closed, this, &
                               WebsocketServer::closed);
16                 }
17         }
18
19 private slots:
20         void onNewConnection()
21         {
22                 QWebSocket *client = m_socket->nextPendingConnection();
23
24                 connect(client, &QWebSocket::textMessageReceived, this, &
                       WebsocketServer::processTextMessage);
25                 connect(client, &QWebSocket::binaryMessageReceived, this, &
                       WebsocketServer::processBinaryMessage);
26                 connect(client, &QWebSocket::disconnected, this, &
                       WebsocketServer::socketDisconnected);
27
28                 qDebug() << "Client connected.";
29                 client->sendTextMessage(QString::fromUtf8("Connection
                       accepted from Websocket server"));
30         }
31
32         void processTextMessage(QString message)
33         qDebug() << "Text message received: " << p_message;
34
35         // parse and publish message to other services in web's topic
36         if (p_message == "REBOOT")
37         {
38                 m_service->publish(messages::topics::WEB_GATEWAY_EVENT_TOPIC,
39                 messages::EventReboot{});
40         }
41         else if (p_message == "OTHER_ACTION")
42         {
43                 // send other message related to OTHER_ACTION
44         }
45
46         void socketDisconnected() { qDebug() << "Client disconnected."; }
47
48 signals:
49         void closed();
50
51 private:
```

```
52          // Socket that holds the actual connection
53          QWebSocketServer *m_socket;
54  };
```

La clase solo contiene un objeto interno: un ***socket*** para el servidor mencionado antes. Se inicializa en el constructor, comienza a escuchar peticiones provenientes de cualquier origen y se conectan las señales *newConnection* y *closed* del *socket* interno a los métodos *onNewConnection* y *closed* respectivamente.

- ***onNewConnection***: para cada señal de nueva conexión al servidor, se instancia un nuevo objeto *QWebSocket* que hace referencia al cliente y se conectan las señales necesarias para cada evento de mensaje por su parte o desconexión. Estos mensajes se procesan en el método ***processTextMessage*** para parsearlos y enviarlos internamente a través del bróker de mensajería ya descrito extensamente.
- ***closed***: se ejecuta al recibir la señal del cliente al cerrar la conexión. Simplemente muestra un mensaje, pero podría también enviar notificaciones internas para que el resto de servicios sean conscientes (aunque en principio no debería ser de interés).

Éste es realmente todo el comportamiento del servicio. Es bastante sencillo ya que se sigue el objetivo de que los módulos cumplan **una sola responsabilidad**.

Ahora bien, *salgamos del robot* y del software embebido y pasemos a ver los servicios web externos.

4.3.2. lazarillo-admin-backend

Para volver a situarnos, véase la imagen 4.4 que muestra el diagrama de la arquitectura de todo el ecosistema. Esta vez hablaremos del servicio que hace las veces de **Intermediario**.

El objetivo que persigue es muy simple: facilitar a ***lazarillo-admin-frontend*** el descubrimiento de los robots que puede gestionar. Respecto a esto, podría llegar la pregunta de "¿Por qué es necesario que un robot se exponga como posible servidor para que el cliente se conecte? ¿No es más fácil fijar directamente la *URL* del robot (o lista de robots) en el servicio cliente y así siempre estará conectado?". La respuesta corta es: **por escalabilidad** (otra vez). Para la respuesta larga necesitamos más párrafos.

Cuando estamos realizando pruebas en nuestra máquina, donde la *URL* suele ser *localhost* y todos los servicios se ejecutan en el mismo sistema, es muy fácil interconectarlos. Sin embargo, en el mundo real las cosas no funcionan así: los robots se desplegarán físicamente por la universidad, el portal web se proporcionará desde un servidor externo y el administrador podría querer acceder a él desde su teléfono conectado a la red móvil.

Teniendo esto en cuenta, viene bien pensar en que sea el robot el que se exponga enviando a un pequeño servidor aparte la *URL* en la que el robot está disponible, para lo cual es necesario que la dirección de dicho servidor **sí** sea fija, o podría implementarse un método para modificarla a través de la pantalla táctil que incluye. El aspecto de que la dirección sea fija es fácil de solucionar desplegando el servicio en cualquier alojamiento al que se redirija a través de un nombre de dominio.

De esta forma, el portal web (*frontend*) obtiene periódicamente la lista de robots que han expuesto su dirección en el servidor mencionado y acto seguido, puede iniciar la conexión con ellos. Veamos la lógica implementada para satisfacer esto:

Generalidades del servicio

Como se ha adelantado, la implementación se ha realizado en el lenguaje *Python* en su versión 3, haciendo uso de la librería ***Flask*** [47] la cual permite crear servidores web de forma rápida y sencilla. Además, se ha utilizado la herramienta *Docker* para crear pequeños contenedores donde ejecutar la aplicación para hacer pruebas en local.

Por otro lado, para asegurar la **persistencia de los datos** utilizamos ***MongoDB*** [48], una base de datos no relacional que permite trabajar en formato *Json*, el cual es muy fácil de compartir con cualquier componente software. Además no requiere apenas configuración para empezar a usarse, por lo que es muy sencillo para pruebas rápidas.

Si observamos la estructura del proyecto en el repositorio, distinguimos varios elementos importantes:

- ***app/***: directorio que contiene el código de la aplicación, el fichero de dependencias de *Python* (*requirements.txt*), y un archivo *Dockerfile* que comentaremos posteriormente.
- ***mongodb/***: tan solo contiene un *Dockerfile* para crear un contenedor donde utilizar la base de datos, además de un simple script para configuración en el momento de la creación de dicho contenedor.
- *docker-compose.yml*: se trata de un formato de fichero concreto para la herramienta *Docker Compose* [49], la cual permite gestionar varios contenedores a la vez.

Gracias al uso de *Docker Compose*, con un simple comando *"docker-compose up"* podemos lanzar un contenedor para la aplicación en *Python* y otro para el servidor de base de datos (*MongoDB*) y comunicarlos entre sí para que el segundo sirva al primero. Veamos el código de configuración utilizado (se omiten algunos campos):

docker-compose.yml

```
1 version: '3'
```

```
2
3  services:
4          lazarillo_flask_backend:
5                  build:
6                          context: app
7                          dockerfile: Dockerfile
8                  image: lazarillo_flask_backend
9                  container_name: lazarillo_flask_backend
10                 environment:
11                         MONGODB_DATABASE: lazarillo
12                         MONGODB_USERNAME: mongouser
13                         MONGODB_PASSWORD: mongo_password
14                         MONGODB_HOSTNAME: lazarillo_mongodb_backend
15                 ports:
16                         - "5000:5000"
17                 depends_on:
18                         - lazarillo_mongodb_backend
19                 networks:
20                         - backend
21
22         lazarillo_mongodb_backend:
23                 build:
24                         context: mongodb
25                         dockerfile: Dockerfile
26                 image: lazarillo_mongodb_backend
27                 container_name: lazarillo_mongodb_backend
28                 command: mongod --auth
29                 environment:
30                         MONGO_INITDB_ROOT_USERNAME: mongouser
31                         MONGO_INITDB_ROOT_PASSWORD: mongo_password
32                         MONGO_INITDB_DATABASE: lazarillo
33                 networks:
34                         - backend
35
36 networks:
37         backend:
38                 driver: bridge
```

La configuración fija la versión de *Docker Compose* que se utiliza, lista los servicios que se quieren desplegar y las redes utilizadas para comunicarlos. Si bajamos un nivel de indentación dentro de cada servicio (*lazarillo_flask_backend* por un lado y *lazarillo_mongodb_backend* por otro), con *build/context/dockerfile* se especifica en qué directorio ha de buscarse el archivo *Dockerfile* para extraer la imagen del servicio en cuestión.

Con *image* y *container_name* se especifica el nombre con que se guardarán tanto la imagen de sistema como el contenedor creado. *environment* contiene variables de entorno que se crean con el valor asignado una vez que el contenedor resultante se inicia. En ambos servicios se establece la misma *network* (red) para que puedan comunicarse entre ellos e intercambiar información.

El resultado de descargar y construir las imágenes y ejecutarlas en contenedores se puede ver en la figura 4.6.

Una vez visto esto, pasemos al código en *Python* que recibe las peticiones (se obvian por ahora las funciones para explicarlas de forma más cómoda en el punto siguiente):

app/app.py

```
1 # Create app and connect to database
2 application = Flask(__name__)
```

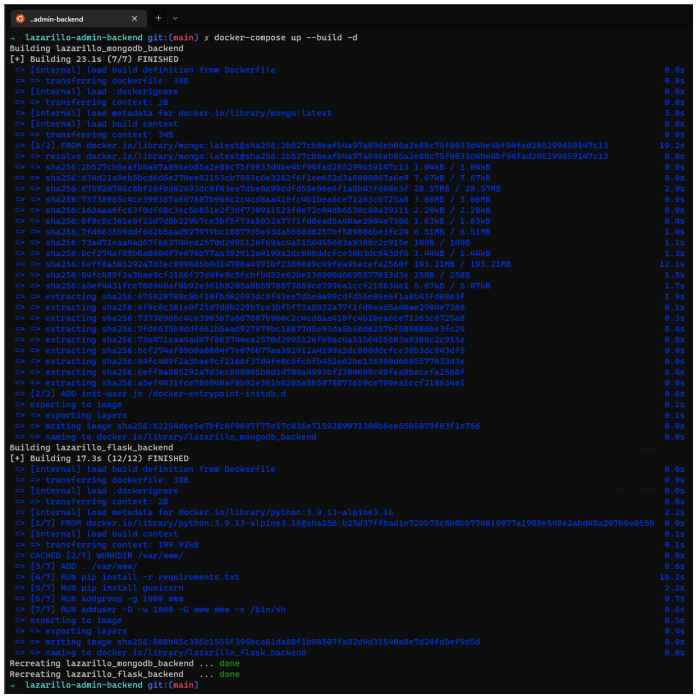

Figura 4.6: Captura de pantalla del comando de creación y ejecución de la imagen basada en *docker-compose*.

```
application.config["MONGO_URI"] = 'mongodb://' + \
        os.environ['MONGODB_USERNAME'] + ':' + os.environ['MONGODB_PASSWORD'] + \
        '@' + os.environ['MONGODB_HOSTNAME'] + \
        ':27017/' + os.environ['MONGODB_DATABASE']

mongo = PyMongo(application)
db = mongo.db


# Route to listen to broadcast messages from Lazarillo devices exposing their Websocket server
@application.route("/broadcast", methods=['POST'])
def read_device_broadcast():
        # code


# Route to receive the devices that exposed their WS server
@application.route("/devices", methods=['GET'])
def send_devices():
        # code


if __name__ == "__main__":
        ENVIRONMENT_DEBUG = os.environ.get("APP_DEBUG", True)
        ENVIRONMENT_PORT = os.environ.get("APP_PORT", 5000)
        application.run(host='0.0.0.0', port=ENVIRONMENT_PORT, debug=ENVIRONMENT_DEBUG)
```

El código básicamente inicializa la conexión al servicio de base de datos utilizando las variables de entorno que habíamos configurado en el fichero *docker-compose.yml* que vimos antes. Ese objeto *db* se utiliza en las rutas que descri-

biremos a continuación. Acto seguido, al final del código, se lanza la aplicación para que empiece a escuchar peticiones.

Rutas de acceso

La aplicación solo contempla dos rutas para recibir peticiones. La primera es ***/broadcast*** y solo acepta el método ***POST*** para escribir nuevos datos. Ésta es la ruta a la que llaman los robots para exponer sus datos y dirección de conexión, implementando la mecánica que explicamos en la sección anterior.

```
@application.route("/broadcast", methods=['POST'])
def read_device_broadcast():
        """Read parameters from device and store them into db"""
        name = request.form.get('name')
        version = request.form.get('version')
        address = request.form.get('address')

        # add device to list of websocket servers
        db.devices.insert_one({
                'name': name,
                'version': version,
                'address': address
        })
        return jsonify(message='success'), 201
```

Al recibir una llamada a ***/broadcast***, se leen los campos "nombre", "versión" y "dirección" de los parámetros especificados. Estos se componen en un documento con "*clave: valor*" y se insertan en la base de datos llamada *devices* usando el objeto *db* antes creado y la función *insert_one({})* para escribir un documento nuevo.

```
@application.route("/devices", methods=['GET'])
def send_devices():
        """Retrieve devices from database and send them as JSON"""
        device = {}
        response = []

        for item in db.devices.find():
        device = {
                'id': item['_id'].__str__(),
                'name': item['name'],
                'version': item['version'],
                'address': item['address']
        }
        response.append(device)

        return jsonify(status=True, data=response)
```

Por otro lado, la ruta ***/devices*** solo recibe métodos ***GET*** para lectura. De forma inversa a la escritura, lo que se hace aquí es leer **todas las entradas** de la base de datos con el método ***find()***. Todos los documentos leídos se incluyen en un *array* y se envían al solicitante.

En este caso el que realiza la llamada aquí es el servicio ***lazarillo-admin-frontend*** y así consigue leer los robots que existen y a los que puede conectarse para su correcta gestión.

Para terminar, comprobamos que nuestra aplicación se está ejecutando correctamente para esperar peticiones (ahora mismo se ejecutan en la máquina local utilizando la aplicación de escritorio de *Docker*), véase la figura 4.7. Se puede apreciar que el servidor web expone el puerto **5000** para escuchar ahí las peticiones.

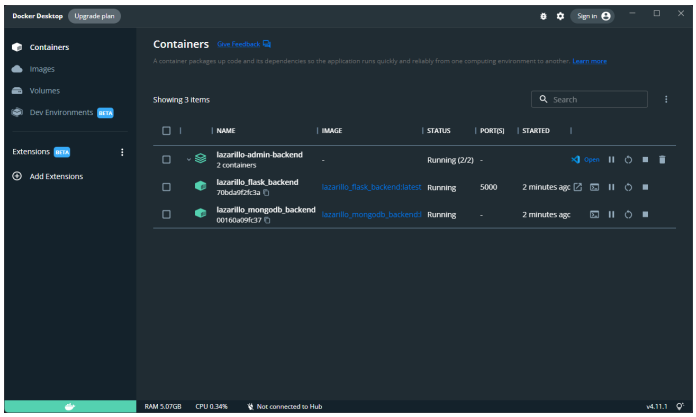

Figura 4.7: Captura de pantalla de *Docker Desktop* mostrando los dos contenedores del servidor web y la base de datos en ejecución.

Para terminar, pasemos a ver el servicio de *frontend* que aprovecha los datos aquí manejados.

4.3.3. lazarillo-admin-frontend

Finalmente llegamos a la aplicación que hará las veces de panel de administración para técnicos que deban administrar los robots, supervisar que todo está bien y, si procede, enviarles según qué comandos. La aplicación aquí descrita está implementada en ***React*** y ***Node/JavaScript*** utilizando la herramienta de creación de apps basadas en *React* llamada ***Create React App*** [50], que permite inicializar un proyecto con todo lo necesario tan solo lanzando un comando. Para poder usarla es necesario tener instalados los paquetes de *Node* y su gestor de paquetes: *NPM*.

Una vez que el comando ha terminado de preparar el proyecto, se habrá creado en la carpeta un archivo llamado *package.json* que sirve para describir la aplicación, listar dependencias y enumerar comandos útiles tanto para el desarrollo como para el despliegue. Veamos su contenido (obviando la parte de las dependencias ya que las comentaremos independientemente a continuación):

package.json

```
1 {
2         "name": "lazarillo-admin",
3         "version": "1.0.0",
4         "author": "Adrian Morente Gabaldon",
```

```
5         "license": "GPL-3.0-or-later",
6         "description": "Web server which provides administration tools for
              Lazarillo robots.",
7         "private": true,
8         "dependencies": { ... },
9         "scripts": {
10                "start": "react-scripts start",
11                "build": "react-scripts build",
12                "test": "react-scripts test"
13        },
14        "eslintConfig": {
15                "extends": [
16                        "react-app",
17                        "react-app/jest"
18                ]
19        },
20        "browserslist": {
21                "production": [
22                        ">0.2%",
23                        "not dead",
24                        "not op_mini all"
25                ],
26                "development": [
27                        "last 1 chrome version",
28                        "last 1 firefox version",
29                        "last 1 safari version"
30                ]
31        }
32 }
```

Como vemos los primeros campos son meramente descriptivos sobre el proyecto en cuestión, mientras que el de "*scripts*" habilita cuatro llamadas del tipo *npm start* para diversas acciones:

- **start**: compila el proyecto y lanza un servidor local que ofrece la web implementada.
- **build**: realiza una compilación optimizada y ofuscada para ser desplegada en *producción*.
- **test**: ejecuta los programas de tests que tenga el proyecto, si los hubiere.

Por otro lado, para llegar a ofrecer una página web usable, adaptable y con una interfaz agradable que siga las guías de estilo de lo que estamos acostumbrados a ver, hacemos uso de la librería ***Material UI (MUI)*** [51]. Contiene componentes web ya preparados para su uso y fáciles de parametrizar para nuestro interés. Estas piezas de software se componen de lenguajes de marcado *HTML* y *CSS* para el esqueleto y el diseño, además del lenguaje de programación *JavaScript* para la lógica.

Adicionalmente, para seguir buenas prácticas y trabajar de acuerdo con el paradigma de *React*, en lugar de guardar los datos de interés en los objetos que implementamos, se utiliza un almacén para manejo y persistencia de estados llamado ***Redux*** [52]. Hacer ésto evita que el usuario de la web se vea obligado a actualizar la página y que ésta solo renderice automáticamente los cambios en los componentes cuando cambie su estado interno.

Estructura del proyecto

Si miramos la lista de directorios y ficheros en el repositorio correspondiente encontramos lo siguiente:

- ***public/***: directorio donde se generan los ficheros que realmente se sirven a los clientes que acceden a la web. El principal es *index.html* que es la página mostrada, el resto suelen ser imágenes y demás recursos necesarios.
- ***src/***: esta carpeta incluye el código desarrollado para esta implementación. Lo comentaremos más detenidamente más adelante.
- ***Dockerfile***: al igual que en el proyecto de *backend*, proveemos un fichero *Dockerfile* definiendo una imagen que poder desplegar la aplicación en un contenedor *Docker*. Éste es más simple que en el otro subproyecto ya que solo inicia un servicio en lugar de dos.
- ***docker-compose.yml***: siguiendo lo comentado arriba, con la configuración descrita en este archivo se inician el contenedor solicitado.
- *package.json*: (comentado anteriormente).

Empecemos por comentar el código implementado para después pasar a ver *la Dockerización* (palabra que en castellano no existe pero que sería la traducción literal de "dockerize", término inventado y muy usado en internet para hacer referencia al acto de embeber una aplicación en un contenedor de *Docker*).

Código implementado

En la carpeta *src* encontramos la siguiente estructura que ordena los distintos ficheros fuente:

- ***components/***: la carpeta de componentes contiene los elementos independientes que pueden ser renderizados en pantalla por *React*.
- ***redux/***: contiene el código referente a la herramienta *Redux* para gestionar los estados de los datos que se manejan en el proyecto, que en este caso solo son los dispositivos *Lazarillo*.
- *App.js*: es el ejecutable de la aplicación. En él se declaran las distintas rutas que contiene la web.
- *ViewLayer.jsx*: es un panel auxiliar del que heredan los componentes mostrados en pantalla para utilizar el mismo estilo.

Empecemos por ver el código del principal, *App.js*:

```
1  const appTheme = createTheme({
2          palette: {
3                  mode: 'dark',
4          },
5  });
6
7  function App() {
8    return (
9      <BrowserRouter>
10       <ThemeProvider theme={appTheme}>
11         <CssBaseline>
```

```
          <Box sx={{ display: 'flex' }}>
            <Provider store={store}>
              <Header />
              <Routes>
                <Route exact path="/" element={<Robots />} />
                <Route exact path="/about" element={<AboutView />} />
              </Routes>
            </Provider>
          </Box>
        </CssBaseline>
      </ThemeProvider>
    </BrowserRouter>
  );
}

export default App;
```

Se instancia el objeto *App* añadiendo diversos componentes que explicaremos por separado:

- ***BrowserRouter***: habilita que se puedan usar rutas dentro de la página.
- ***ThemeProvider***: hace que la web se muestre con el tema definido arriba (*appTheme*, el cual establece el modo oscuro).
- ***Provider***: relaciona la aplicación con el almacén de datos y estados basado en *Redux* que antes anunciamos.
- ***Header***: se trata de un componente propio definido en la carpeta *components/*, que incluye la barra de aplicación superior de la página y el ***drawer*** (o panel lateral).
- ***Route***: cada objeto *route* añade una nueva ruta a la web. En este caso solo hay dos: la principal (*/*) y la de información (*/about*). Cada una se asocia a un componente, en este caso **Robots** y **AboutView** respectivamente, ambos en la carpeta *components*.

Visto esto, centrémonos en los componentes interesantes que enrutamos. Por un lado la lista de dispositivos en *Robots* y en otro la de *AboutView* para mostrar una información muy básica sobre el proyecto que crecería con el tiempo.

components/Robots.js

```
const Robots = () => {
  const dispatch = useDispatch();
  const { robots, updatedRobots } = useSelector((state) => state.robots);

  if (!updatedRobots) {
    fetch("http://localhost:5000/devices").then((res) =>
      res.json()
    ).then((data) => {
      dispatch(fetchRobotsList(data.data));
    });
  }

  return (
    <ViewLayer>
      <Paper>
        <Grid container alignItems="center" padding={2}>
          {
            robots.map((item, index) =>
              <Grid item key={index}>
                <RobotComponent key={index} item={item}></RobotComponent>
```

```
              </Grid>
            )
          }
        </Grid>
      </Paper>
    </ViewLayer>
  );
}
export default Robots;
```

En primer lugar, se inicializa el *dispatcher* para guardar en el almacén de *Redux* la lista de dispositivos. Se hace la llamada al *endpoint* de ***lazarillo-admin-backend*** para obtener la colección de robots que ha de mostrar al administrador en la web. **Nota**: algo que queda por arreglar aquí es que la solicitud se haga periódicamente con un temporizador.

Para terminar, se itera sobre la lista para añadir un objeto *Paper* a la cuadrícula.

Para ilustrar cómo funciona el almacén de *Redux*, necesitamos ver dos clases: un ***reducer***, que actúa como organizador de los datos que recibe para que puedan ser guardados después, y una ***store*** que los almacena realmente.

redux/robotReducer.js

```
import { createSlice, PayloadAction } from '@reduxjs/toolkit'

const initialState = {
  robots: [],
  updatedRobots: false,
}

const robotsSlice = createSlice({
  name: 'robots',
  initialState,
  reducers: {
    fetchRobotsList(state, action) {
      state.robots = action.payload;
      state.updatedRobots = true;
      console.log("Robots list: ", state.robots);
    }
  }
})

export const { fetchRobotsList } = robotsSlice.actions;
export default robotsSlice.reducer;
```

Utilizando el set de herramientas de *Redux* se crea un ***slice***, que es una "porción de datos" de las que se guardan en el almacén. El estado de arranque (representado por *initialState*) inicializa la lista a un *array* vacío, y utiliza un valor booleano para controlar si se necesita realizar una nueva petición de lectura de dispositivos. Con el temporizador que hemos mencionado antes que podría añadirse, aquí se automatizaría dicha lectura.

El método *fetchRobotsList* es llamado desde el componente ***Robots*** para asignar el valor al documento cuando la lectura ha sido exitosa. Hecho esto, se exporta el *reducer* para utilizarse en la *store* que vemos a continuación:

redux/store.js

```
import { configureStore } from "@reduxjs/toolkit";
import robotReducer from "./robotReducer";

export const store = configureStore({
  reducer: {
    robots: robotReducer,
  }
})
```

Esta clase es muy sencilla pero eficaz. Gracias a *Redux* basta con llamar al método *configureStore* para crear un almacén con los tipos de dato descritos por los *reducers* especificados por parámetros del documento.

Volviendo a las vistas, veamos ahora la sencilla pestaña de información:

components/AboutView.js

```
const AboutView = () => {
  return (
    <ViewLayer>
      <Paper elevation={8} sx={{ m: 1, p: 2, width: 250 }}>
        <Grid container>
          <Grid item xs={12}>
            <Typography variant="h4">Proyecto: Lazarillo Admin Frontend</
                Typography>
          </Grid>
          <Grid item xs={4}>
            <Typography variant="h5">Realizado por:</Typography>
          </Grid>
          <Grid item xs={8}>
            <Typography variant="h5">Adrian Morente</Typography>
          </Grid>
          <Grid item xs={12}>
            <Typography variant="h6">ETSIIT - Universidad de Granada</
                Typography>
          </Grid>
        </Grid>
      </Paper>
    </ViewLayer>
  );
}
export default AboutView;
```

Solamente muestra en el componente central de la pantalla unos textos diciendo el autor del proyecto. Una idea futura sería automatizar la versión actual del software con cada *release* nueva para que se muestre aquí.

Aunque en la fase alcanzada del desarrollo es una versión muy simple, para ver cómo luce actualmente la página web implementada, véase la figura 4.8.

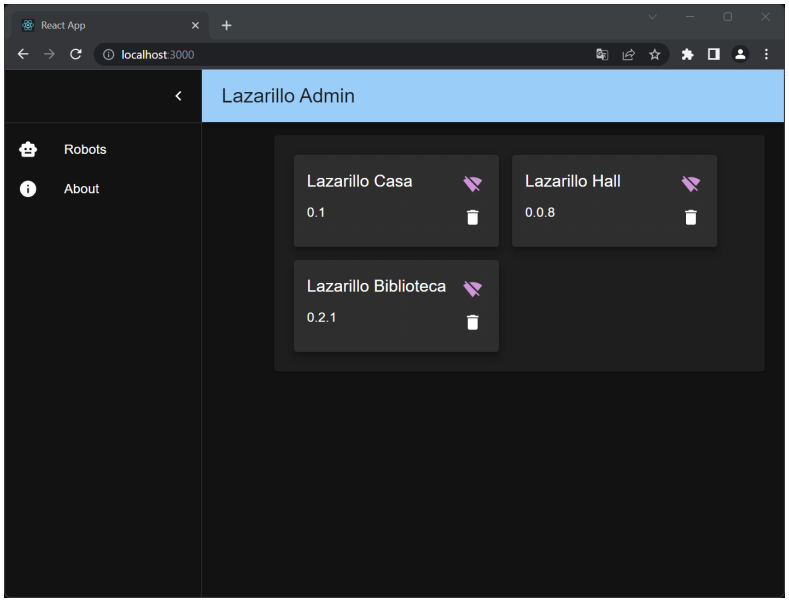

Figura 4.8: Captura de pantalla de ***lazarillo-admin-frontend*** mostrando el panel lateral y tres dispositivos no conectados.

Capítulo 5

Trabajos futuros

Un buen *brainstorming* sobre el desarrollo de una idea novedosa y (se supone que) útil siempre es motivador y consigue animar a un programador a llevar la idea a la práctica, intentando la consecución del mayor número de requisitos posibles. Sin embargo, el tiempo siempre es uno de los factores más determinantes a la hora de dictaminar cómo de lejos se llega con el proyecto.

Dicho esto, en este capítulo comentaremos cuáles de las ideas iniciales no llegaron a finalizarse a tiempo para la entrega del proyecto, y cuáles se decidieron postergar en el tiempo para un futuro desarrollo.

Cabe destacar que dada la naturaleza libre del proyecto, cualquier interesado o interesada podría continuar con la parte del desarrollo que más le interese; ya que esta idea nació como una herramienta abierta a la que poder contribuir y de la que cualquiera pueda sacar utilidad.

Entrando ya en materia, dividiremos este capítulo en breves secciones relacionadas con las ya vistas en el capítulo de **Implementación**, detallando los cabos sueltos que hayan quedado en cada uno de ellos.

5.1. Sistema operativo

5.2. Servicios propios

5.3. Portal web de administración

5.4. Otros

TODO: rellenar secciones (tras terminar capítulo de Implementación)

Capítulo 6

Conclusiones

Bibliografía

[1] IEEE.org, "IEEE - Advancing Technology for Humanity." Enlace: https://www.ieee.org/.

[2] IEEE - Robots, "What Is a Robot? Top roboticists explain their definition of robot." Enlace: https://robots.ieee.org/learn/what-is-a-robot/.

[3] iRobot, "Robots aspiradores Roomba." Enlace: https://www.irobot.es/es_ES/roomba.html.

[4] D. Mukherjee, A. Saha, P. Mendapara, D. Wu, and Q. M. J. Wu, "A cost effective probabilistic approach to localization and mapping," 07 2009.

[5] SoftBank Robotics, "Pepper le robot humanoïde et programmable." Enlace: https://www.softbankrobotics.com/emea/es/pepper.

[6] Noticias de Navarra, "Suspendida la producción del robot japonés Pepper por la falta de demanda." Enlace: https://www.noticiasdenavarra.com/ciencia-y-tecnologia/2021/06/29/suspendida-produccion-robot-japones-pepper-2131197.html.

[7] Tesla, "Artificial Intelligence and Autopilot." Enlace: https://www.tesla.com/AI.

[8] Arduino Store, "Arduino Boards." Enlace: https://store.arduino.cc/collections/boards.

[9] Balara, "Zowi BQ, el robot didáctico de Clan." Enlace: https://www.balara.es/zowi-robot-bq-clan/.

[10] Raspberry Pi, "Products store." Enlace: https://www.raspberrypi.com/products/.

[11] NXP, "i.MX Applications Processors." Enlace: https://www.nxp.com/products/processors-and-microcontrollers/arm-processors/i-mx-applications-processors:IMX_HOME.

[12] Asus.com, "Tinker Board." Enlace: https://www.asus.com/es/Networking-IoT-Servers/AIoT-Industrial-Solutions/All-series/Tinker-Board/.

[13] Raspberry Pi, "Raspbian - Operating system images." Enlace: https://www.raspberrypi.com/software/operating-systems/.

[14] Buildroot, "Making Embedded Linux Easy." Enlace: https://buildroot.org/.

[15] The Yocto Project, "It's not an embedded Linux distribution, it creates a custom one for you.." Enlace: https://www.yoctoproject.org/.

[16] Mads Doré Hansen, "Yocto or Debian for Embedded Systems," 08 2017. Enlace: https://www.prevas.dk/download/18.58aaa49815ce6321a327da/1506087244328/Yocto_Debian_Whitepaper.pdf.

[17] Codete, "8 top programming languages for robotics." Enlace: https://codete.com/blog/top-8-programming-languages-for-robotics.

[18] ROS.org, "Why ROS?." Enlace: https://www.ros.org/blog/why-ros/.

[19] NIRYO, "8 reasons why you should use ROS for robotics projects." Enlace: https://service.niryo.com/en/blog/8-reasons-use-ros-robotics-projects.

[20] Digital Ocean, "SOLID: Los primeros 5 principios del diseño orientado a objetos." Enlace: https://www.digitalocean.com/community/conceptual_articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design-es.

[21] GeeksforGeeks, "Inter Process Communication (IPC)." Enlace: https://www.geeksforgeeks.org/inter-process-communication-ipc/.

[22] ROS.org, "Writing a Simple Publisher and Subscriber (C++)." Enlace: http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28c%2B%2B%29.

[23] Amazon AWS, "Pub/Sub Messaging." Enlace: https://aws.amazon.com/es/pub-sub-messaging/.

[24] LuisLlamas.es, "¿Qué es MQTT." Enlace: https://www.luisllamas.es/que-es-mqtt-su-importancia-como-protocolo-iot/.

[25] Redis.io, "Introduction to Redis." Enlace: https://redis.io/docs/about/.

[26] Redis.io, "How to use pub/sub channels in Redis." Enlace: https://redis.io/docs/manual/pubsub/.

[27] Mozilla Developer, "HTTP Protocol," 2022.

[28] ResearchHubs.com, "The Hypertext Transfer Protocol (HTTP)." Enlace: https://researchhubs.com/post/computing/web-application/the-hypertext-transfer-protocol-http.html.

[29] Mozilla Developer, "WebSockets Protocol," 2022.

[30] Wallarm.com, "A simple explanation of what a WebSocket is." Enlace: https://www.wallarm.com/what/a-simple-explanation-of-what-a-websocket-is.

[31] Github.com, "GitHub pricings." Enlace: https://github.com/pricing.

[32] Adrián Morente Gabaldón - Github.com, "lazarillo-embedded - Github.com." Enlace: https://github.com/adrianmorente/lazarillo-embedded.

[33] Adrián Morente Gabaldón - Github.com, "lazarillo-admin-frontend - Github.com." Enlace: https://github.com/adrianmorente/lazarillo-admin-frontend.

[34] Adrián Morente Gabaldón - Github.com, "lazarillo-admin-backend - Github.com." Enlace: https://github.com/adrianmorente/lazarillo-admin-backend.

[35] Adrián Morente Gabaldón - Github.com, "meta-lazarillo - Github.com." Enlace: https://github.com/adrianmorente/meta-lazarillo.

[36] Adrián Morente Gabaldón - Github.com, "documentacion-tfm - Github.com." Enlace: https://github.com/adrianmorente/documentacion-tfm.

[37] GNU Operating System, "GNU General Public License." Enlace: https://www.gnu.org/licenses/gpl-3.0.html.

[38] Yocto Project, "Yocto Project Quick Build." Enlace: https://docs.yoctoproject.org/brief-yoctoprojectqs/index.html.

[39] Yocto Project, "Yocto Releases." Enlace: https://wiki.yoctoproject.org/wiki/Releases.

[40] Qt.io, "Qt for Open Source Development." Enlace: https://www.qt.io/download-open-source.

[41] g0hl1n, "psplash - Very simple boot splash screen." Enlace: https://github.com/g0hl1n/psplash.

[42] openembedded.org, "OpenEmbedded." Enlace: https://www.openembedded.org/wiki/Main_Page.

[43] CMake.org, "Cmake." Enlace: https://cmake.org/.

[44] Clang LLVM, "Clang 16.0.0git documentation." Enlace: https://clang.llvm.org/docs/ClangFormat.html.

[45] ReactiveProgramming.io, "Factory Method - Creational pattern." Enlace: https://reactiveprogramming.io/blog/en/design-patterns/factory-method.

[46] doc.qt.io, "Qt QML Documentation." Enlace: https://doc.qt.io/qt-5/qtqml-index.html.

[47] Flask - PalletsProjects, "Flask - Web development, one drop at a time." Enlace: https://flask.palletsprojects.com/en/2.2.x/.

[48] MongoDB, "MongoDB Documentation." Enlace: https://www.mongodb.com/docs/.

[49] Docker, "Docker Compose Docs." Enlace: https://docs.docker.com/compose/.

[50] Create React App, "Set up a modern web app by running one command." Enlace: https://create-react-app.dev/.

[51] MUI, "Material UI - Overview." Enlace: https://mui.com/material-ui/getting-started/overview/.

[52] Redux, "Redux · Readme." Enlace: https://es.redux.js.org/.