UNIVERSIDAD DE MÁLAGA

GRADO EN INGENIERÍA INFORMÁTICA

Gemelo digital para componentes del robot Husky
Digital twin for Husky robot components

Realizado por

Daniel Cerezo García

Tutorizado por

Enrique Soler Castillo

Julia Robles Medina

Departamento

Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA

MÁLAGA, JUNIO DE 2022

Agradecimientos

A mi tutor Enrique Soler Castillo por la confianza y oportunidad que me ha ofrecido, por su disposición, enseñanza y aporte del material necesario. Agradecer de la misma manera a Julia Robles Medina por su ayuda y apoyo constante, que se ha comportado como una segunda tutora.

Al equipo de trabajo que conforma el grupo ERTIS, por su compañerismo y ayuda siempre que lo he necesitado.

A Rocío, por su inconmensurable paciencia durante este recorrido, por ayudarme a manejar los problemas que surgieron durante la carrera y por su total confianza en mí.

A mis colegas, por estar siempre al pie del cañón y sacarme unas risas cuando más lo necesitaba.

Por último, quiero agradecer a mi familia por sus años de apoyo incondicional y comprensión, por enseñarme que todo sacrificio tiene una recompensa y por apoyarme en los momentos difíciles.

Muchas gracias a todos por hacer realidad este Trabajo de Fin de Grado.

Resumen

Vivimos en una sociedad en el que la digitalización, las nuevas tecnologías y la transformación digital están a la orden del día, ya que gracias a estas herramientas somos capaces de avanzar en la industria de una forma que nunca antes hubiésemos imaginado. La entrada de nuevas tecnologías y conceptos, como pueden ser la *IA* (Inteligencia Artificial), el *IoT* (Internet of things) o el *Machine Learning*, ha permitido simplificar, optimizar y ser más eficiente en los proyectos.

En este contexto aparece el concepto de gemelo digital (*Digital Twin* en inglés), clave en la cuarta revolución industrial y que consiste en la representación de un objeto del mundo real en uno virtual con el objetivo de aprovechar los datos de sensores para mejorar procesos y arreglar problemas. Dentro de esta idea se encuentra un concepto llamado sombra digital (*Digital Shadow)*, que digitaliza procesos reales y utiliza el modelado y la simulación de procesos para crear una copia de la realidad lo más idéntica posible.

Este trabajo se basa en la creación de una sombra digital del robot Husky perteneciente al grupo ERTIS de la UMA. Se desplegarán los servicios de *Eclipse Ditto* y *Eclipse Hono* en un clúster local de *Kubernetes* con *Minikube*, que serán los encargados de recibir los datos de los componentes del robot para crear la sombra digital. El manejo del robot se consigue gracias al sistema operativo *ROS* que lleva implementado y que permite acceder a los datos de cada uno de sus componentes. La comunicación del objeto real con el digital se realiza mediante *MQTT*, un protocolo de comunicación estándar para aplicaciones IoT. Por último, se ha creado un modelo 3D del robot Husky en la plataforma de *Unity* para representarlo visualmente junto a los datos de cada componente en *Grafana*.

Palabras clave:

Gemelo digital, Sombra digital, ROS, Eclipse Ditto, Eclipse Hono

Abstract

We live in a society in which digitization, new technologies and digital transformation are the order of the day, since thanks to these tools we are able to advance in the industry in a way that we had never imagined before. The entry of new technologies and concepts such as AI (Artificial Intelligence), IoT (Internet of things) or Machine Learning allows us to simplify, optimize and be more efficient in projects.

In this context, the Digital Twin concept appears, a key concept in the fourth industrial revolution, which consists of the representation of a real world object in a virtual one in order to take advantage of sensor data to improve processes and fix problems. Within this idea is a concept called Digital Shadow, which digitizes real processes and uses process modeling and simulation to create a copy of reality that is as identical as possible.

This work is based on the creation of a digital shadow of the Husky robot belonging to the ERTIS group of the UMA. Eclipse Ditto and Eclipse Hono services will be deployed in a local Kubernetes cluster with Minikube, which will be in charge of receiving the data from the robot components to create the digital shadow. The management of the robot is achieved thanks to the implemented ROS operating system, which allows access to the data of each of its components. The communication of the real and digital object is done through MQTT, a standard communication protocol for IoT applications. Finally, a 3D model of the Husky robot has been created in the Unity platform to represent it visually together with the data of each component in Grafana.

Keywords:

Digital twin, Digital shadow, ROS, Eclipse Ditto, Eclipse Hono

Índice de contenido

1. Introducción **1**

1.1 Motivación **1**

1.2 Planteamiento del problema **3**

1.3 Objetivos **4**

1.4 Metodología **5**

1.5 Estructura de la memoria **6**

2. Descripción del sistema **9**

2.1 Robot Husky **10**

2.1.1 Conociendo el sistema ROS 12

2.1.2 Conclusiones de ROS 15

2.2 Creación de la sombra digital **15**

2.2.1 Instalación de forma local 16

2.2.2 Despliegue Eclipse Ditto y Eclipse Hono 16

2.2.3 Transmisión de datos de los componentes de Husky 18

2.2.4 Visualización de datos 19

2.2.5 Problemas al exportar los servicios a internet del clúster 22

2.3 Creación del modelo 3D **23**

2.3.1 Creación de objetos en Blender 25

2.3.2 Modelo final del robot Husky en 3D 26

2.4 Uniendo las partes 27

2.5 Requisitos hardware 28

3. Tecnologías utilizadas 29

3.1 Docker 29

3.2 Kubernetes 30

3.3 Minikube 32

3.4 Eclipse Ditto 33

3.5 Eclipse Hono 34

3.6 Eclipse Mosquitto y Eclipse Paho 35

3.7 Apache Kafka 37

3.8 InfluxDB 37

3.9 Grafana 38

3.10 ROS (Robotic Operating System) 39

3.11 Visual Studio Code 40

3.12 Ubuntu Linux 40

3.13 Python 41

3.14 Unity 41

3.15 Blender y 3D Builder 42

4. Desarrollo del proyecto 43

4.1 Requisitos previos para realizar los pasos 43

4.2 FASE 1: Instalación de Minikube 44

4.2.1 Instalación del Backend 44

4.2.2 Conociendo el backend 47

4.3 FASE 2: Configurando el robot Husky 49

4.3.1 Comunicación con el robot Husky 50

4.3.2 Conectando a internet al robot Husky 54

4.3.3 Creando el script para enviar datos 55

4.3.4 Como ejecutar el script para enviar datos desde Husky 60

4.4 FASE 3: Creación de la sombra en el clúster local 61

4.4.1 Registrar el dispositivo 61

4.4.2 Crear la sombra digital de Husky 62

4.5 FASE 4: Consulta de la sombra digital 64

4.6 FASE 5: Enviar datos al clúster 65

4.7 FASE 6: Representar datos en Grafana 67

4.7.1 Despliegue de Apache Kafka 68

4.7.2 Despliegue de InfluxDB y Telegraf 69

4.7.3 Despliegue de Grafana 70

5. Conclusiones y líneas futuras 73

5.1 Desarrollo del proyecto 73

5.2 Líneas futuras 75

6. Referencias 77

Apéndice A. Manual de Usuario 83

Apéndice B. Creación de objetos 3D en Blender 85

Índice de figuras

Figura 1. Diferencias entre el modelo digital, la sombra digital y el gemelo digital [3] 2

Figura 2. Robot Husky 10

Figura 3. Comunicación de nodos en ROS 13

Figura 4. Funcionamiento del nodo maestro 14

Figura 5. Arquitectura inicial open-source [21] 16

Figura 6. Descripción general del paquete cloud2edge 17

Figura 7. Arquitectura del proyecto 21

Figura 8. Modelo inicial del robot Husky en 3D 24

Figura 9. Escáner láser LMS111 en 3D 25

Figura 10. Tapa chasis superior en 3D 26

Figura 11. Router Wifi en 3D 26

Figura 12. Modelo final del robot Husky en 3D 27

Figura 13. Logotipo de Docker 30

Figura 14. Logotipo de Kubernetes 32

Figura 15. Logotipo de Minikube 33

Figura 16. Logotipo Eclipse Ditto 34

Figura 17. Logotipo de Eclipse Hono 35

Figura 18. Logotipo del protocolo MQTT que engloba ambas tecnologías 37

Figura 19. Logotipo de Apache Kafka 37
Figura 20. Logotipo de InfluxDB 38
Figura 21. Logotipo de Grafana 39
Figura 22. Logotipo del sistema ROS 40
Figura 23. Logotipo de Visual Studio Code 40
Figura 24. Logotipo de Ubuntu Linux 41
Figura 25. Logotipo de Python 41
Figura 26. Logotipo de Unity 42
Figura 27. Logotipo de Blender 42
Figura 28. Versiones de kubectl similares 45
Figura 29. Resultado tras instalar el paquete cloud2edge 47
Figura 30. Servicios de cloud2edge desplegados en Minikube 48
Figura 31. Lista de tópicos del robot Husky 51
Figura 32. Datos del tópico /status 52
Figura 33. Datos del tópico /scan 53
Figura 34. Datos del tópico /join_states 53
Figura 35. Datos del tópico /gps/fix 54
Figura 36. Cabecera script 55
Figura 37. Constantes de conectividad MQTT 56
Figura 38. Función para crear la conexión MQTT. 57
Figura 39. Función que se suscribe a un nodo publicador 57

Figura 40. Función que recoge los datos del nodo publicador y lo estructura para enviarlo como JSON 58

Figura 41. Función que muestra lo que está enviando el script 59

Figura 42. Datos que contiene el tópico */status* [52] 59

Figura 43. Sensores de la sombra junto a los datos recibidos 65

Figura 44. Envío de datos del tópico de ROS a nuestro clúster. 67

Figura 45. Plataforma influxDB desplegada y recibiendo datos. 70

Figura 46. Representación en Grafana de la sombra digital sin recibir datos. ...71

Figura 47. Representación en Grafana de la sombra digital recibiendo datos 72

1

Introducción

En este documento se hará un acercamiento a la industria 4.0 permitiendo al lector adentrarse en el mundo del gemelo digital y su gran utilidad como herramienta de soporte al desarrollo continuo.

1.1 Motivación

El concepto de gemelo digital tiene su origen en 2002 en la Universidad de Michigan y consiste en crear una imitación digital de un sistema real, el cual será considerado su gemelo [1]. Dentro del gemelo digital se encuentra un concepto llamado sombra digital, que digitaliza procesos reales y utiliza el modelado y la simulación de procesos para crear una copia de la realidad lo más idéntica posible [2]. La diferencia entre estos dos conceptos es que la sombra digital no envía información ni actúa sobre el sistema real, en cambio, el gemelo digital sí permite una comunicación bidireccional en la que este sí puede afectar en el comportamiento del sistema.

En la Figura 1, se pueden apreciar visualmente las diferencias entre cada uno. La sombra digital es menos compleja de implementar, ya que es una etapa

anterior que debe modelarse antes de llegar al desarrollo completo del propio gemelo digital. Así, la idea principal de este trabajo se centrará en el concepto de sombra digital.

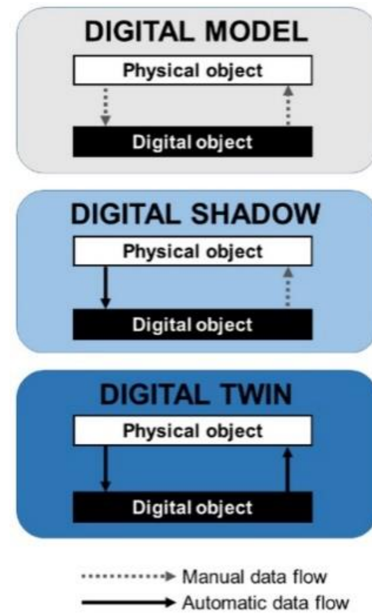


Figura 1. Diferencias entre el modelo digital, la sombra digital y el gemelo digital [3]

El grupo de investigación ERTIS de la UMA dispone de un robot Husky [4], la primera plataforma robótica terrestre operada mediante el sistema operativo ROS [5] desde su configuración de fábrica. Además, es posible agregarle distintos módulos, como pueden ser cámaras, LIDAR (Light Detection and Ranging) o GPS. Debido a su gran abanico de posibilidades de explotación, han sido publicados numerosos trabajos utilizando Husky como objeto principal, sirviendo como punto de referencia para establecer nuevos retos de investigación y desarrollo de otros robots.

Realizar una sombra digital del robot es un primer paso que permite el conocimiento e integración de diversas tecnologías. Además, se podrá contar con un entorno muy versátil de toda la información de la que dispone y genera el robot. Esto abre un gran abanico de posibilidades, desde hacer simulaciones

complejas sin poner en riesgo el robot y su entorno hasta desarrollar y probar todo tipo de aplicaciones que saquen partido del robot.

1.2 Planteamiento del problema

En la industria de los robots, los métodos de programación se pueden clasificar como fuera de línea o en línea. El primer tipo consiste en programar y simular el trabajo del robot en un software de programación para robótica. La programación en línea, por otro lado, implica detener el robot y usar su dispositivo de aprendizaje para crear el programa. [6]

La mayor parte del trabajo de programación a menudo utiliza el método fuera de línea, ya que permite reducir los tiempos de inactividad, acelera la integración del robot y mejora continuamente el programa del robot sin afectar a la productividad. Sin embargo, al exportar el programa al robot real, no siempre se da el caso de que todo funcione con precisión.

La puesta a punto es una *"tarea importante en todo el ciclo de vida del sistema de producción de ingeniería"* [3] , que suele ser necesario cuando se utiliza la programación en línea. Lamentablemente, este paso implica manejar no solo un robot industrial real con sus herramientas, sino también, a veces, el objeto de trabajo que, dependiendo de la tarea, pueden ser costosas e incluso peligrosas para el usuario que las maneja.

La combinación de una sombra digital del robot junto a una visualización 3D puede ayudar a aumentar la seguridad y, eventualmente, incluso hacer que el proceso de diseño sea más rápido. Las sombras digitales de herramientas y objetos de trabajo se pueden crear y superponer en el robot en un modelo visual. Esto permitiría a un usuario trabajar con el robot sin carga, eliminando, por ejemplo, la necesidad de utilizar el robot físico e interrumpir su funcionamiento. Además,

dado que no se utilizan las herramientas y objetos de trabajo reales, estos no representan una amenaza para los trabajadores ni son susceptibles a sufrir daños. El uso de este nuevo enfoque le permitiría al trabajador realizar una puesta a punto con mayor seguridad que en la programación en línea y manteniendo la precisión de la que carece la programación fuera de línea y también ayudará al crecimiento de una forma más avanzada en la industria, ya que haciendo uso de robots para automatizar el proceso de fabricación, las empresas pueden ofrecer alternativas eficientes y viables en lugar de cerrar las brechas de conocimiento y deslocalización en áreas donde es difícil contratar al personal adecuado.

1.3 Objetivos

El objetivo principal de este proyecto es lograr la conexión del robot Husky y, seguidamente, recolectar datos de este para implementar una sombra digital capaz de obtener información de sus componentes y poder representarlos junto a modelo en 3D del propio robot.

Como primer paso, será necesario crear un clúster local de Minikube [7] para poder desplegar las tecnologías de Eclipse Ditto [8] y Eclipse Hono [9], tecnologías que serán necesarias para el desarrollo de la sombra digital, ya que en Ditto se crearán los componentes que tendrá la sombra, así como la estructura de los datos de cada componente y Hono se encargará de recibir la información recibida por parte del robot, así como enrutar la información recibida al componente correspondiente creado en Ditto.

Se optó por el uso de la tecnología de Minikube para el clúster local debido a que permite crear un fantástico entorno de pruebas en la fase inicial de un proyecto. Esto es debido a la facilidad de despliegue local que proporciona al usuario, y no solo eso, sino que también es capaz de lograr que los servicios desplegados sean

capaces de estar comunicados entre sí, gracias a los enrutamientos internos creados por Minikube. Otras de las ventajas que tiene, es que, si por cualquier motivo un servicio falla, este es capaz de levantarse e incluso replicarse en caso de eliminación, permitiendo de esta manera una interfaz estable para el usuario, sin preocupación en caso de fallo.

Por otro lado, se utilizará el sistema ROS interno del robot en combinación del lenguaje *Python* [10] para obtener los datos de cada componente y sensor del mismo y así poder enviarlos a través del protocolo MQTT [11] al clúster local, ya que ROS es el sistema que trae por defecto y el lenguaje de Python es una de las opciones disponibles para el manejo de scripts internos de este sistema y que a su vez, es compatible con el protocolo MQTT. Para mostrar la información de forma visual se hará a través de Grafana [12] ya que esta plataforma nos permite darle representación a los datos, donde se cargará un modelo desarrollado del robot 3D en la plataforma de Unity [13] con la ayuda de *Blender* [14], dando como resultado final una plataforma donde aparecerá el modelo 3D junto a los datos del robot en tiempo real. Todo esto se explicará con mayor detenimiento en el capítulo 4 de esta memoria que trata sobre el desarrollo del proyecto.

Además, el hecho de realizar este Trabajo de Fin de Grado ha implicado otros objetivos que no solo incluyen el producto final, ya que se ha tenido que realizar un aprendizaje y estudio profundo de herramientas y conceptos nunca antes utilizados, como bien puede ser el uso de Kubernetes [15], el sistema ROS, modelado en Unity y el completo funcionamiento de los dispositivos IoT [16].

1.4 Metodología

Durante el desarrollo del proyecto se ha utilizado la metodología ágil. Este concepto surge de la necesidad de acelerar los métodos tradicionales del desarrollo

de trabajos dado que las metodologías anteriores eran muy estáticas. Esta nueva metodología pretende acabar con esto. Es una fórmula que destaca por su rapidez y flexibilidad, adaptándose siempre al máximo a las necesidades de sus clientes.

En concreto, se ha utilizado la técnica Scrum, dado que es imposible definir el proyecto al completo desde un principio, buscando darle más importancia a la velocidad y a la capacidad de adaptación de los cambios que surjan a medida del desarrollo del mismo sin importar el tamaño. Dicho por los propios creadores, "*Scrum es una forma de creación de conocimiento organizada, especialmente buena para promover innovación continua, de forma incremental*"[17].

Los proyectos que utilizan esta metodología generalmente evolucionan de acuerdo con un circuito de retroalimentación en lugar de planificarse o desarrollarse por adelantado. Durante estos periodos se diseñan y desarrollan tareas específicas, y al final de estos periodos se proporciona el avance al cliente y se reanuda el proceso. De esta forma, los clientes pueden recibir de forma paulatina las últimas novedades sobre sus productos, realizar cambios y priorizarlos.

1.5 Estructura de la memoria

Esta memoria consta de cinco capítulos, entre los que se encuentra el capítulo Introducción que acaba de ser presentado. A continuación, se mostrará el listado de capítulos junto al contenido de cada uno de ellos:

- *Capítulo 2. Descripción del sistema*

  En este capítulo, se analizará el proyecto terminado y las lecciones que se han tenido que tomar, así como también se observará el progreso a medida que se ha ido logrando.

- *Capítulo 3. Tecnologías utilizadas*

En este capítulo se hará una descripción de las tecnologías que han sido utilizadas en este proyecto para lograr una mayor comprensión por parte del lector.

- *Capítulo 4. Desarrollo del proyecto*

  En este capítulo se hará una descripción fase por fase de cómo fue realizándose el Trabajo de Fin de Grado, así como la solución a los problemas encontrados en cada una de las mismas.

- *Capítulo 5. Conclusiones y líneas futuras*

  En este capítulo se describirá la conclusión final de haber realizado este Trabajo de Fin de Grado, así como ideas que han ido surgiendo para líneas futuras.

2

Descripción del sistema

En este capítulo se hará una descripción del sistema y su organización de forma superficial, pues en el capítulo 4 se mostrará en detalle cómo se fue realizando paso a paso la configuración del proyecto.

Este proyecto fue posible gracias al grupo de investigación ERTIS de la UMA, ya que tenían en su departamento el robot Husky de la marca Clearpath, ofreciendo infinidad de posibilidades para crear un Trabajo de Fin de Grado con él, pues esta marca ofrece robots con muchas de aplicaciones y herramientas por defecto. Es entonces cuando surgió la oportunidad de adentrarse en el mundo de la robótica y de crear una sombra digital como primer paso hacia la creación de un gemelo digital y así poder explotar su funcionamiento.

Antes de comenzar se tuvo que analizar el alcance del trabajo a desarrollar y si en realidad era posible lograrlo. Como primer paso, se obtuvieron conocimientos

de las distintas herramientas disponibles para crear una sombra digital, así como fue necesario analizar su parte tecnológica, para finalmente, conocer cuáles serían los requerimientos mínimos (hardware) necesarios del ordenador donde se desplegaría para que el sistema funcionase de forma correcta.

2.1 Robot Husky

Husky es uno de los primeros robot de exterior que ya poseen ROS instalado de fábrica, lo que facilitará su manejo. Para saber cómo crear la sombra digital, primero habrá que analizar cuál es su equipamiento interno y configuración, para así averiguar cómo se puede conseguir enviar la información de los componentes que tiene.

Figura 2. Robot Husky

Como se puede observar en la figura 2, el robot Husky tiene los siguientes elementos incorporados:

- **Láser LMS111:** Proporciona lecturas sólidas de rango interior/exterior en un campo de visión 2D.
- **GPS:** Obtiene la localización exacta del robot

- **Router:** Crea una red local donde conectarse para acceder al robot, ya con una dirección IP asignada para poder realizar *SSH* y conectarse al servidor.
- **Antena wifi:** Permite la conexión a internet
- **Gamepad inalámbrico F710:** Ya configurado y conectado, permite manejar al robot
- **Ordenador:** Actúa como servidor del robot, tiene instalado *Ubuntu* [18] 18.04 y el sistema ROS con la versión *Melodic* [19]. En la imagen no se aprecia, pero se encuentra debajo de la placa metálica para que esté resguardado.

Una vez observados todos los componentes de forma superficial, se procede a examinar su configuración interna.

En primer lugar, se observó que posee un router configurado para crear una red local, pero el robot en sí no tiene configurada la antena wifi para poder acceder a internet. Esto era necesario, así que se tuvo que investigar para lograr que se conectase a un punto de acceso, siendo un teléfono personal el que actuó de router durante todo el proyecto, debido a que por motivos de seguridad es muy difícil configurar su acceso a la red de la UMA. Este paso es de los más importantes, dado que esto permitirá instalar complementos necesarios a través de internet en el servidor del robot y poder acceder a direcciones IP que se encuentren fuera de su red local.

Aclarado el área de redes, ahora toca observar la parte de sistemas. Este robot posee un ordenador con Ubuntu como servidor y el sistema ROS ya preinstalado. Los compañeros del departamento del grupo ERTIS facilitaron una máquina virtual con Ubuntu 18.04 y ROS ya configurado. Con esta máquina

virtual en un ordenador personal es como se iba a realizar toda la comunicación con el robot.

2.1.1 Conociendo el sistema ROS

Este robot, como se ha comentado con anterioridad, posee un ordenador con Ubuntu como servidor y el sistema ROS Melodic preinstalado, por lo que no tendremos que preocuparnos de cómo se monta el sistema, sino, únicamente, de su funcionamiento.

La primera y más importante pregunta fue ¿Cómo es posible obtener la información de los datos de cada sensor/componente que posee el robot? Es aquí, donde primero se tiene que conocer en cómo se estructura el sistema ROS, donde las características que más interesan son las siguientes [20] :

- **Paquetes**

  Es un elemento clave de su arquitectura. El paquete puede contener procesos de tiempo de ejecución de ROS (nodos), conjuntos de datos, archivos de configuración o lo que necesite. De hecho, todo el software ROS se encuentra empaquetado. El propósito de esto es permitir que cada paquete proporcione una funcionalidad específica para que el software se pueda crear y reutilizar fácilmente. Lógicamente, esta herramienta permite la creación de nuevos paquetes con el fin de desarrollar funcionalidades propias. Además, los paquetes pueden tener dependencias entre sí.

- **Nodos**

  El nodo es en realidad solo un archivo ejecutable en el paquete ROS. Los nodos ROS utilizan bibliotecas de clientes para comunicarse

con otros nodos. Los nodos pueden publicar o suscribirse a temas (tópicos). Además, pueden usar o proporcionar ciertos servicios.

- **Tópicos**

Los mensajes se enrutan a través del sistema de transporte utilizando la semántica de publicación/suscripción. Los nodos envían mensajes publicándolos sobre temas específicos. El tópico es el nombre utilizado para identificar el contenido del mensaje. Los nodos que están interesados en un tipo particular de datos se suscriben al tema correspondiente. Puede haber múltiples publicadores y suscriptores simultáneos para un solo tema, y un solo nodo puede publicar y/o suscribirse a múltiples temas. Los editores y los suscriptores generalmente desconocen la existencia del otro. La idea es separar la generación de información (publicación) de su consumo (suscriptor).

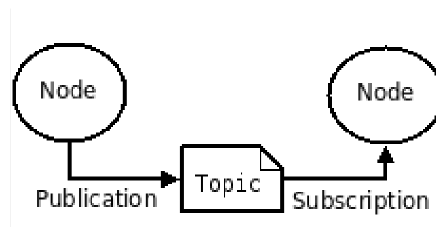


Figura 3. Comunicación de nodos en ROS

- **Mensajes**

Los nodos se comunican entre sí enviando mensajes. El mensaje es solo la estructura de datos que contiene el dato. Se admiten diferentes tipos estándares ya conocidos en otros ámbitos de la programación, como enteros, flotantes, booleanos e, incluso, matrices.

- **Maestro**

El maestro ROS es el nodo más importante de todos, pues establece la comunicación entre los nodos. Este es solo un nodo maestro que brinda servicios de nombres y registro a los nodos en el sistema ROS y, además, realiza un seguimiento de los editores de temas y los suscriptores. El papel del maestro es permitir que los nodos ROS individuales se encuentren entre sí. El protocolo más común utilizado para la conectividad es el Protocolo de control de transmisión/Protocolo de Internet (TCP/IP) o Protocolo de Internet, que se denomina TCP ROS en ROS. Si estos nodos pueden encontrarse, pueden comunicarse entre sí como pares.

Una de las funciones del nodo maestro es realizar un seguimiento del nodo a medida que se inicia en el sistema. De esta forma, el asistente proporciona asignaciones de conexión dinámicas. Sin embargo, los nodos no pueden comunicarse hasta que el maestro notifique la existencia a los demás.

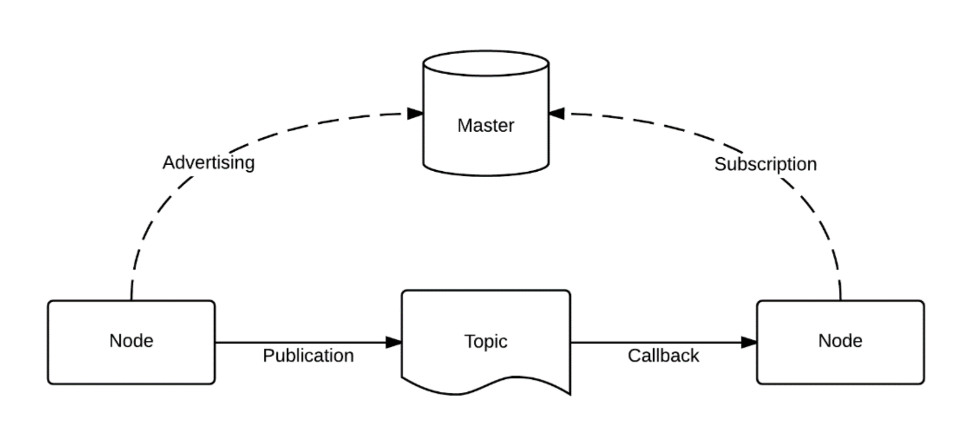


Figura 4. Funcionamiento del nodo maestro

2.1.2 Conclusiones de ROS

Una vez analizado el sistema ROS, se puede concretar que el propósito es crear un nuevo paquete y dentro de él un script en Python que sea capaz de crear un nuevo nodo suscriptor que se suscriba a un tópico de un nodo publicador de los que ya posee el robot, en donde cada tópico será la información que está transmitiendo cada componente del robot. Una vez obtenida la información, esta se transmitirá a donde se quiera a través del protocolo MQTT.

2.2 Creación de la sombra digital

Es aquí donde empieza la cuestión del asunto y la parte clave de este Trabajo de Fin de Grado, pues serán necesarias tecnologías capaces de recibir y estructurar los datos que van a llegar de parte del robot Husky.

Estas plataformas encargadas de realizar la sombra digital son Eclipse Ditto y Eclipse Hono, parte del proyecto Eclipse. Para lograr el funcionamiento, se hará uso de una arquitectura *open-source*, la cual se muestra en la Figura 5, comentándose posteriormente sus componentes y características.

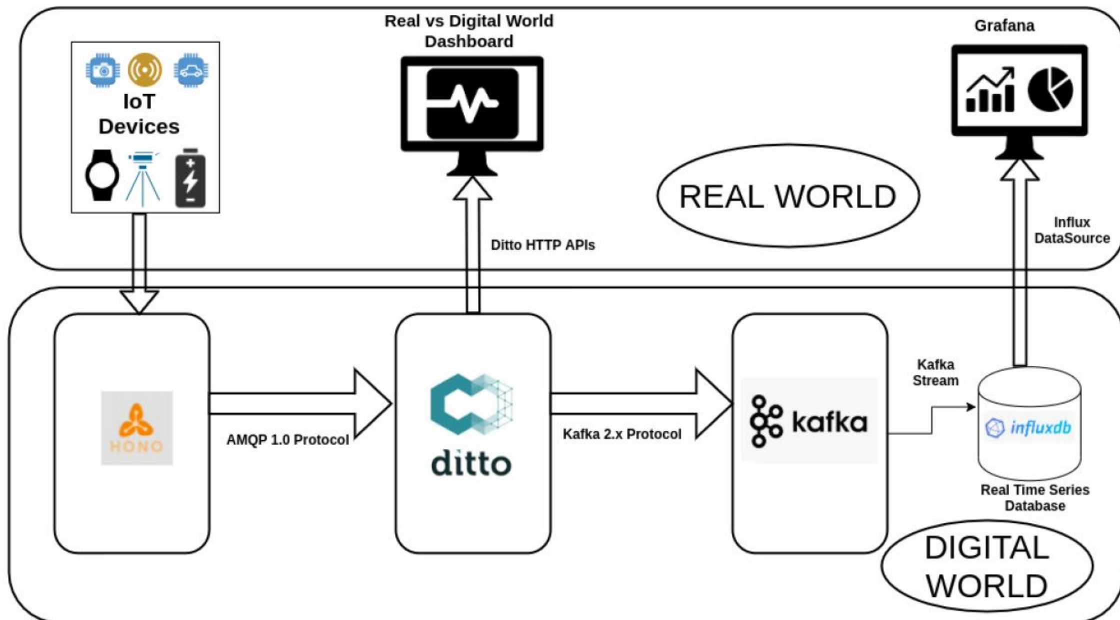


Figura 5. Arquitectura inicial open-source [21]

2.2.1 Instalación de forma local

Para conseguir la arquitectura mostrada con anterioridad y un funcionamiento consistente, era necesario el despliegue de una serie de servicios y componentes en un clúster de Kubernetes, pues esta era la plataforma que más se ajustaba a las necesidades de este proyecto, pero al no poseer un clúster de Kubernetes, este tenía que pasar a ser desplegado de manera local. Esto se llevaría a cabo a través de Minikube, permitiendo crear un clúster local de Kubernetes y un buen entorno de pruebas. Para logralo, primero había que hacer uso de Docker [22], ya que el despliegue de Minikube no es más que el despliegue de un contenedor con Kubernetes. Una vez instalado, ya se pudo proceder al despliegue de Eclipse Ditto y Eclipse Hono.

2.2.2 Despliegue Eclipse Ditto y Eclipse Hono

Una vez creado el clúster de Minikube junto a ciertos requisitos, ya se podía proceder al despliegue de Eclipse Ditto y Eclipse Hono dentro del mismo

clúster a través de repositorios. Estos repositorios son fáciles de instalar de forma local gracias a *Helm* [23], siendo este un administrador de aplicaciones de Kubernetes. En concreto, se instalará el paquete de *cloud2edge* [24], que contiene a Eclipse Ditto y Eclipse Hono y que permitirá conectar y administrar dispositivos de tipo sensor a la vez que se procesan sus datos en una plataforma de gemelos digitales. Para una mejor comprensión por parte del lector, se muestra la Figura 6 y una breve descripción del paquete:

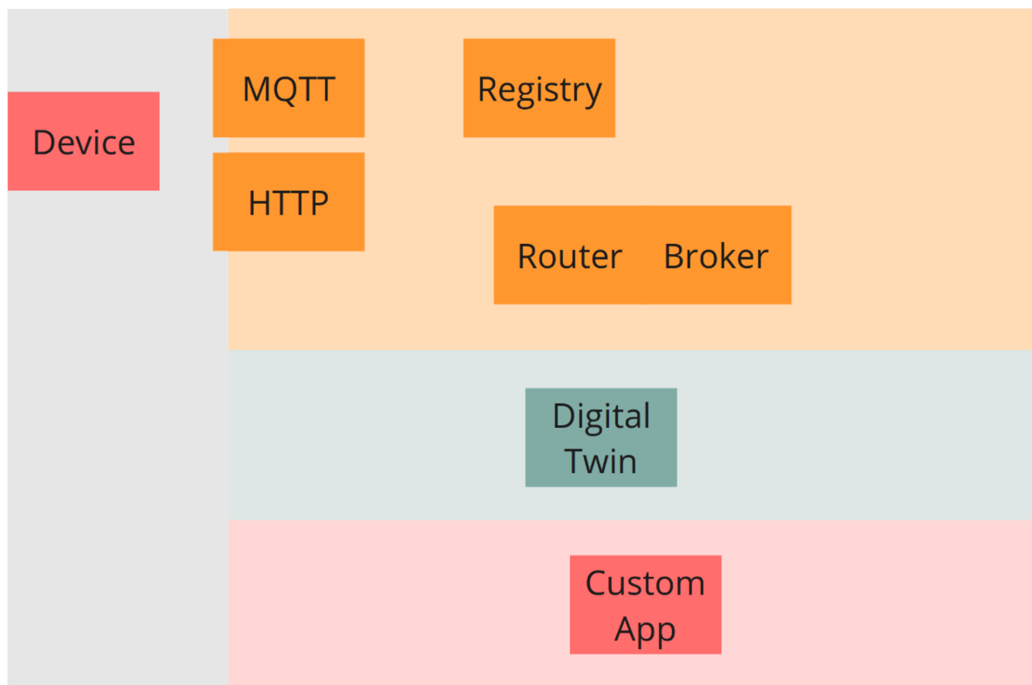


Figura 6. Descripción general del paquete cloud2edge

- **Eclipse Hono:** Proporciona la capa de mensajería de IoT. Permite que los dispositivos se conecten al clúster mediante varios protocolos específicos de IoT (en este caso, MQTT). Las conexiones se autenticarán en función de los datos del *device registry.* Los mensajes se pasarán a través del *router* , para los mensajes volátiles, y a través del *broker* para los mensajes persistentes.

- **Eclipse Ditto:** Aprovecha los datos de flujo de mensajes provenientes los dispositivos conectados a Eclipse Hono. Estos son procesados para que las aplicaciones puedan trabajar con datos más estructurados.

Este paquete permite obtener una aplicación personalizada que se comunicará de forma óptima con los dispositivos, utilizando un modelo de datos estructurados y una capa de mensajería escalable. Además, también proporciona una función para planificar, orquestar y monitorear la implementación de actualizaciones de software en los dispositivos conectados.

Al desplegar las plataformas en Minikube, se crean servicios de los mismos en el que cada servicio tiene una función diferente que nos servirá para la creación de la sombra digital. Estos servicios se encuentran comunicados unos con otros dentro de Minikube, con lo que cada uno puede acceder a la información del otro a través de la IP local.

2.2.3 Transmisión de datos de los componentes de Husky

En esta parte, un servicio del proyecto cloud2edge será el encargado de recibir los datos del robot Husky mediante un adaptador MQTT, siendo este el protocolo estándar para los dispositivos IoT y el que se va a utilizar en este proyecto.

El propósito es crear un script en Python dentro del sistema ROS de Husky para que envíe información constante del componente que deseemos. Esto no fue nada sencillo, pues hubo que profundizar en el funcionamiento del sistema ROS hasta encontrar una solución óptima, concluyendo en que la mejor opción sería crear con el script de Python un nodo suscriptor a un nodo publicador interno con el que poder acceder a los datos (tópicos) que muestra de forma constante el nodo publicador.

Hubo complicaciones a la hora de crear el script, pues no fue nada sencillo poder traer los datos y desglosarlos para enviarlo a la sombra digital creada en Ditto a través de Hono de la forma que se quisiera. Para esto también se investigó en el lenguaje Python usado dentro del sistema ROS para así conocer que paquetes del sistema ROS (*rospy, msgs*) había que importar para que el script funcionase y que más adelante se comprenderá el por qué la importancia de conocer dichos paquetes, que serán los que permitirán manejar los datos de cada tópico dentro del script.

Lo que más tardó en averiguarse fue la manera en exportar en internet o en LAN los servicios de cloud2edge, pues había errores en la comunicación entre ambos (robot y clúster). Esto se solucionó y se logró enviar los datos de manera satisfactoria mediante el script que se mostrará en el capítulo 4.

2.2.4 Visualización de datos

Estas plataformas comentadas son muy interesantes para recibir datos de componentes y crear gemelos digitales, pero tanto Eclipse Ditto como Eclipse Hono necesitan de otras plataformas para poder crear una buena solución y poder visualizar los datos de una forma simple y eficaz.

El objetivo principal era crear una solución final donde el usuario que quisiera interactuar con la sombra digital lo hiciese a través de la plataforma de Grafana, donde nos aparecerá un panel con el modelo en 3D del robot Husky desarrollado en Unity junto a los datos que recibe de Eclipse Ditto. Para ello, era necesario el uso de herramientas intermedias, tales como *Apache Kafka* [25], *Telegraf* [26] e *InfluxDB* [27].

Estas herramientas servirán para que los datos le lleguen a Grafana. Para que esto fuese así, Kafka creará una comunicación con Eclipse Ditto y recibirá los

datos de los componentes de una sombra digital creada en uno de sus tópicos. Telegraf por su parte actuará como consumidor de dicho tópico y los cargará en InfluxDB, que guardará los datos en forma de series temporales.

Por último, en la plataforma de Grafana, se podrá configurar para que le lleguen los datos que hay actualmente en InfluxDB, permitiendo así poder visualizar los datos de los componentes del robot de una forma más representativa.

Una vez realizado el proyecto, dará lugar a la siguiente arquitectura, que será la que se muestre en la Figura 7.

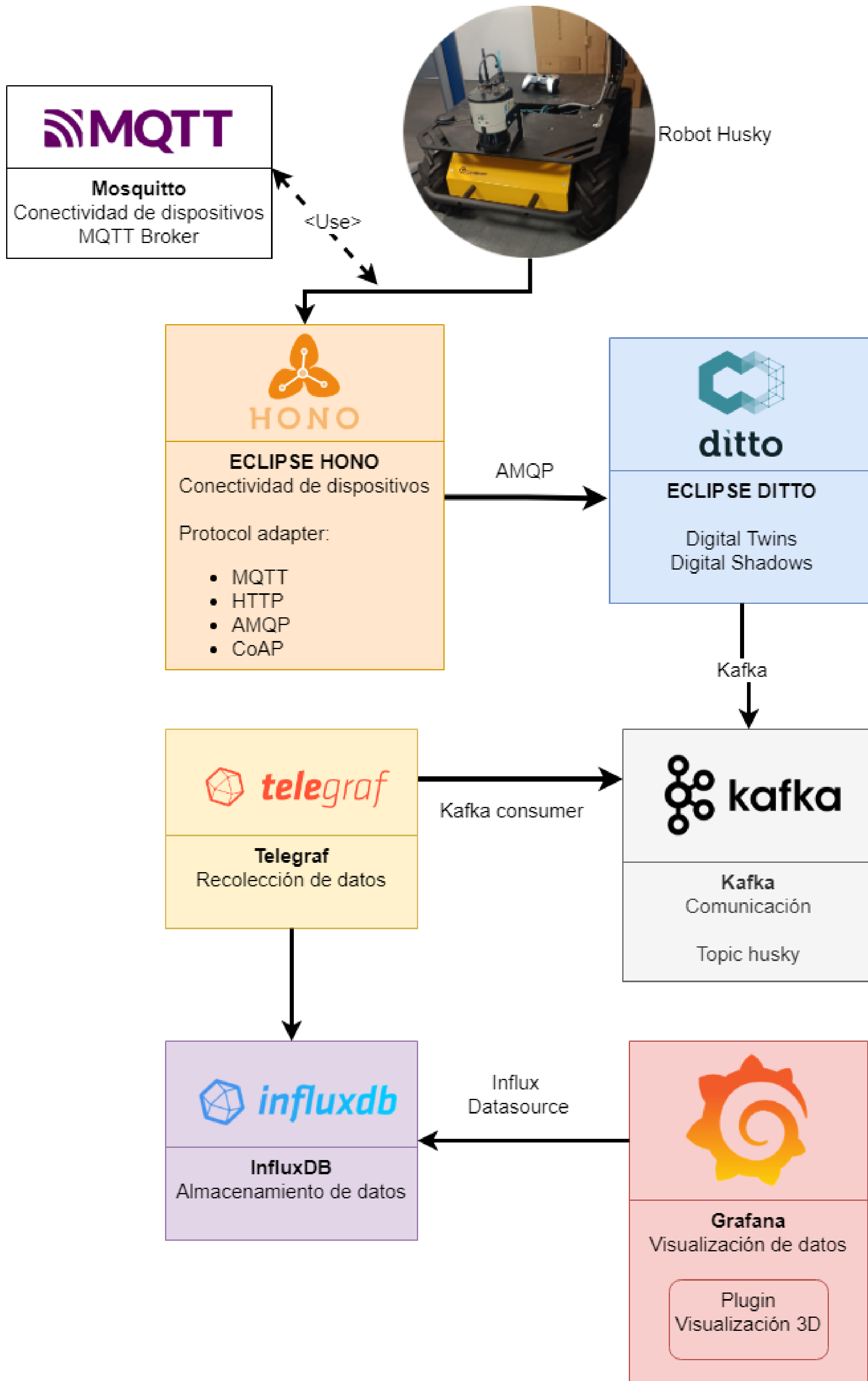


Figura 7. Arquitectura del proyecto

2.2.5 Problemas al exportar los servicios a internet del clúster

Uno de los mayores problemas que hubo durante la realización de este proyecto, era el poder acceder a los servicios de cloud2edge de forma externa. Estos poseen una IP para acceder a ellos, pero estas IP son locales del clúster de Minikube y no se puede acceder a ellos desde internet ni en local, solo desde dentro del propio clúster. Esto dificultaba enormemente la posibilidad de enviar los datos de los componentes del robot.

En los servicios desplegados, hay distintos *ServiceType* [28] dentro de Minikube, que no son más que la forma de poder exportar estos servicios a internet, existiendo las siguientes aparte de las que podemos ver arriba:

- **ClusterIp:** Proporciona al servicio una dirección IP interna del clúster. Si selecciona este valor, el servicio solo estará disponible desde el clúster. Este es el tipo de servicio predeterminado.
- **NodePort:** El servicio lo proporciona la dirección IP de cada nodo en el puerto estático (NodePort). El servicio ClusterIP se crea automáticamente y se enruta el NodePort del servicio. Pudiendo acceder al servicio NodePort desde fuera del clúster.
- **LoadBalancer:** Nos permite exponer el servicio de forma externa usando un balanceador de carga conectado al proveedor de la nube. Esto hará que los servicios creados automáticamente con NodePort y ClusterIP apuntaran al balanceador de carga.

Visto de esta manera, parece sencillo acceder al NodePort, pero esta opción no funciona tal cual en Minikube, pues no se puede acceder fuera del clúster o al menos, no con los conocimientos que teníamos en ese momento sobre Kubernetes y Minikube.

En vista del problema de exportación de servicios del clúster local, fue necesario encontrar una manera de poder acceder a dicho servicio interno. Una de ellas fue la creación de un *port-forward* [29], conocido como redirección de puertos o túnel de red, para seguidamente, con el puerto asignado al realizar el túnel de forma local, poder exportar este puerto (y el servicio adjunto a dicho puerto) a internet a través de una aplicación llamada *localtunnel* [30] para así poder acceder desde el robot. Esta aplicación permite obtener una página web con protocolo *HTTPS* de dicho puerto local, pero surgieron problemas al momento de enviar datos desde el robot, debido a que el protocolo MQTT para el envío de datos no es compatible con una dirección web con protocolo HTTPS.

Otra solución posible era exportar el servicio al menos en la red LAN. Debido a que el robot ya tenía acceso a internet, en concreto, usando un móvil como punto de acceso, se podía conectar también un portátil al mismo punto de acceso, permitiendo de esta manera que cualquier puerto que se expusiera en la dirección IP del portátil tuviese acceso desde la misma red LAN.

Esta última solución no fue nada simple de implementar, pues cuando se realizaba un túnel, el puerto del servicio se aplicaba a nuestro *localhost* (127.0.0.1) pero no a la dirección IP del portátil. Esto se solucionó tras una larga investigación y logrando exportar los puertos de los servicios de Minikube a la red local.

Los problemas surgidos y su solución serán detallados más a fondo en el capítulo 4, tal y como se comentó al principio de este.

2.3 Creación del modelo 3D

Desde el primer momento del proyecto, se tuvo claro que crear una representación 3D del robot Husky en la plataforma de Unity era parte fundamental para poder crear la sombra digital, pues el visionado de los datos

junto a un modelo 3D forman el núcleo de este Trabajo de Fin de Grado para obtener una representación visual y concisa.

Para la obtención del modelo, se puso en contacto con la empresa Clearpath. En su contestación, enviaron un diseño base pero en formato *IGS*, que es poco usual.

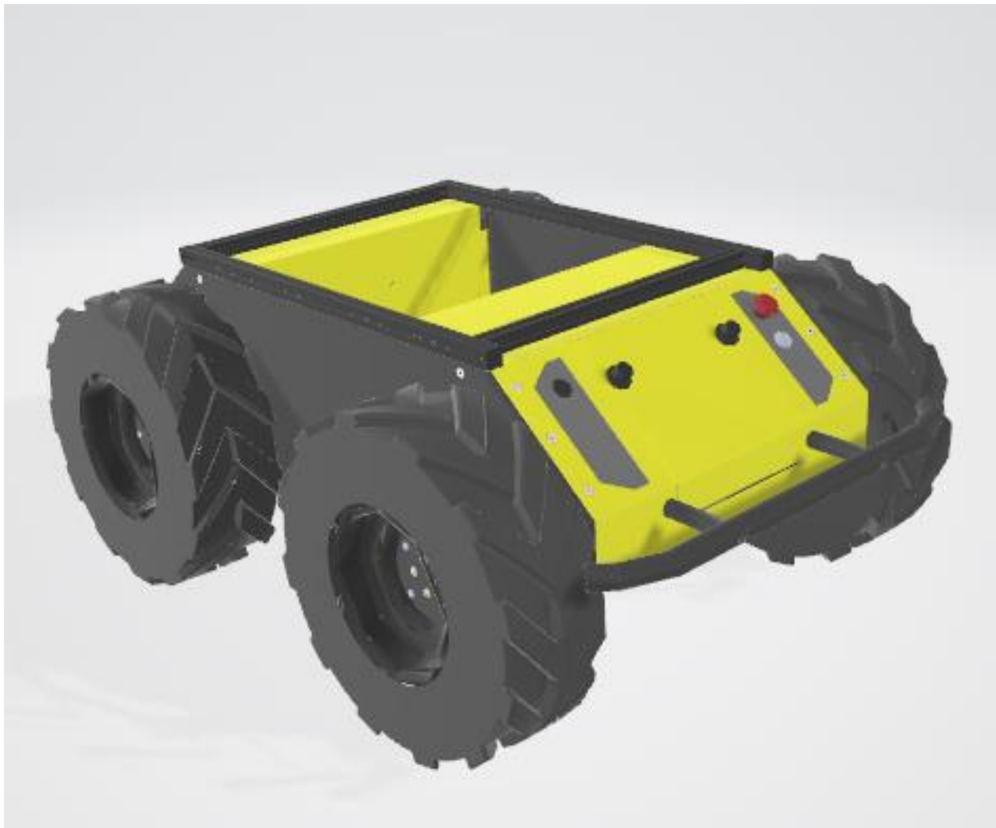


Figura 8. Modelo inicial del robot Husky en 3D

Este tipo de formato no era compatible con Unity, así que se tuvo que investigar para poder convertirlo en uno que sí lo fuera, como por ejemplo el formato *FBX*. La conversión de IGS a FBX se realizó a través de un programa llamado *SketchUp* [31], un software de diseño 3D y en el que una vez abierto el modelo en 3D del robot, pude exportarlo al formato que se deseaba.

Una vez obtenido el formato, se cargó en Unity, pero algunas texturas (colores) no eran parecidas al robot Husky del grupo ERTIS y tampoco traía los

mismos complementos, ya que el robot 3D implementado en Unity era el básico sin complementos.

2.3.1 Creación de objetos en Blender

Debido a que el robot Husky 3D desplegado en Unity era el básico, se tuvo que utilizar herramientas de modelado compatibles con Unity, es por ello que se utilizó Blender, un programa donde se pudo crear algunos objetos para que el modelo 3D fuese lo más realista posible. En este programa se crearon los siguientes objetos:

- Escáner laser

Figura 9. Escáner láser LMS111 en 3D

- Tapa chasis superior

Figura 10. Tapa chasis superior en 3D

- Router Wifi

Figura 11. Router Wifi en 3D

2.3.2 Modelo final del robot Husky en 3D

Ya creados los objetos que faltaban para que fuese lo más realista posible, se agregó al modelo de Unity obteniendo el siguiente resultado final del robot Husky en 3D:

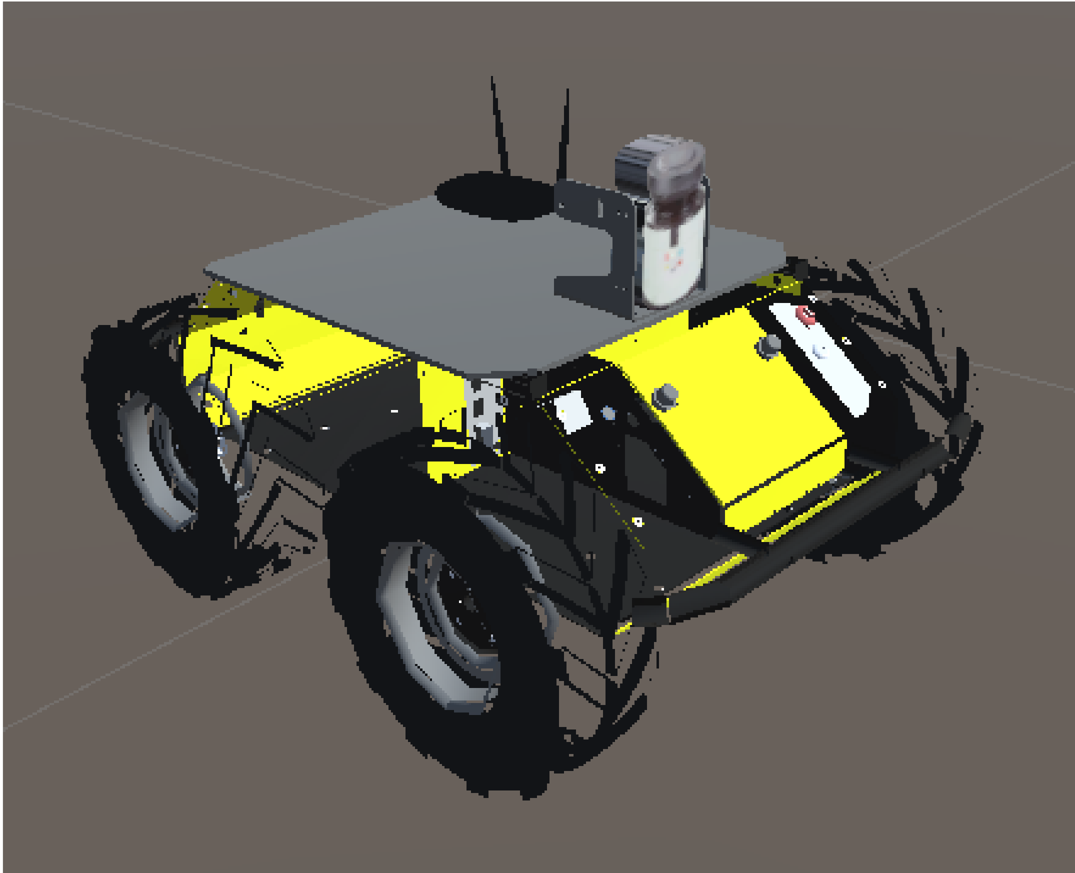


Figura 12. Modelo final del robot Husky en 3D

Siendo la Figura 12, el resultado final en la plataforma en Unity. Las texturas se muestran diferente al formato de la Figura 8 porque en esa imagen está abierto en otro programa (visor 3D de Windows). En esta última imagen lo vemos así debido a que es un problema ocasional de Unity, pues hace que por ejemplo las ruedas se vean como si fuese transparentes en algunos ángulos para poder ver lo que se encuentra detrás de ellas.

2.4 Uniendo las partes

Por último, desplegados los servicios necesarios en Minikube y conseguido que los scripts de Python creados dentro de ROS envíen los datos de cada componente en tiempo real, se logró así la creación de la sombra digital. Ya solo

quedaba mostrar el resultado final en Grafana, para así poder interactuar con la sombra digital creada.

Para ello, en la plataforma de Grafana, gracias a un plugin que fue desarrollado por Julia Robles Medina, se pudo cargar el modelo en 3D del robot Husky. Además, se le dio representación a los datos recibidos por los componentes, obteniendo de esta manera un panel que podrá visualizar el usuario.

2.5 Requisitos hardware

Una vez descrito todo lo que se ha realizado en este proyecto, se ha podido observar por parte del lector el múltiple uso de tecnologías actuales.

Para el desarrollo de este proyecto, se iba a utilizar un ordenador personal y por tanto, se tuvo que realizar la compra de uno nuevo, pues el que se tenía no iba a ser capaz de ejecutar todas estas tecnologías de forma eficiente.

El ordenador portátil adquirido y que cumplió las expectativas sin problemas tiene las siguientes características, que se situarán como requerimientos necesarios para toda persona que quiera realizar un proyecto similar:

- **Procesador:** Intel i5 de 8ª generación
- **Memoria RAM:** 16 Gb
- **Memoria interna:** 512 Gb (recomendable tecnología NVME)
- **Gráficos:** Integrados en procesador

Estas características son las que tienen el portátil personal adquirido para llevar a cabo este proyecto, no quiere decir que con menos no funcionaría, pero sí probable que se viese afectado el rendimiento e incluso el buen funcionamiento del producto final.

3

Tecnologías utilizadas

3.1 Docker

Es una plataforma de código abierto lanzada en 2013 que permite crear, probar e implementar aplicaciones de forma sencilla. Este permite empaquetar software de forma estándar en forma de "contenedores" que contienen todas las herramientas necesarias para ejecutar un proyecto. Todo esto proporciona una capa adicional de abstracción y automatización de virtualización de aplicaciones independientemente del sistema operativo. [32]

Estos contenedores han sido una solución a problemas habituales, sobre todo en el ámbito de DevOps, ya que permite moverse entre entornos de desarrollo como puede ser en local o en un entorno real de producción. Permite testear de forma segura un proyecto, ya que nuestro código no se comportará de una manera

distinta, ya que todo lo que necesitas se encuentra dentro del contenedor y no será fluctuante en los diferentes entornos.

Docker es una tecnología que emerge de la virtualización ligera debido a que es posible crear múltiples sistemas totalmente aislados entre si sobre el mismo hardware o sistema. Además, en comparación con las típicas máquinas virtuales, el rendimiento del CPU, memoria, disco duro, etc. es superior a la hora de utilizar estos contenedores, lo que nos permite concluir que las máquinas virtuales son menos eficientes. Es por eso por lo que Docker es cada día más utilizado en el mundo de la informática, además de por rendimiento, porque ofrece una mayor seguridad, una característica muy codiciada y que se encuentra a la orden del día. [33]

Figura 13. Logotipo de Docker

3.2 Kubernetes

Desarrollado por Google en 2014, es una de las plataformas de código abierto más populares que existen hoy en día, que nos permite orquestar las aplicaciones que se encuentran en los contenedores mencionados con anterioridad en la tecnología de Docker. Implementa sistemas escalables y confiables en los contenedores a través de API orientadas a estas aplicaciones.

Kubernetes se ha convertido en la API estándar y es una infraestructura creada para sistemas distribuidos (equipos independientes que actúan como uno solo) que facilita todos los elementos necesarios a la hora de crear aplicaciones en

la nube a cualquier escala. Esta herramienta es muy potente y se utiliza en casi todos los ámbitos que puedas imaginar. Permite la ejecución en varias máquinas a la vez que se conectan a través de la red y permite coordinar sus acciones a través de ella.

¿Por qué se denomina como confiable y escalable? Esto se debe a las siguientes características que posee:

- **Alta disponibilidad:** Si falla algún servicio, es capaz de reiniciarse o crear otro en su lugar. Esto es muy importante, ya que la mayoría de los servicios que se utilizan no se pueden permitir estar fuera de servicio durante demasiado tiempo.
- **Balanceo de carga automático:** Debido a que todos los contenedores se encuentran orquestados por Kubernetes, este mismo se encargará de balancear la carga sobre ellos cuando sea necesario, sin necesidad de otros mecanismos.
- **Escalamiento:** A medida que un servicio crece, permite escalar tanto los equipos como el software que los desarrollan, logrando la escalabilidad y favoreciendo las arquitecturas desacopladas.
- **Abstracción de su infraestructura:** Permite que los desarrolladores la puedan usar fácilmente debido a su infraestructura de autoservicio.

En vista a todo lo anterior, el único inconveniente que se le puede encontrar a esta potente herramienta es su complejidad para montar toda la configuración por primera vez. [34]

Figura 14. Logotipo de Kubernetes

3.3 Minikube

Este es un clúster de Kubernetes de un solo nodo recomendado para un solo entorno de aprendizaje alojado. Minikube facilita la prueba de su instalación de Kubernetes en su máquina local. Se necesitan más recursos para probar algo o implementar una nueva aplicación en una máquina local, pero en un Minikube, tanto el maestro como el trabajador se ejecutan en el mismo nodo. Viene con el *runtime* del contenedor Docker preinstalado. Desde Minikube, puede ejecutar contenedores y pods en su máquina local, ya sea a través de una caja virtual o usando algún tipo de hipervisor.

Minikube puede crear y ejecutar clústeres de Kubernetes en su máquina local. Sin embargo, necesita una forma de interactuar con el clúster o un servidor API que actúe como puerta de enlace para el clúster. Para hacer esto, necesita instalar kubectl. Kubectl puede interactuar con los clústeres de Kubernetes creados por Minikube. Esta es una herramienta de línea de comandos que le permite interactuar con la configuración mediante comandos.[35]

Es por ello que se ha optado por usar Minikube para este trabajo, ya que contiene todo lo necesario para crear un clúster de forma fácil y eficaz.

Figura 15. Logotipo de Minikube

3.4 Eclipse Ditto

Es el software de código abierto de la Fundación Eclipse. Se puede descargar e instalar mediante contenedores. En este caso se despliega en un clúster de Minikube a través de un repositorio Helm y la comunicación se realiza a través de WebSockets a través de *API REST*, Kafka, *AMQP*, MQTT o *HTTP*, por lo que puede conectarse a prácticamente cualquier dispositivo.

Este marco considera que un gemelo digital es una abstracción de un objeto real, pero tiene todas sus características. Los gemelos digitales pueden duplicar objetos reales, brindar servicios a su alrededor, mantenerlos sincronizados con los objetos y usarse tanto a nivel industrial como de consumo. Según Eclipse Ditto, la plataforma de gemelos digitales debe poder interactuar con el gemelo, permitir que solo los usuarios autorizados accedan a este, interactuar con el mismo e integrarse con la infraestructura del servidor. Eclipse Ditto crea un gemelo digital como archivo JSON con "atributos" para metadatos estáticos y "características" para datos de estado dinámico. El atributo puede contener tantas claves JSON como necesite y tantos valores como necesite. Con las características pasa lo mismo, pero cada una tiene un objeto JSON llamado "propiedad". Cada objeto se considera una "cosa". Cada "cosa" consta de un identificador, una lista de control de acceso, atributos y características. [36]

En el contexto de este trabajo, Eclipse Ditto es el encargado del registro de gemelos digitales, de almacenar su estado actual y de hacerlo disponible en

cualquier momento. Además, Ditto lo integramos con Eclipse Hono, lo que lo convierte en un buen paquete para servir más protocolos diferentes para las aplicaciones y dispositivos.

Figura 16. Logotipo Eclipse Ditto

3.5 Eclipse Hono

Es un centro IoT multiprotocolo que conecta un gran número de dispositivos con una aplicación basada en la nube (en nuestro caso Ditto desplegado en un clúster local). Una de las principales ventajas de Hono es su soporte con diversos protocolos como HTTP, MQTT, *CoAP* y AMQP 1.0 para interactuar con los dispositivos a través de la llamada API.

Para cada uno de estos protocolos, Hono dispone de un adaptador de protocolo que transporta los datos a su red de mensajería. A continuación, permite que Ditto recupere esos datos de la red utilizando un punto final AMQP 1.0. También permite soportar protocolos adicionales mediante la implementación del adaptador de protocolo correspondiente.

Eclipse Hono soporta tres tipos de intercambio de datos. En primer lugar, permite la transmisión de datos de telemetría a nuestra aplicación principal, como, por ejemplo, la información sobre la temperatura o la humedad. En segundo lugar es posible transmitir eventos a la aplicación para indicar situaciones como la finalización de un paso del proceso. La principal diferencia entre la telemetría y los datos de eventos es la semántica de entrega. En el caso de los eventos, la

entrega ocurre "al menos una vez". En el caso de los mensajes de telemetría el modo de entrega es "al menos una vez" o "como máximo una vez". En tercer lugar, permite que las aplicaciones puedan llevar a cabo los patrones de interacción de comando y control para desencadenar acciones en un dispositivo y enviar datos al mismo, aunque esta última característica no la usaremos en este trabajo.

Eclipse Hono cubre un aspecto importante de toda infraestructura de IoT, como es la vinculación de los dispositivos con las aplicaciones finales. Especialmente, los dispositivos que no exponen APIs RESTful a través de HTTP y pueden depender de una infraestructura como en los protocolos de publicación-suscripción (por ejemplo, MQTT) se benefician de un *hub* de dispositivos centralizado, la cual será el fundamento de nuestro trabajo para el envío de datos. [37]

Como conclusión tendremos que Eclipse Hono es la opción de conexión más común cuando se pretende crear sombras o gemelos digitales con Eclipse Ditto. Por lo tanto, estas dos tecnologías formaran el núcleo principal del trabajo a desarrollar.

Figura 17. Logotipo de Eclipse Hono

3.6 Eclipse Mosquitto y Eclipse Paho

Eclipse Mosquitto [38] para el broker y Eclipse Paho [39] para el cliente son las implementaciones necesarias para utilizar el protocolo MQTT, formando también parte de la fundación Eclipse.

Eclipse Mosquitto es un intermediario de mensajes de código abierto que implementa varias versiones, aunque nosotros utilizaremos la 3.1 del protocolo MQTT. Mosquitto es liviano y adecuado para usar en todos los dispositivos, desde computadoras de placa única de bajo consumo hasta servidores completos. El protocolo MQTT proporciona una forma simplificada de enviar mensajes utilizando el modelo de publicación/suscripción. Esto lo hace adecuado para la mensajería IoT que se utiliza, por ejemplo, en sensores de baja potencia, dispositivos móviles como teléfonos, computadoras integradas, microcontroladores y, en nuestro caso, robots. El proyecto Mosquitto también proporciona un cliente MQTT y una biblioteca C para la implementación de clientes MQTT de línea de comandos: mosquitto_pub (publicador) y mosquitto_sub (suscriptor).

En cambio, Eclipse Paho, en concreto la versión para Python ya que es la compatible junto a nuestro sistema de ROS, implementa también la versión 3.1 del protocolo MQTT para ser compatible con el bróker. Este proporciona una clase de cliente que permite que las aplicaciones se conecten a un intermediario MQTT para publicar mensajes, suscribirse a temas y recibir mensajes publicados. También proporciona algunas funciones de ayuda para que la publicación de mensajes únicos en un servidor MQTT sea muy sencilla.

Para aclarar al lector, el protocolo MQTT es un protocolo de conectividad de máquina a máquina/IoT. Diseñado como un transporte de mensajería de publicación/suscripción extremadamente liviano, es útil para conexiones con ubicaciones remotas donde se requiere una huella de código pequeña y/o el ancho de banda de la red es pequeño.

Figura 18. Logotipo del protocolo MQTT que engloba ambas tecnologías

3.7 Apache Kafka

Los clústeres de Kafka se dedican a almacenar flujos de datos. Un flujo de datos es una serie de mensajes/eventos que una aplicación genera de manera continua y otras aplicaciones consumen de forma secuencial y gradual. La API Connect se usa para llevar datos a Kafka y exportar el flujo de datos a sistemas externos, como bases de datos y sistemas de archivos distribuidos. Para el procesamiento de flujos de datos, la API Streams permite a los desarrolladores especificar una canalización de procesamiento de flujos compleja que lee el flujo de entrada del clúster de Kafka y vuelve a escribir los resultados en Kafka. [40]

Es por ello que se ha implementado Kafka en el clúster de Minikube, ya que será el mediador encargado de conectarse con Eclipse Ditto y recibir los datos del gemelo digital, los cuales recolectará Telegraf para almacenarlos en InfluxDB.

Figura 19. Logotipo de Apache Kafka

3.8 InfluxDB

Es un sistema de gestión de base de datos que almacena datos de series temporales, ideal para datos de componentes o protocolos con marcas temporales, además, permite crear software de IoT, análisis y monitoreo. Utiliza un lenguaje propio de consultas llamado Flux, que es muy parecido a JavaScript. [41]

Su forma de funcionar está basado en *push*, es decir, requiere una aplicación que esté insertando datos en la base de datos de forma constante. En este proyecto lo que se ha hecho es utilizar un complemento que forma parte de InfluxDB llamado Telegraf y que hará de mediador conectándose con Apache Kafka para recibir los datos.

Telegraf no es más que un agente basado en servidor que se dedica a recoger y almacenar todo tipo de datos para luego cargarlos principalmente en InfluxDB, aunque también puede integrarse con otras tecnologías.

Figura 20. Logotipo de InfluxDB

3.9 Grafana

Es una solución de análisis y monitoreo que nos proporciona un modelo de fuente de datos adicional y es compatible con muchas de las bases de datos de series temporales más populares, como pueden ser: Graphite, Prometheus, Elastic search e InfluxDB. Además, nos permite a los usuarios consultar, ver y alertar métricas y registros desde múltiples ubicaciones de almacenamiento. [42]

Esta herramienta nos será útil para culminar el trabajo. Con ella podremos visualizar los datos del robot junto a su diseño en 3D, ya que nos facilita la opción de crear nuevos paneles a modo de complemento, pues por defecto solo trae paneles simples.

Figura 21. Logotipo de Grafana

3.10 ROS (Robotic Operating System)

ROS (Robot Operating System) es una plataforma abierta para el desarrollo de sistemas robóticos. Proporciona una variedad de servicios y bibliotecas que facilitan con creces la creación de aplicaciones robóticas complejas. Su mayor ventaja reside en que es un solo sistema que combina lo mejor de cada uno de los sistemas de robots ya existentes para interactuar con los robots modernos.

Desde sus inicios, ROS ha sido diseñado para facilitar el intercambio de software entre entusiastas de los robots y profesionales de todo el mundo a través de un enfoque educativo y abierto. Además, se está creando una gran comunidad de colaboradores en todo el mundo.

Durante el desarrollo, ROS puede usar una variedad de lenguajes de programación. De forma oficial, es compatible con Python, C ++, Lisp y muchos más. ROS puede ejecutarse en máquinas similares a Unix, principalmente Ubuntu y Mac OS X, pero se puede encontrar soporte comunitario en otras plataformas como Fedora y Gentoo. En nuestro caso, su ejecución se basó en Ubuntu 18.04.

ROS tiene una gran comunidad de desarrolladores: Investigadores, entusiastas, fabricantes de hardware, empresas de soporte como Willow Garage y Google. Además de todo esto, numerosos ejemplos, bibliotecas listas para usar (algoritmos de navegación, visión artificial, etc.) y la creciente comunidad ROS

son buenos puntos de partida para embarcarse en un viaje alrededor del mundo de la robótica.

Figura 22. Logotipo del sistema ROS

3.11 Visual Studio Code

Visual Studio Code [43] es un editor de código fuente ligero y potente que se ejecuta en escritorio y funciona en Windows, macOS y Linux. Contiene compatibilidad integrada con JavaScript, TypeScript, Node.js y un rico ecosistema de extensiones para otros lenguajes (C ++, Java, Python, PHP, etc.) y tiempos de ejecución (Unity, etc.).

Figura 23. Logotipo de Visual Studio Code

3.12 Ubuntu Linux

Ubuntu es una distribución de Linux basada en Debian GNU/Linux y contiene principalmente software gratuito de código abierto. Se puede utilizar en ordenadores y servidores. Está dirigido al usuario promedio con un enfoque en la facilidad de uso y la mejora de la experiencia del usuario. [44]

En esta caso, Ubuntu está instalado en la máquina del robot como soporte para ROS y en el ordenador personal a través de una máquina virtual para poder conectarse al servidor de Husky.

Figura 24. Logotipo de Ubuntu Linux

3.13 Python

Es un lenguaje de programación de alto nivel utilizado para desarrollar todo tipo de aplicaciones. A diferencia de otros lenguajes como Java y .NET, este es un lenguaje interpretado. Esto significa que no tiene que compilar para ejecutar una aplicación escrita en Python, se ejecuta directamente desde su computadora usando un programa llamado intérprete, dado que no es necesario "traducir" en lenguaje máquina. [45]

Existen varias versiones de Python, pero en este caso se usará la 2.7.17, ya que es la que tiene instalada Husky en su servidor por defecto y la que permitirá realizar los scripts capaces de enviar la información del robot al clúster local.

Figura 25. Logotipo de Python

3.14 Unity

Es un potente motor 3D multiplataforma con un entorno de desarrollo fácil de usar que lo hace sencillo para el principiante y a su vez potente para el experto. Además, este es interesante para cualquiera que desee crear con total facilidad

juegos y aplicaciones en 3D para dispositivos móviles, de escritorio, web y consolas. Siendo estos últimos casos los más interesantes para este proyecto.

Figura 26. Logotipo de Unity

3.15 Blender y 3D Builder

Blender es un paquete gratuito de creación 3D de código abierto. Es compatible con toda la canalización 3D, incluido el modelado, la edición, la animación, la simulación, el renderizado, la composición, el seguimiento de movimiento e incluso la edición de video y la creación de juegos. Blender es ideal para individuos y pequeños estudios que se benefician de una canalización unificada y un proceso de desarrollo ágil.

En cambio, 3D Builder [46] es un programa gratuito que se encuentra en la store de Microsoft, que nos permitirá visualizar, crear y personalizar objetos en 3D.

El uso de estas dos tecnologías será útil para el desarrollo de objetos 3D que serán exportados a Unity.

Figura 27. Logotipo de Blender

4

Desarrollo del proyecto

En los capítulos anteriores se ha realizado una descripción general del proyecto y las tecnologías a utilizar para que el lector pueda entrar en contexto a la hora de visualizar los siguientes pasos. En este capítulo, se contará en detalle cómo se ha ido desarrollando por fases la creación del producto final, haciendo hincapié en los problemas y en cómo se solucionaron los mismos.

4.1 Requisitos previos para realizar los pasos

Antes de comenzar, si se realiza desde Windows como en este caso, es necesario una herramienta para el uso de *BASH*. Para cumplir dicho requisito se ha utilizado el entorno de *Git Bash* [26]. Además, es necesario la previa instalación de *curl* y Helm para el envío de peticiones y el manejo de repositorios de Kubernetes, respectivamente. También comentar que se ha utilizado *Terminal Windows* [47] para facilitar el uso con líneas de comando, una aplicación gratuita

que se puede obtener en la tienda de Microsoft y que nos permite abrir todas las terminales que se quiera. Además, se da por supuesto tener todas las tecnologías comentadas en el capítulo anterior.

4.2 FASE 1: Instalación de Minikube

Para poder desplegar el proyecto cloud2edge de forma local, una de las opciones viables era a través de un clúster local, para lo que había que inicializar un clúster de Kubernetes local, en concreto Minikube. Todas las líneas que aparecerán a continuación se ejecutaran en *Powershell*, a no ser que se especifique lo contrario.

4.2.1 Instalación del Backend

Antes de desplegar el clúster, hay que descargarse mediante Helm el repositorio de cloud2edge que contiene todo lo necesario para crear la sombra digital. Esto se realiza con los siguientes comandos:

```
helm repo add eclipse-iot https://eclipse.org/packages/charts

helm repo update
```

Luego, se procede a inicializar el clúster con la siguiente línea de comando:

```
Minikube start --cpus 4 --disk-size 40gb --memory 8192 --kubernetes-version v1.17.0
```

Surgieron varios problemas al añadir la cantidad de memoria RAM mostrada, pero se pudo solucionar modificando el archivo *.wslconfig* para añadir toda la memoria RAM que nos permitiese el sistema. En realidad no es necesario darle tanta RAM al clúster, solo que más adelante nos dará más fluidez cuando se hayan desplegado todos los servicios. También en el comando se obliga a que la versión utilizada sea la v1.17.0 a la hora de crear el clúster debido a que es la

necesaria para este proyecto. Una vez creado el clúster, para que la versión local se descargue y se sincronice con dicha versión, ya que eso es importante para que no haya conflicto de versiones antes instaladas, se ejecuta la siguiente línea:

```
Minikube kubectl –– get pods -A
```

Este comando descarga el kubectl necesario para tanto el cliente como el servidor tenga la misma versión de kubectl, ya que estos deben de tener la misma versión, o como mucho, variar en una (1.18-1.16). Esto en realidad es imprescindible, pues el proyecto no funcionó hasta que se realizó este paso que solucionaba el conflicto de versiones. Lo comprobaremos con el siguiente comando.

```
Minikube kubectl -- version
```

Si todo se ha instalado de forma correcta, el resultado debe ser similar a este.

```
Client Version: version.Info{Major:"1", Minor:"17", GitVersion:"v1.17.0", GitCommit:"70132b0f130acc0bed193d9ba59dd186f0e6
34cf", GitTreeState:"clean", BuildDate:"2019-12-07T21:20:10Z", GoVersion:"go1.13.4", Compiler:"gc", Platform:"windows/amd
64"}
Server Version: version.Info{Major:"1", Minor:"17", GitVersion:"v1.17.0", GitCommit:"70132b0f130acc0bed193d9ba59dd186f0e6
34cf", GitTreeState:"clean", BuildDate:"2019-12-07T21:12:17Z", GoVersion:"go1.13.4", Compiler:"gc", Platform:"linux/amd64
"}
```

Figura 28. Versiones de kubectl similares

Después de haber cargado el repositorio, se debe crear un *Persistent Volumen* con extensión *.yaml*, que no es más que un archivo que contiene los datos necesarios para crear el almacenamiento para el clúster, para que la instalación funcione bien y los datos persistan aunque el contenedor sea reiniciado. Para ello se debe crear este archivo en una carpeta donde tengamos acceso desde la terminal:

```
apiVersion: v1

kind: PersistentVolume

metadata:
```

```
 name: pv-device-registry
spec:
 accessModes:
- ReadWriteOnce
 capacity:
  storage: 1Mi
 hostPath:
  path: /mnt/
  type: Directory
```

Ya creado el archivo, en este caso llamado *persistente.yaml*, se inserta lo siguiente para que se despliegue en el clúster.

```
kubectl create -f persistente.yaml
```

El paso anterior es importante, ya que sin ello no se podrá realizar el siguiente paso, tal y como pasó en los inicios de este proyecto.

Después de esto, ya se puede proceder a la instalación del paquete cloud2edge, que se guardará en un *namespace* llamado cloud2edge para así tenerlo organizado. Esto se realizará en una terminal con Git Bash:

```
NS=cloud2edge
kubectl create namespace $NS
RELEASE=c2e
helm install -n $NS $RELEASE eclipse-iot/cloud2edge --set hono.prometheus.createInstance=false --set hono.grafana.enabled=false --dependency-update --debug
```

El comando *debug* nos servirá para ir viendo cómo se va instalando el paquete y saber si todo ha ido como debe.

```
NOTES:
Thank you for installing the Cloud2Edge Eclipse IoT Package.

Your release is named 'c2e'.

It might take some time for all of the package's services to start up.
You can check the current status by issuing the following command:

  $ kubectl get -n cloud2edge pods

Once all services are up and running, the output should look similar to this:

NAME                                        READY   STATUS    RESTARTS   AGE
c2e-adapter-amqp-vertx-65cfb4d675-g5wn4     1/1     Running   0          3m15s
c2e-adapter-http-vertx-66bd6bb89c-mng5t     1/1     Running   0          3m15s
c2e-adapter-mqtt-vertx-765fcd578b-5rl7n     1/1     Running   0          3m15s
c2e-artemis-f8f7dc7f4-864cj                 1/1     Running   0          3m15s
c2e-dispatch-router-6c77dc78bd-hjn4l        1/1     Running   0          3m15s
c2e-ditto-concierge-5949cb449d-46vp5        1/1     Running   0          3m15s
c2e-ditto-connectivity-779fc6f574-gd5gq     1/1     Running   0          3m15s
c2e-ditto-gateway-5cfdc9f9fc-kl6mh          1/1     Running   0          3m15s
c2e-ditto-nginx-864cffd948-mvcdg            1/1     Running   0          3m15s
c2e-ditto-policies-86748888d6-g2xs7         1/1     Running   0          3m15s
c2e-ditto-swaggerui-75677db684-h5ngx        1/1     Running   0          3m15s
c2e-ditto-things-67b86569fb-z4w7b           1/1     Running   0          3m15s
c2e-ditto-thingssearch-55fb58cd96-52zg8     1/1     Running   0          3m15s
c2e-service-auth-84d9695cfc-5wlfh           1/1     Running   0          3m15s
c2e-service-command-router-8d6bcc664-5xfz8  1/1     Running   0          3m15s
c2e-service-device-registry-975dfcdb7-d6d6h 1/1     Running   0          3m15s
ditto-mongodb-8ff6bb7cf-r7hh5               1/1     Running   0          3m15s

Once all pods have status 'Running', the Cloud2Edge package is ready to be used.
```

Figura 29. Resultado tras instalar el paquete cloud2edge

Con esto ya se tendría instalado el paquete cloud2edge de manera correcta en el ordenador. A partir de aquí, los siguientes pasos se harán sobre esta instalación.

4.2.2 Conociendo el backend

Una vez desplegado el backend del proyecto, hay que observar la IP de los servicios que se van a utilizar y que fueron explicados con anterioridad. Con el siguiente comando se puede visualizar en Git Bash:

```
kubectl get services --namespace $NS
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
c2e-adapter-amqp-vertx	NodePort	10.96.82.249	<none>	5672:32672/TCP,5671:32671/TCP	13d
c2e-adapter-http-vertx	NodePort	10.96.230.223	<none>	8080:30080/TCP,8443:30443/TCP	13d
c2e-adapter-mqtt-vertx	NodePort	10.96.161.26	<none>	1883:31883/TCP,8883:30883/TCP	13d
c2e-artemis	ClusterIP	10.96.144.203	<none>	5671/TCP	13d
c2e-dispatch-router	ClusterIP	10.96.204.131	<none>	5673/TCP	13d
c2e-dispatch-router-ext	NodePort	10.96.69.234	<none>	15671:30671/TCP,15672:30672/TCP	13d
c2e-ditto-gateway	ClusterIP	10.96.137.174	<none>	8080/TCP	13d
c2e-ditto-nginx	NodePort	10.96.163.134	<none>	8080:32698/TCP	13d
c2e-ditto-swaggerui	ClusterIP	10.96.200.189	<none>	8080/TCP	13d
c2e-service-auth	ClusterIP	10.96.214.170	<none>	5671/TCP	13d
c2e-service-command-router	ClusterIP	10.96.61.210	<none>	5671/TCP	13d
c2e-service-device-registry	ClusterIP	10.96.21.215	<none>	5671/TCP,8080/TCP,8443/TCP	13d
c2e-service-device-registry-ext	NodePort	10.96.92.128	<none>	28080:31080/TCP,28443:31443/TCP	13d
ditto-mongodb	ClusterIP	10.96.235.83	<none>	27017/TCP	13d

Figura 30. Servicios de cloud2edge desplegados en Minikube

En la figura se observan los servicios desplegados, pero en este caso solo se hará uso de tres de ellos de forma directa:

- **c2e-service-device-registry-ext:** Servicio para registrar los sensores en Eclipse Hono.
- **c2e-ditto-nginx:** Servicio para guardar el estado de los sensores en Eclipse Ditto. Información que recibirá a través de Eclipse Hono.
- **c2e-adapter-mqtt-vertx:** Servicio que sirve para recibir la información de los sensores a través del protocolo MQTT.

Como ya se comentó en el capítulo 2, la IP's que se observan en los servicios son las que se encuentran dentro del clúster de Minikube, pero en este caso lo que interesa es publicarlas en la red para poder acceder a ellas desde el robot Husky, en concreto el servicio de *c2e-adapter-mqtt-vertx*, pues será el encargado de recibir los datos de cada componente. Estas IP's ni siquiera se pueden acceder desde la terminal, pues el ordenador no las reconocerá como existentes localmente, por tanto si queremos realizar algún comando dentro del clúster, debemos de realizar un SSH dentro del Minikube.

```
Minikube ssh -n Minikube
```

La primera opción para exportar los servicios fue crear un port-forward (túnel) del servicio que quisiéramos con la siguiente línea en Git Bash, la cual solo es capaz de exportar el servicio de forma local.

```
kubectl port-forward service/<servicio> PuertoEx:PuertoIn -n namespace
```

- **Servicio:** El servicio del paquete que queremos exportar.
- **PuertoIn:** El puerto que tiene dicho servicio dentro del clúster.
- **PuertoEx:** El puerto que queremos que tenga dicho servicio en localhost (de forma local).

Una vez creado el túnel de forma local y con un puerto externo asignado para el servicio que quisiéramos, teniendo instalado localtunnel, exportamos el servicio a internet con la siguiente línea.

```
lt --port <puerto>
```

En donde puerto se refiere al que tiene el servicio en el localhost asignado con antelación. Esto devolverá una página web con un dominio aleatorio que se mantendrá abierta siempre y cuando no se cierre la terminal ni se deje de exportar el servicio mediante un túnel. En las siguientes fases del proyecto, observaremos que esto no funciona debido a problemas con el protocolo MQTT que se explicará más adelante.

4.3 FASE 2: Configurando el robot Husky

En esta fase, el objetivo era poder adentrarse dentro del robot Husky para poder enviar los datos hacía el clúster local desplegado y así poder crear la sombra digital. Todo lo que se realice referente al robot, será dentro de la máquina virtual de Ubuntu 18.04.

4.3.1 Comunicación con el robot Husky

Lo primero de todo fue conectarse al robot para observar sus nodos y tópicos y así conocer cuáles de ellos serían los más interesantes para la obtención de datos.

Este robot ya posee un router wifi para poder conectarse a él. Su conexión wifi se llama *mal02_radio* y, una vez conectado al robot, se puede observar desde la terminal diferentes datos.

Es importante recordar que, siempre que se abra una terminal cuando uno esté conectado al robot y quiera interactuar con él, se debe insertar la siguiente línea para definir el nodo master que posee el robot y que permitirá acceder a sus nodos y tópicos, ya que este nodo maestro es el encargado de comunicarse y organizar a los demás.

```
export ROS_MASTER_URI=http://192.168.131.1:11311
```

La dirección IP mostrada puede variar según como se realice la conexión con el robot, ya que no será la misma si la conexión se realiza a través de su router (IP estática) o si se realiza a través de un punto de acceso al que el robot se encuentre conectado vía wifi (IP dinámica). La IP utilizada en este ejemplo es la asignada dentro de su propio router. El puerto no variará, ya que pertenece al nodo maestro.

Una vez insertado ya se puede proceder a ejecutar comando propios de ROS, como algunos de los siguientes que son los primeros que utilizaremos:

- *Rostopic list*: Devuelve una lista con los tópicos activos.
- *Rosnode list*: Devuelve una lista con los nodos activos.

En este punto, lo que interesa es elegir el tópico que devuelva la información más interesante del robot, para ello, hay que ver cuales están disponibles en Husky.

```
rostopic list
```

```
administrator@cpr-a200-0664:~$ rostopic list
/cloud
/cmd_vel
/diagnostics
/diagnostics_agg
/diagnostics_toplevel_state
/e_stop
/encoder
/gps/fix
/gps/nmea_sentence
/gps/nmea_sentence_out
/gps/time_reference
/gps/vel
/husky_velocity_controller/cmd_vel
/husky_velocity_controller/odom
/husky_velocity_controller/parameter_descriptions
/husky_velocity_controller/parameter_updates
/imu
/imu/data
/imu/data_raw
/imu/mag
/imu_filter/parameter_descriptions
/imu_filter/parameter_updates
/imu_manager/bond
/imu_um7/data
/imu_um7/mag
/imu_um7/rpy
/imu_um7/temperature
/joint_states
/joy_teleop/cmd_vel
/joy_teleop/joy
/joy_teleop/joy/set_feedback
/odometry/filtered
/odometry/gps
/rosout
/rosout_agg
/scan
/set_pose
/sick_lms_1xx/parameter_descriptions
/sick_lms_1xx/parameter_updates
/status
/tf
/tf_static
/twist_marker_server/cmd_vel
/twist_marker_server/feedback
/twist_marker_server/update
/twist_marker_server/update_full
```

Figura 31. Lista de tópicos del robot Husky

Con la introducción del siguiente comando junto al tópico deseado se puede ver el contenido exacto de cada uno, así como la estructura que utilizan sus datos (*int*, *float*, etc).

```
rostopic info /<tópico>
```

De la lista de tópicos mostrados se hará uso de los siguientes debido a la información que poseen, ya que parecen una buena introducción de datos para crear la sombra digital.

- /status → Contiene información de la batería, temperatura de motores, etc.

```
ertis@ertis-virtual-machine:~/catkin_ws$ rostopic info /status
Type: husky_msgs/HuskyStatus

Publishers:
 * /husky_node (http://cpr-a200-0664:38453/)

Subscribers: None


ertis@ertis-virtual-machine:~/catkin_ws$ rosmsg show husky_msgs/HuskyStatus
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
uint32 uptime
float64 ros_control_loop_freq
float64 mcu_and_user_port_current
float64 left_driver_current
float64 right_driver_current
float64 battery_voltage
float64 left_driver_voltage
float64 right_driver_voltage
float64 left_driver_temp
float64 right_driver_temp
float64 left_motor_temp
float64 right_motor_temp
uint16 capacity_estimate
float64 charge_estimate
bool timeout
bool lockout
bool e_stop
bool ros_pause
bool no_battery
bool current_limit
```

Figura 32. Datos del tópico /status

- /scan → Contiene la información que recibe del láser incorporando, ángulo máximo, ángulo mínimo, etc.

```
ertis@ertis-virtual-machine:~/catkin_ws$ rostopic info /scan
Type: sensor_msgs/LaserScan

Publishers:
 * /sick_lms_1xx (http://cpr-a200-0664:46431/)

Subscribers: None


ertis@ertis-virtual-machine:~/catkin_ws$ rosmsg show sensor_msgs/LaserScan
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
float32 angle_min
float32 angle_max
float32 angle_increment
float32 time_increment
float32 scan_time
float32 range_min
float32 range_max
float32[] ranges
float32[] intensities
```

Figura 33. Datos del tópico /scan

En las siguientes figuras, se mostrarán los datos que están publicando los tópicos de forma constante para obtener otra visión del tópico.

- /joint_states → Contiene la información de las ruedas que tiene, la posición de cada una, etc.

```
administrator@cpr-a200-0664:~$ rostopic echo /joint_states
header:
  seq: 38286
  stamp:
    secs: 1652283867
    nsecs: 308953431
  frame_id: ''
name: [front_left_wheel, front_right_wheel, rear_left_wheel, rear_right_wheel]
position: [-0.6965475469412478, -0.7571168988491823, -0.6965475469412478, -0.7571168988491823]
velocity: [0.0, 0.0, 0.0, 0.0]
effort: [0.0, 0.0, 0.0, 0.0]
---
header:
  seq: 38287
  stamp:
    secs: 1652283867
    nsecs: 407852371
  frame_id: ''
name: [front_left_wheel, front_right_wheel, rear_left_wheel, rear_right_wheel]
position: [-0.6965475469412478, -0.7571168988491823, -0.6965475469412478, -0.7571168988491823]
velocity: [0.0, 0.0, 0.0, 0.0]
effort: [0.0, 0.0, 0.0, 0.0]
---
header:
```

Figura 34. Datos del tópico /join_states

- /gps/fix → Contiene la información de la latitud y longitud exacta de Husky

```
^Cadministrator@cpr-a200-0664:~$ rostopic echo /gps/fix
header:
  seq: 14805
  stamp:
    secs: 1652282998
    nsecs: 874406276
  frame_id: "navsat"
status:
  status: -1
  service: 1
latitude: 36.7165781667
longitude: -4.49984416667
altitude: nan
position_covariance: [nan, 0.0, 0.0, 0.0, nan, 0.0, 0.0, 0.0, nan]
position_covariance_type: 1
---
header:
  seq: 14806
  stamp:
    secs: 1652282999
    nsecs:  74272548
  frame_id: "navsat"
status:
  status: -1
  service: 1
latitude: 36.7165781667
longitude: -4.49984416667
altitude: nan
position_covariance: [nan, 0.0, 0.0, 0.0, nan, 0.0, 0.0, 0.0, nan]
position_covariance_type: 1
---
```

Figura 35. Datos del tópico /gps/fix

4.3.2 Conectando a internet al robot Husky

Para poder enviar la información de estos tópicos al clúster local era necesario que el robot estuviese en la misma red LAN que el ordenador portátil y, además, este no se podía conectar a la red wifi creada por el robot porque era necesario que tuviese acceso a internet.

En Husky, el wifi se configura con *netplan* [48], pero este servicio que ya traía instalado era erróneo (un calendario), así que para arreglarlo se tuvo que instalar a través de un pendrive el netplan correspondiente al configurador de red. Una vez instalado, se creó el archivo 60-wireless.yaml en /etc/netplan para configurar que el usb wifi del robot se conecte al punto de acceso preferido.

Para finalizar, se logró conectar a internet configurando como punto de acceso un móvil personal. Así que cada vez que se quiera cambiar el punto a donde se quiera que se conecte el robot Husky habrá que modificar el archivo mencionado.

4.3.3 Creando el script para enviar datos

Para este paso, hubo bastantes complicaciones tanto para crear el ejecutable dentro de ROS como a la hora de crear el script necesario.

Lo primero que había que realizar era escoger el tópico necesario y el nodo que estuviese publicando dicha información (paso 4.3.1). Una vez localizado estos datos, se procedía a crear el script correspondiente. Como ejemplo, se va a basar en el script para el tópico */status* durante los siguientes pasos, señalando las partes a tener en cuenta. También mencionar, que la versión de Python utilizada por parte del sistema ROS es la 2.7.17 y, por tanto, se tuvo que instalar el cliente *paho-mqtt* para la versión correspondiente antes de proceder.

```
#!/usr/bin/env python
import rospy
from husky_msgs.msg import HuskyStatus
from paho.mqtt import client as mqtt_client
import time
import random
import json
```

Figura 36. Cabecera script

Aquí se puede observar los paquetes importados, “rospy” es necesario para saber que procede del sistema ROS y “paho.mqtt” es parte del protocolo MQTT, sin embargo, lo más importante aparece en el paquete importado “from husky_msg.msg import HuskyStatus” pues sirve para extraer el tipo de mensaje que está enviándose en este tópico. Para conocer qué tipo de paquete hay que importar para traer los datos del tópico que se quiere, es necesario recordar el

paso 4.3.1, en el que cada vez que se crea un script hay que reconocer primero qué tipo de mensaje tiene cada tópico.

```
#topic status
#Constantes de conectividad con Eclipse Hono
broker = "XXX.XXX.X.X" #IP de eclipse Hono
port = XXXX #Puerto Eclipse Hono
topic = "telemetry" #siempre el mismo
rand = random.randint(0, 1000)
client_id = "python-mqtt-" + str(rand)
username = "statusid@husky" #id@tenant
password = "password"

#Constantes para crear el mensaje de Eclipse Ditto
DITTO_NAMESPACE = "sensor"
DITTO_THING_ID = "status"

client = None
```

Figura 37. Constantes de conectividad MQTT

Además, para configurar la conexión con el clúster, hay que definir los siguientes elementos. En donde *bróker* hay que poner la IP y puerto de Eclipse Hono (servicio MQTT) y la variable *topic* que siempre será la misma y que se definirá más tarde en Ditto junto al *username, password*, *DITTO_NAMESPACE* y *DITTO_THING_ID*. El cliente, por defecto, siempre será el mismo.

Además, será necesario establecer una función para crear la conexión a través del protocolo MQTT. La estructura siempre será la misma, lo que si cambiará son las constantes que quieras utilizar. [49]

```python
def connect_mqtt():
    def on_connect(client, userdata, flags, rc):
        if rc == 0:
            print("Connected to MQTT Broker!")
        else:
            print("Failed to connect, return code %d\n", rc)
    # Set Connecting Client ID
    client = mqtt_client.Client(client_id)
    client.username_pw_set(username, password)
    client.on_connect = on_connect
    client.connect(broker, port)
    return client
```

Figura 38. Función para crear la conexión MQTT.

Luego, para crear un nodo suscriptor a uno publicador existente, es necesario describir esta función en el paquete.

```python
def listener():
    global client
    client = connect_mqtt()
    client.loop_start()
    time.sleep(5)
    rospy.init_node('sendstatus',anonymous=True) #nombre del nodo que se ha creado para suscribirse al topico
    rospy.Subscriber('/status',HuskyStatus,getValues) #aqui el topico al que me suscribo
    rospy.spin()
```

Figura 39. Función que se suscribe a un nodo publicador

En "init_node" se sitúa el nodo suscriptor que se va a crear y en "suscriber" el tópico al que se va a suscribir junto a su tipo de mensaje publicado. Además, este llamará a la función donde se recogerán los datos que devuelve dicho tópico. [50]

Así mismo, para el envío de datos, hay que tener en cuenta que Eclipse Ditto funciona con una estructura de tipo JSON, por lo tanto, hay que enviar los datos con dicha estructura. Esto es fundamental, pues hay que respetarla, ya que luego en la creación de la sombra dentro de Ditto habrá que seguir el mismo esquema.

```python
def getValues(data):
    battery_v = data.battery_voltage #voltage bateria voltio
    capacity_battery = data.capacity_estimate #capacidad bateria estimada
    #temperatura motores
    l_motor = data.left_motor_temp
    r_motor = data.right_motor_temp
    if battery_v is not None and capacity_battery is not None and l_motor is not None and r_motor is not None:
        output = {
            "topic": DITTO_NAMESPACE + "/" + DITTO_THING_ID + "/things/twin/commands/modify",
            "headers": {
                "response-required": False,
                "content-type": "application/vnd.eclipse.ditto+json"
            },
            "path": "/features",
            "value": {
                "battery_voltage": {
                    "properties": {
                        "value": battery_v
                    }
                },
                "capacity_battery": {
                    "properties": {
                        "value": capacity_battery
                    }
                },
                "left_motor_temp": {
                    "properties": {
                        "value": l_motor
                    }
                },
                "right_motor_temp": {
                    "properties": {
                        "value": r_motor
                    }
                }
            }
        }
        publish(json.dumps(output))
    else:
        print("Falla en la lectura. Revisa el circuito")
        return None
```

Figura 40. Función que recoge los datos del nodo publicador y lo estructura para enviarlo como JSON

Como se puede observar, esta es la parte fundamental del script, pues guarda las variables que se recogen en "data" que son los valores que está recibiendo del tópico que publica el nodo interno y se hará uso de los valores que más interesen para situarlos en JSON. Esto mismo habrá que hacerlo de una manera similar para los demás scripts. Además, tanto en Python como en JSON no hay que declarar si es un tipo primitivo (int, double, etc), por lo que no es necesario saber qué tipo de dato se está trayendo para declararlo.

Para completar el script y conocer de manera eficaz si su funcionalidad es la adecuada, es posible realizar una función que por pantalla muestre en la terminal lo que está enviando a través del tópico

```python
def publish(msg):
    if msg is not None:
        result = client.publish(topic, msg)
        status = result[0]
        if status == 0:
            print("Send " + msg  + " to topic " + topic)
        else:
            print("Failed to send message to topic " + topic)
```

Figura 41. Función que muestra lo que está enviando el script

Por último, si se quiere saber en concreto que tipo de datos se trae cada tópico, solo se tendría que buscar en la wiki de ROS [51], que es la página que contiene toda la información del sistema ROS. Como ejemplo, se mostrará la información del tópico que se ha creado en este script.

husky_msgs/HuskyStatus Message

File: husky_msgs/HuskyStatus.msg

Raw Message Definition

```
Header header

# MCU Uptime, in ms
uint32 uptime

# ROS Control loop frequency (PC-side)
float64 ros_control_loop_freq

# Current draw of platform components, in amps
float64 mcu_and_user_port_current
float64 left_driver_current
float64 right_driver_current

# Voltage of platform components, in volts
float64 battery_voltage
float64 left_driver_voltage
float64 right_driver_voltage

# Component temperatures, in C
float64 left_driver_temp
float64 right_driver_temp
float64 left_motor_temp
float64 right_motor_temp

# Battery capacity (Wh) and charge (%) estimate
uint16 capacity_estimate
float64 charge_estimate

# Husky error/stop conditions
bool timeout
bool lockout
bool e_stop
bool ros_pause
bool no_battery
bool current_limit
```

Figura 42. Datos que contiene el tópico */status* [52].

De esta manera, se puede saber qué es lo que se está trayendo cada variable y cuál es su tipo primitivo.

4.3.4 Como ejecutar el script para enviar datos desde Husky

Para poder ejecutar el script, primero hay que realizar ciertos pasos dentro del robot Husky.

Una vez dentro del servidor del robot Husky a través de SSH, hay que situarse en la carpeta catkin_ws y ejecutar el siguiente comando.

```
roscreate-pkg <nombrepaquete> <variables>
```

Con este se creará un nuevo paquete dentro del robot donde se situaran los scripts recolectores de datos. En este caso, se ejecuta la siguiente instrucción.

```
roscreate-pkg tfg std_msgs rospy
```

Para que este paquete lo reconozca el sistema, habrá que dirigirse por comandos a la carpeta del paquete, en este caso llamada “tfg” y ejecutar lo siguiente.

```
rosmake
```

Esto lo que hará es compilar el paquete y hacerlo visible dentro del sistema. Ya terminado el comando, dirigirse a la carpeta tfg/src y allí alojar los scripts a ejecutar. No hay que olvidar darle permiso a estos scripts para que estos sean ejecutables.

```
chmod +x <nombrescript>
```

Realizado todos los pasos anteriores, ahora se puede escribir lo siguiente para que el script se ejecute y devuelva por pantalla un *log* con el mensaje del tópico enviado al clúster.

```
rosrun <nombrepaq> <Script>
```

4.4 FASE 3: Creación de la sombra en el clúster local

Antes de proceder con la ejecución del script para enviar los datos, se debe crear la sombra de Husky en el clúster con la estructura correspondiente para cada componente. [53]

4.4.1 Registrar el dispositivo

En este paso, el IP utilizado será el servicio de registro de dispositivos de Eclipse Hono. Lo primero es crear un *tenant*. Este representa una entidad a la que pertenecen los sensores. Por ejemplo, *husky* sería nuestro *tenant* y cada uno de sus sensores un *device*.

Este se crea con el siguiente comando, en el que husky es el nombre que queremos ponerle al *tenant*.

```
curl -I -X POST http://XXX.XXX.XX.XX:XXXXX/v1/tenants/husky
```

Ya creado el *tenant* se añaden los sensores que están conectados a Husky, al menos los que se tienen pensado recoger. En este caso solo se pondrá el ejemplo de uno de ellos, siendo este el sensor *status* (que representa al tópico).

```
curl -I -X
POSThttp://XXX.XXX.XX.XX:XXXXX/v1/devices/husky/sensor:status
```

En este momento, se ha creado la entidad en Eclipse Hono, pero no se ha asociado a ningún dispositivo fuera del clúster. El dispositivo como en sí no se asocia en ningún momento, sino que mediante unas credenciales envía la información pertinente a una API y, para asignar esas credenciales, hay que ejecutar el siguiente comando.

```
curl -i -X PUT -H "Content-Type: application/json" --data '[
```

```
{
 "type": "hashed-password",
 "auth-id": "statustid",
 "secrets": [{
  "pwd-plain": "password"
 }]
}]' http:// XXX.XXX.XX.XX:XXXXX /v1/credentials/husky/sensor:status
```

Con esto ya estaría preparada la conexión del dispositivo con Eclipse Hono. El dispositivo deberá utilizar estos datos para enviar su información a través del servicio MQTT que ofrece este mismo.

4.4.2 Crear la sombra digital de Husky

Si se quiere hacer una sombra digital con Eclipse Ditto, hay que establecer una conexión con cada tenant que se cree en Eclipse Hono. Para ello se ejecuta el siguiente comando, estableciendo previamente las variables necesarias.

La IP y el puerto de los siguientes comandos corresponden con el servicio de Ditto (ditto-nginx).

```
NS=cloud2edge
RELEASE=c2e
HONO_TENANT=raspberry
DITTO_DEVOPS_PWD=$(kubectl --namespace ${NS} get secret ${RELEASE}-ditto-gateway-secret -o jsonpath="{.data.devops-password}" | base64 --decode)
```

Luego, hay que crear una política que irá asociada a cada sensor que se cree dentro de Ditto.

```
curl -i -X PUT -u ditto:ditto -H 'Content-Type: application/json' --data '{
"entries": { "DEFAULT": { "subjects": { "{{ request:subjectId }}": { "type":
```

```
"Ditto user authenticated via nginx" } }, "resources": { "thing:/": { "grant":
["READ", "WRITE"], "revoke": [] }, "policy:/": { "grant": ["READ", "WRITE"],
"revoke": [] }, "message:/": { "grant": ["READ", "WRITE"], "revoke": [] } } },
"HONO": { "subjects": { "pre-authenticated:hono-connection": { "type":
"Connection to Eclipse Hono" } }, "resources": { "thing:/": { "grant": ["READ",
"WRITE"], "revoke": [] }, "message:/": { "grant": ["READ", "WRITE"],
"revoke": [] } } } } }'
http://XXX.XXX.XX.XX:XXXXX/api/2/policies/sensor:policy
```

Para finalizar, hay que crear la sombra digital como tal dentro de Eclipse Ditto, este trabaja con el formato JSON y hay que tenerlo en cuenta a la hora de realizar el script de Python. Con este comando, se creará dentro de Ditto el esquema que contendrá los datos que envía cada sensor en el script creado y, además, también se asignará la política creada.

```
curl -X PUT -u ditto:ditto -H 'Content-Type: application/json' --data '{
"policyId": "sensor:policy",
  "features": {
   "battery_voltage": {
          "properties": {
            "value": null
          }
        },
        "capacity_battery": {
          "properties": {
            "value": null
          }
        },
        "left_motor_temp": {
          "properties": {
```

```
            "value": null
          }
        },
        "right_motor_temp": {
          "properties": {
            "value": null
          }
        }
} }' http://XXX.XXX.XX.XX:XXXXX/api/2/things/sensor:status
```

Con todo lo anterior realizado, ya estaría creada la sombra digital dentro de Eclipse Ditto. Luego, si se quiere añadir más sensores, solo habría que registrarlo en Eclipse Hono como se ha hecho en el paso 4.4.1 y luego crear su esquema JSON dentro de Eclipse Ditto tal y como acaba de observarse en el último comando.

4.5 FASE 4: Consulta de la sombra digital

Antes de proceder con el almacenamiento de los datos que enviará el robot Husky, se debe probar que la sombra digital ha sido creada de forma satisfactoria.

Para ello, bastaría con la siguiente instrucción, en el que se debe ver la sombra creada junto a los sensores, además de los campos de datos de cada uno [54]. Deberían de aparecer con los atributos en “null” si no ha recibido nada, y, una vez que haya recibido datos, aparecerá el último que le ha llegado.

```
curl -X GET "http://XXX.XXX.XX.XX:XXXXX/api/2/search/things" -u ditto:ditto -H "accept: application/json"
```

En este caso, mostrará la Figura 43, en donde aparecerán los sensores creados (scan, status, joint, gps) junto a sus atributos. En el momento de la toma de esta captura, ya hubo envío de datos.

```
hingsr@minikube:~$ curl -u ditto:ditto -w '\n' http://10.96.163.134:8080/api/2/th
```


```
[{"thingId":"org.eclipse.packages.c2e:demo-device","policyId":"org.eclipse.packages.c2e:demo-device","attributes":{"loca
tion":"Germany"},"features":{"temperature":{"properties":{"value":null}},"humidity":{"properties":{"value":null}}}},{"th
ingId":"sensor:gps","policyId":"sensor:policy","features":{"latitude":{"properties":{"value":36.716486833333335}},"longi
tude":{"properties":{"value":-4.4997375}}}},{"thingId":"sensor:joint","policyId":"sensor:policy","features":{"position":
{"properties":{"value":[0,0,0,0]}},"effort":{"properties":{"value":[0,0,0,0]}},"name":{"properties":{"value":["front_lef
t_wheel","front_right_wheel","rear_left_wheel","rear_right_wheel"]}},"velocity":{"properties":{"value":[0,0,0,0]}}}},{"t
hingId":"sensor:scan","policyId":"sensor:policy","features":{"scan_time":{"properties":{"value":0.019999999552965164}},"
angle_min":{"properties":{"value":-2.356194496154785}},"range_max":{"properties":{"value":25}},"range_min":{"properties"
:{"value":0.05000000074505806}},"angle_max":{"properties":{"value":2.356194496154785}}}},{"thingId":"sensor:status","pol
icyId":"sensor:policy","features":{"left_motor_temp":{"properties":{"value":20.55}},"battery_voltage":{"properties":{"va
lue":24.13}},"capacity_battery":{"properties":{"value":480}},"right_motor_temp":{"properties":{"value":20.06}}}}]
```


Figura 43. Sensores de la sombra junto a los datos recibidos

4.6 FASE 5: Enviar datos al clúster

Ya creada la sombra, se procede a enviar los datos desde dentro del sistema ROS. Es aquí cuando surgieron problemas, pues resultaba que el túnel usado para exportar los servicios a internet no era compatibles con el protocolo MQTT. Esto se debe a que, con el programa localtunnel, el servicio se devolvía en forma de página web con protocolo HTTPS, que no es compatible con conexiones MQTT.

Por tanto, se investigó la forma de realizar un túnel de modo que ese puerto pueda ser expuesto en la red local y que toda máquina que esté conectada al mismo punto de acceso pueda acceder a él.

Se intentó con el siguiente comando:

```
kubectl port-forward service/c2e-adapter-mqtt-vertx 86:1883 -n cloud2edge
```

Para exponer ese puerto de forma local, además, hubo que añadir la excepción de ese puerto en el *firewall* de Windows para que aceptara las conexiones entrantes, aunque tampoco surgió efecto.

Buscando más a fondo en la web, se consiguió comprender el problema. Esto no llegaba a funcionar puesto que Kubernetes utiliza esta función para reenviar una aplicación que se ejecuta en la red interna a través del host del cliente de Kubernetes y, por defecto, se vincula a 127.0.0.1 y no acepta solicitudes de hosts externos. Es por eso que las aperturas de puertos no lograban abrir las peticiones.

Para que funcione la conexión, hay que hacer que este puerto escuche en "0.0.0.0". Esto se puede lograr fácilmente agregando un parámetro adicional al comando. La sintaxis se da a continuación.

```
kubectl port-forward svc/<service> <extport>:<service-port> --address='0.0.0.0' -n <namespace>
```

Que en este caso, sería lo siguiente para exponer el servicio MQTT con puerto 1883:

```
kubectl port-forward service/c2e-adapter-mqtt-vertx 86:1883 --address="0.0.0.0" -n cloud2edge
```

Entonces, esto daba lugar a la siguiente pregunta ¿Cuál es la diferencia entre 127.0.0.1 y 0.0.0.0?.

127.0.0.1 es la dirección de *loopback* (también conocida como *localhost*) y 0.0.0.0 hace que se aplique a todas las direcciones IPv4 en la máquina local. Si un host tiene dos direcciones IP, 192.168.1.1 y 10.1.2.1, y un servidor que se ejecuta en el host escucha en 0.0.0.0, será accesible en ambas direcciones IP.

De esta manera aceptará solicitudes de todos los hosts que se encuentren en la misma red local. Además, es necesario desactivar el firewall del equipo para que permita las conexiones y no ocurra ningún fallo por bloqueo de conexiones entrantes.

Una vez configurado esto, en el script del robot Husky para los sensores, tenemos que verificar que la IP que se encuentra declarada como *broker* es la misma que la IP que tiene asignada nuestro portátil en el punto de acceso y que el puerto no variará, siempre y cuando el túnel se haga en ese mismo puerto y no se cambie.

Ahora, cuando se ejecuta el script dentro de ROS con el comando *rosrun*, mostrará lo siguiente si todo ha funcionado de forma adecuada.

```
administrator@cpr-a200-0664:~/catkin_ws/src/tfg/src$ rosrun tfg sendgps.py
Connected to MQTT Broker!
Send {"topic": "sensor/gps/things/twin/commands/modify", "headers": {"content-type": "application/vnd.eclipse.ditto+json", "response-required
16666667}}, "longitude": {"properties": {"value": -4.4996715}}}, "path": "/features"} to topic telemetry
Send {"topic": "sensor/gps/things/twin/commands/modify", "headers": {"content-type": "application/vnd.eclipse.ditto+json", "response-required
16666667}}, "longitude": {"properties": {"value": -4.499671833333333}}}, "path": "/features"} to topic telemetry
Send {"topic": "sensor/gps/things/twin/commands/modify", "headers": {"content-type": "application/vnd.eclipse.ditto+json", "response-required
}}, "longitude": {"properties": {"value": -4.499672166666667}}}, "path": "/features"} to topic telemetry
Send {"topic": "sensor/gps/things/twin/commands/modify", "headers": {"content-type": "application/vnd.eclipse.ditto+json", "response-required
833333336}}, "longitude": {"properties": {"value": -4.4996725}}}, "path": "/features"} to topic telemetry
Send {"topic": "sensor/gps/things/twin/commands/modify", "headers": {"content-type": "application/vnd.eclipse.ditto+json", "response-required
833333336}}, "longitude": {"properties": {"value": -4.499672833333333}}}, "path": "/features"} to topic telemetry
Send {"topic": "sensor/gps/things/twin/commands/modify", "headers": {"content-type": "application/vnd.eclipse.ditto+json", "response-required
66666667}}, "longitude": {"properties": {"value": -4.499673166666667}}}, "path": "/features"} to topic telemetry
Send {"topic": "sensor/gps/things/twin/commands/modify", "headers": {"content-type": "application/vnd.eclipse.ditto+json", "response-required
66666667}}, "longitude": {"properties": {"value": -4.499673333333333}}}, "path": "/features"} to topic telemetry
Send {"topic": "sensor/gps/things/twin/commands/modify", "headers": {"content-type": "application/vnd.eclipse.ditto+json", "response-required
5}}, "longitude": {"properties": {"value": -4.499673666666666}}}, "path": "/features"} to topic telemetry
```

Figura 44. Envío de datos del tópico de ROS a nuestro clúster.

4.7 FASE 6: Representar datos en Grafana

Una vez recibidos los datos dentro del clúster y en la sombra creada, se dispone a compartir los datos con Grafana para poder darle una representación visual.

Pero esto no es posible hacerse de forma directa entre Eclipse Ditto y Grafana, por lo que será necesario el uso de intermediarios como Apache Kafka e InfluxDB. Esto se explicará de una forma menos detallada, pues lo más importante de nuestro Trabajo de Fin de Grado era la exploración de Minikube, las tecnologías del proyecto Eclipse y el sistema ROS.

4.7.1 Despliegue de Apache Kafka

Una vez desplegado este servicio en Minikube, se tendrá que generar un clúster dentro del mismo junto a tópico, que será el encargado de recibir los datos de Eclipse Ditto. Además, el servicio tendrá una dirección IP dentro del clúster de Minikube para poder vincularlo. Esta es una de las ventajas de Minikube, pues dentro de él, todo está conectado y se puede acceder de un servicio a otro.

Ya recogida la IP del servicio de Kafka, se procede a realizar esta instrucción dentro del Minikube que creará una conexión entre Ditto y Kafka para que el primero publique sus eventos en el segundo.

```
curl -i -X POST -u devops: ${DITTO_DEVOPS_PWD} -H 'Content-Type:
application/json' --data '{

 "targetActorSelection": "/system/sharding/connection",

 "headers": {

  "aggregate": false

 },

 "piggybackCommand": {

  "type": "connectivity.commands:createConnection",

  "connection": {

    "id": "kafka123",

    "connectionType": "kafka",

    "connectionStatus": "open",

    "failoverEnabled": true,

    "uri": "tcp://150.214.106.114:8041",

    "specificConfig": {

                    "bootstrapServers": "150.214.106.114:8041",

                    "saslMechanism": "plain"
```

```
    },
    "sources": [],
    "targets": [
            {
      "address": "husky",
      "topics": [
      "_/_/things/twin/events",
      "_/_/things/live/messages"
      ],
      "authorizationContext": [
        "nginx:ditto"
      ]
            }
    ]
  }
 }
}' http://XXX.XXX.XX.XX:XXXXX  /devops/piggyback/connectivity
```

4.7.2 Despliegue de InfluxDB y Telegraf

Estos se instalarán a través de repositorios Helm y, luego, habrá que introducir la IP de Kafka dentro de los valores del propio Telegraf para crear de esta manera la conexión entre ambos y pudiendo este último consumir los datos que le llegan. Luego, a través de un port-forward de InfluxDB que permitirá acceder al mismo desde el navegador, se configurará la conexión con Telegraf para que pueda escribirle los datos. Además, también se configurará la conexión con Grafana para su posterior envío.

Ya configurado, debería de aparecer lo siguiente en InfluxDB, donde podremos ver los datos de los sensores que están llegando de la sombra creada en Ditto.

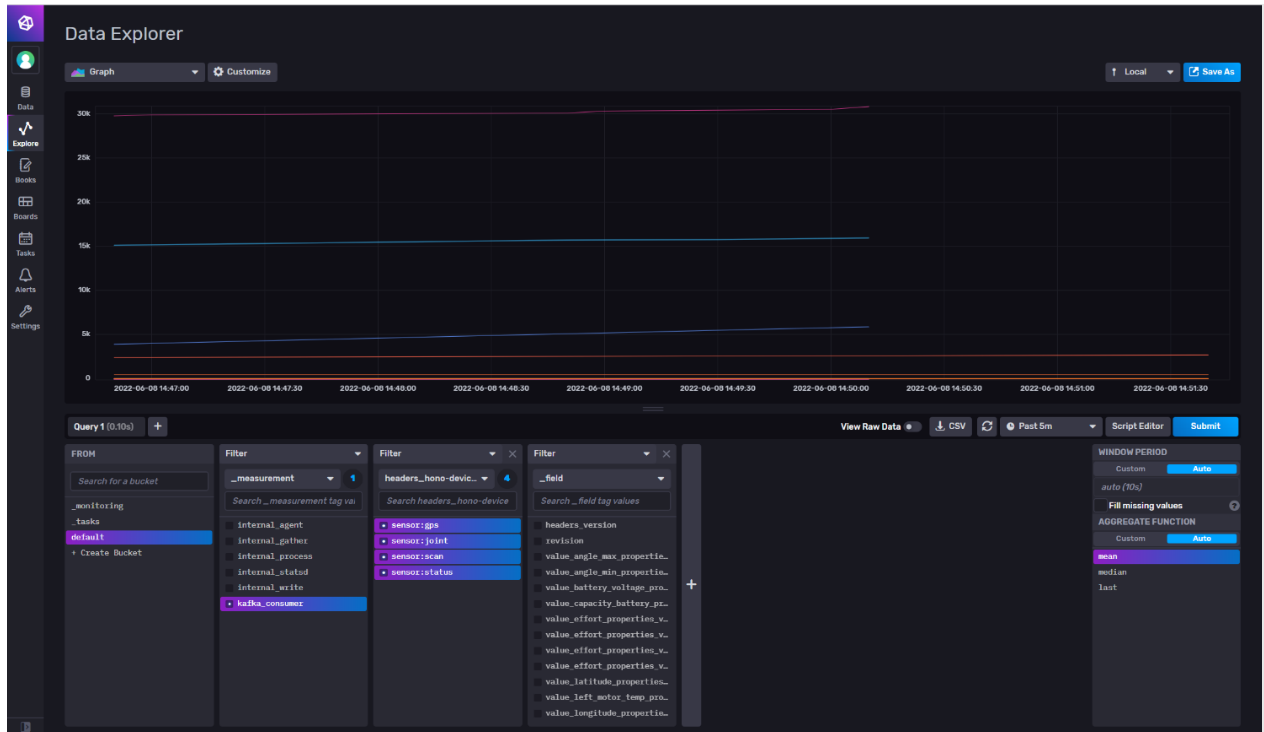


Figura 45. Plataforma influxDB desplegada y recibiendo datos.

4.7.3 Despliegue de Grafana

Este fue instalado de manera local en Windows, no dentro del propio clúster, ya que no era necesario. En los ajustes, se crea la conexión con InfluxDB para que reciba los datos que le están llegando a este y así poder crear un *Dashboard* con el modelo 3D junto a los datos.

Después de instalar el plugin comentado en el apartado 2.4, ya se podía cargar el modelo 3D creado en Unity del robot Husky, además, se configuraron paneles para el muestreo de los datos de forma organizada, quedando el siguiente resultado (Figura 46) cuando los scripts no están enviando datos.

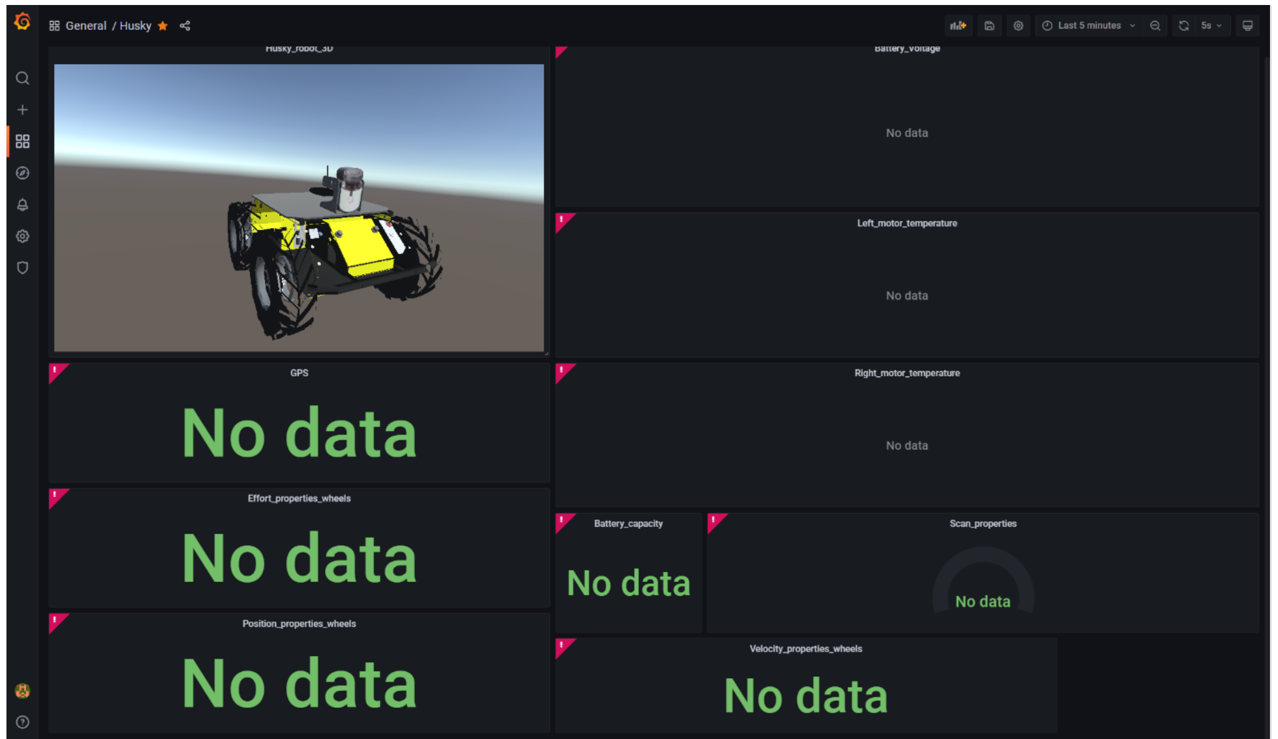


Figura 46. Representación en Grafana de la sombra digital sin recibir datos.

Y cuando ya los scripts están en funcionamiento y enviando los datos de cada componente del robot, da lugar al resultado final (Figura 47) de este proyecto.

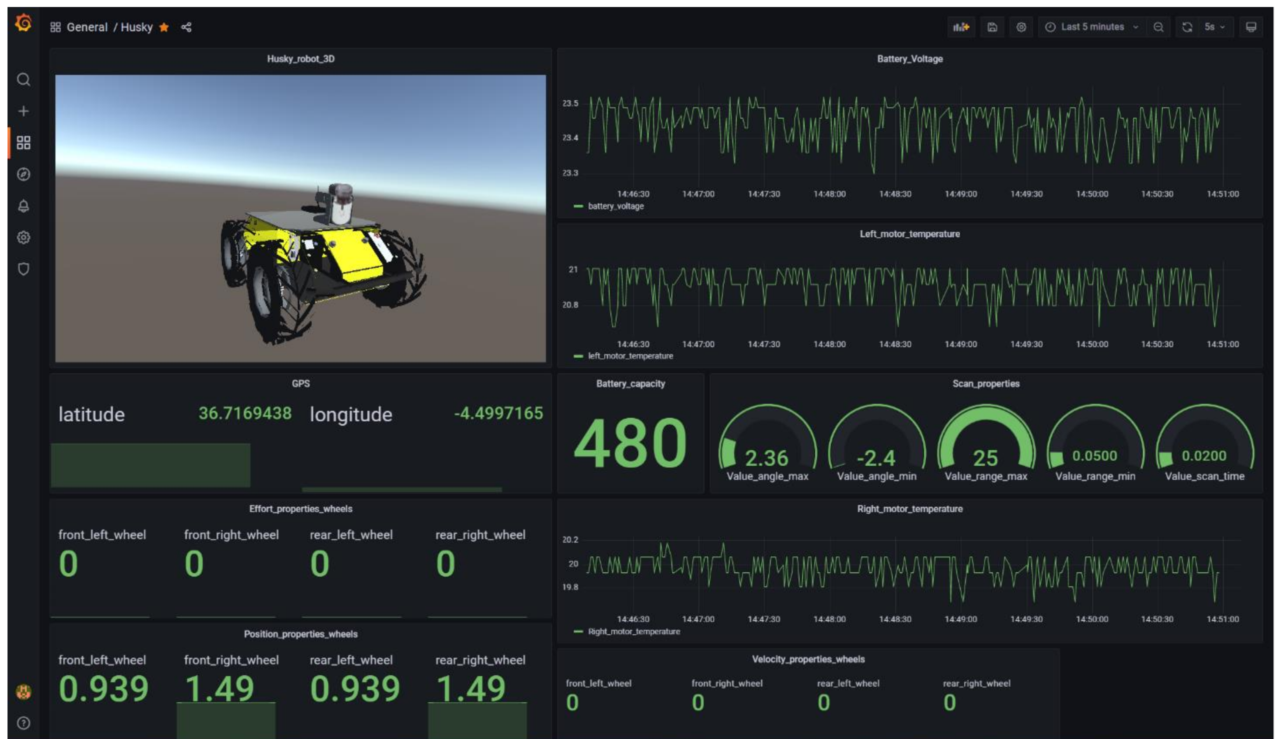


Figura 47. Representación en Grafana de la sombra digital recibiendo datos

En donde se puede ver como llegan de manera continua datos reales y actuales del robot Husky en ese mismo instante. Algunos de los datos son estáticos por defecto en el robot y solo nos proporciona la información que el mismo nos da. Otros, como el voltaje de la batería, temperatura de motores, etc., si se puede apreciar en las gráficas que están cambiando de forma constante.

Estos datos podrían ampliarse con la creación de nuevos scripts con distintos tópicos, en este caso solo se han hecho para cuatro tópicos para obtener una demostración, observar que todo funciona de forma óptima, y, por último, hacernos saber que se ha logrado crear una sombra digital.

5

Conclusiones y líneas futuras

Para finalizar esta memoria, se comentarán las conclusiones alcanzadas durante el desarrollo del proyecto y se plantearán líneas en las que se puede ampliar este proyecto en el futuro.

5.1 Desarrollo del proyecto

El desarrollo de este proyecto ha supuesto el realizar un aprendizaje y estudio profundo de herramientas y conceptos nunca antes utilizados por el autor, siendo algunos de ellos fundamentales por las tecnologías internas de los productos usados y otros elegidos debido a que eran los más óptimos para poder llevar a cabo el proyecto.

En primer lugar, se ha tenido que investigar acerca de la industria 4.0 y el mundo de los gemelos digitales, descubriendo una amplia gama de posibilidades

en el ámbito de los robots, pues la simulación es algo fundamental para avanzar de forma más rápida y eficaz.

Además, se ha tenido que llevar a cabo el aprendizaje del sistema ROS para poder crear la sombra digital, debido a que es el sistema que trae por defecto el robot Husky. Esto fue necesario para poder realizar el script y lograr de forma satisfactoria el envío de los datos de cada componente, consiguiendo de esta manera, un amplio conocimiento en el mundo de los robots y en la industria 4.0.

Por otro lado, se tuvo que encontrar las tecnologías necesarias para la creación de la sombra digital, siendo estas Eclipse Ditto y Eclipse Hono, un proyecto de Eclipse bastante moderno, por lo cual, no existen proyectos parecidos que sirvan como referencia para este Trabajo de Fin de Grado, dando valor a este producto, pues es novedoso. Para estas herramientas sirvieron los conocimientos previamente adquiridos por Julia Robles Medina, quien hizo de guía en el despliegue de estas.

Sin lugar a dudas, la parte que mayor esfuerzo y tiempo ha supuesto ha sido el despliegue de todo el proyecto de forma local, pues nunca antes se había hecho uso de tecnologías como son Kubernetes y Minikube, herramientas necesarias para el buen funcionamiento del producto final, pero que, a su vez, son demasiado complejas en el despliegue, pues requieren amplios conocimientos previos sobre redes y sistemas.

Del mismo modo, también ha sido necesario aprendizaje de tecnologías intermediarías, como son Apache Kafka, InfluxDB, Telegraf y el protocolo MQTT. Pues estas en sí no suponen la base de estudio de este proyecto, pero son fundamentales para el funcionamiento del mismo.

Así mismo, para poder visualizar la sombra digital, se adquirieron conocimientos en Unity para el desarrollo del modelo 3D y también en la plataforma de Grafana para poder visualizar los datos junto al modelo creado, formando así de esta manera, un producto visualmente atractivo.

En definitiva, como resultado de este Trabajo de Fin de Grado se logró el primer paso hacía un gemelo digital, creando así de esta manera, la llamada sombra digital. Este producto final, permite a cualquier persona interesada observar el robot y sus datos de una forma que no sea física y poder descubrir, de esta manera, posibles mejores o fallas sin estar cerca del mismo. Es por eso que todo ha sido detallado para quien, con esta memoria, quiera replicarlo e incluso expandirlo.

5.2 Líneas futuras

Visto el buen funcionamiento, este producto aún puede ser ampliado y mejorado en un futuro, planteando las siguientes líneas futuras.

En primer lugar, en un nuevo proyecto se podría aplicar Machine Learning a los datos recibidos de los componentes, para de esta manera poder predecirlos en caso de desconexión con el robot Husky. De hecho, esta tecnología se pretendía añadir al principio de este Trabajo de Fin de grado pero, debido a la complejidad añadida junto a la creación de este mismo proyecto, esto no fue posible.

Otra propuesta interesante sería poder interactuar directamente con el robot Husky. Esto sería posible con la inclusión en Grafana de unos comandos de Unity, los cuales, al pulsarlos, interactúen directamente sobre el robot físico. Estos comandos podrían ser, por ejemplo, botones que indiquen los posibles movimientos del mismo. Además, esto también sería posible con la arquitectura

utilizada, ya que se puede tanto enviar como recibir datos. Esto supondría la creación final de un gemelo digital, ya que se habla de recepción y envío de datos.

6

Referencias

[1] A. Bilberg and A. A. Malik, “Digital twin driven human–robot collaborative assembly,” *CIRP Annals*, vol. 68, no. 1, pp. 499–502, Jan. 2019, doi: 10.1016/J.CIRP.2019.04.011.

[2] G. Schuh, C. Kelzenberg, J. Wiese, and T. Ochel, “Data structure of the digital shadow for systematic knowledge management systems in single and small batch production,” *Procedia CIRP*, vol. 84, pp. 1094–1100, 2019, doi: 10.1016/J.PROCIR.2019.04.210.

[3] R. P. Nikula, M. Paavola, M. Ruusunen, and J. Keski-Rahkonen, “Towards online adaptation of digital twins,” *Open Engineering*, vol. 10, no. 1, pp. 776–783, Jan. 2020, doi: 10.1515/eng-2020-0088.

[4] “Husky UGV - Robot de investigación de campo para exteriores de Clearpath.” https://clearpathrobotics.com/husky-unmanned-ground-vehicle-robot/ (accessed May 27, 2022).

[5] “es - ROS Wiki.” http://wiki.ros.org/es (accessed Jun. 02, 2022).

[6] "Cómo funciona la Robótica Industrial y cómo se programa." https://www.cursosaula21.com/como-funciona-la-robotica-industrial/ (accessed May 25, 2022).

[7] "minikube start | minikube." https://minikube.sigs.k8s.io/docs/start/ (accessed Jun. 22, 2022).

[8] "Eclipse Ditto™ • marco de código abierto para gemelos digitales en IoT." https://www.eclipse.org/ditto/ (accessed Jun. 02, 2022).

[9] "Connect, Command & Control IoT devices :: Eclipse Hono™." https://www.eclipse.org/hono/ (accessed Jun. 02, 2022).

[10] "¿Qué es Python? | Blog Becas Santander." https://www.becas-santander.com/es/blog/python-que-es.html (accessed Jun. 02, 2022).

[11] "MQTT - The Standard for IoT Messaging." https://mqtt.org/ (accessed Jun. 22, 2022).

[12] "Qué es Grafana y primeros pasos | OpenWebinars." https://openwebinars.net/blog/que-es-grafana-y-primeros-pasos/ (accessed Jun. 02, 2022).

[13] "Plataforma de desarrollo en tiempo real de Unity | Motor de VR y AR en 3D y 2D." https://unity.com/es (accessed Jun. 02, 2022).

[14] "blender.org - Home of the Blender project - Free and Open 3D Creation Software." https://www.blender.org/ (accessed Jun. 02, 2022).

[15] "Kubernetes." https://kubernetes.io/ (accessed Jun. 22, 2022).

[16] "¿Qué es el Internet de las cosas (IoT)? | Oracle España." https://www.oracle.com/es/internet-of-things/what-is-iot/ (accessed Jun. 22, 2022).

[17] H. Takeuchi and I. Nonaka, “The New New Product Development Game”, Accessed: May 27, 2022. [Online]. Available: https://hbr.org/1986/01/the-new-new-product-development-game

[18] “Ubuntu - Wikipedia, la enciclopedia libre.” https://es.wikipedia.org/wiki/Ubuntu (accessed Jun. 02, 2022).

[19] “melodic - ROS Wiki.” http://wiki.ros.org/melodic (accessed Jun. 22, 2022).

[20] “ROS/Concepts - ROS Wiki.” http://wiki.ros.org/ROS/Concepts#ROS_Filesystem_Level (accessed Jun. 03, 2022).

[21] “IEEE Xplore Full-Text PDF:” https://ieeexplore--ieee--org.uma.debiblio.com/stamp/stamp.jsp?tp=&arnumber=9119497&tag=1 (accessed Jun. 18, 2022).

[22] “Home - Docker.” https://www.docker.com/ (accessed Jun. 22, 2022).

[23] “Helm.” https://helm.sh/ (accessed Jun. 22, 2022).

[24] “Overview.” https://www.eclipse.org/packages/packages/cloud2edge/ (accessed Jun. 06, 2022).

[25] “Apache Kafka.” https://kafka.apache.org/ (accessed Jun. 22, 2022).

[26] “Telegraf Open Source Server Agent | InfluxDB.” https://www.influxdata.com/time-series-platform/telegraf/ (accessed Jun. 22, 2022).

[27] “InfluxDB: Open Source Time Series Database | InfluxData.” https://www.influxdata.com/ (accessed Jun. 02, 2022).

[28] "Service | Kubernetes." https://kubernetes.io/es/docs/concepts/services-networking/service/ (accessed Jun. 05, 2022).

[29] "Redirección de puertos - Wikipedia, la enciclopedia libre." https://es.wikipedia.org/wiki/Redirecci%C3%B3n_de_puertos (accessed Jun. 22, 2022).

[30] "Localtunnel ~ Expose yourself to the world." https://localtunnel.github.io/www/ (accessed Jun. 05, 2022).

[31] "Software de diseño 3D | Modelado 3D en la web | SketchUp." https://www.sketchup.com/es (accessed Jun. 22, 2022).

[32] "Docker (software) - Wikipedia, la enciclopedia libre." https://es.wikipedia.org/wiki/Docker_(software) (accessed Jun. 14, 2022).

[33] A. M. Potdar, D. G. Narayan, S. Kengond, and M. M. Mulla, "Performance Evaluation of Docker Container and Virtual Machine," *Procedia Computer Science*, vol. 171, pp. 1419–1428, 2020, doi: 10.1016/J.PROCS.2020.04.152.

[34] B. Burns, J. Beda, and K. Hightower, "Kubernetes: Up and Running".

[35] M. M. Khalel, M. Arul Pugazhendhi, and G. R. Raj, "Enhanced Load Balancing in Kubernetes Cluster By Minikube," *2022 International Conference on Smart Technologies and Systems for Next Generation Computing (ICSTSN)*, pp. 1–5, Mar. 2022, doi: 10.1109/ICSTSN53084.2022.9761317.

[36] "Eclipse Ditto™ documentation overview • Eclipse Ditto™ • a digital twin framework." https://www.eclipse.org/ditto/intro-overview.html (accessed Jun. 22, 2022).

[37] "Documentation :: Eclipse Hono™." https://www.eclipse.org/hono/docs/ (accessed Jun. 22, 2022).

[38] "Eclipse Mosquito." https://mosquitto.org/ (accessed Jun. 02, 2022).

[39] "Eclipse Paho | The Eclipse Foundation." https://www.eclipse.org/paho/index.php?page=clients/python/index.php (accessed Jun. 02, 2022).

[40] "Apache Kafka." https://kafka.apache.org/documentation/ (accessed Jun. 22, 2022).

[41] "InfluxDB: Open Source Time Series Database | InfluxData." https://www.influxdata.com/products/influxdb-overview/ (accessed Jun. 22, 2022).

[42] "Documentation | Grafana Labs." https://grafana.com/docs/ (accessed Jun. 22, 2022).

[43] "Visual Studio Code - Code Editing. Redefined." https://code.visualstudio.com/ (accessed Jun. 22, 2022).

[44] "Ubuntu - Wikipedia, la enciclopedia libre." https://es.wikipedia.org/wiki/Ubuntu (accessed Jun. 22, 2022).

[45] "¿Qué es Python? | Blog Becas Santander." https://www.becas-santander.com/es/blog/python-que-es.html (accessed Jun. 22, 2022).

[46] "3D Builder, el software de modelado 3D gratuito de Microsoft - 3Dnatives." https://www.3dnatives.com/es/3d-builder-software-gratuito-microsoft/ (accessed Jun. 22, 2022).

[47] "Introducción a Windows Terminal | Microsoft Docs." https://docs.microsoft.com/es-es/windows/terminal/ (accessed Jun. 22, 2022).

[48] "Netplan: Configurar la red en Ubuntu 20.04 - ochobitshacenunbyte." https://www.ochobitshacenunbyte.com/2021/04/26/netplan-configurar-la-red-en-ubuntu-20-04/ (accessed Jun. 22, 2022).

[49] "How to use MQTT in Python (Paho) | EMQ." https://www.emqx.com/en/blog/how-to-use-mqtt-in-python (accessed Jun. 10, 2022).

[50] "ROS/Tutorials/WritingPublisherSubscriber(python) - ROS Wiki." http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29 (accessed Jun. 08, 2022).

[51] "es - ROS Wiki." http://wiki.ros.org/es (accessed Jun. 08, 2022).

[52] "husky_msgs/HuskyStatus Documentation." http://docs.ros.org/en/indigo/api/husky_msgs/html/msg/HuskyStatus.html (accessed Jun. 08, 2022).

[53] "Take a tour." https://www.eclipse.org/packages/packages/cloud2edge/tour/ (accessed Jun. 09, 2022).

[54] "HTTP API • Eclipse Ditto™ • a digital twin framework." https://www.eclipse.org/ditto/2.0/http-api-doc.html#/ (accessed Jun. 09, 2022).

Apéndice A

Manual de Usuario

Ya desarrollado el proyecto, para cualquier usuario que quiera iniciarlo y visualizar los datos en Grafana, deberá de realizar los siguientes pasos para ponerlo en funcionamiento. Todos estos pasos suponen que el robot y el portátil contenedor del clúster están conectados al mismo punto de acceso.

1. Comprobar que la dirección IP del ordenador contenedor del clúster no ha cambiado dentro de la red wifi, si esto ocurre, habrá que cambiar la dirección IP a la que envían los scripts en ROS.
2. Iniciar el clúster de Minikube.

```
Minikube start
```

3. Crear un túnel del servicio **c2e-adapter-mqtt-vertx** de forma que sea visible en la LAN para recibir los datos.

```
kubectl port-forward service/c2e-adapter-mqtt-vertx 86:1883 --
address="0.0.0.0" -n cloud2edge
```

4. Crear un túnel de InfluxDB para que le lleguen los datos a través de Kafka y esté expuesto para que Grafana pueda consumir de él. Este puede estar simplemente en el localhost.

```
kubectl port-forward service/influxdb-influxdb2 9001:80 -n cloud2edge
```

5. Acceder al sistema ROS interno del robot y ejecutar el script de los datos que quiera enviar.
6. Abrir Grafana para poder visualizar los datos

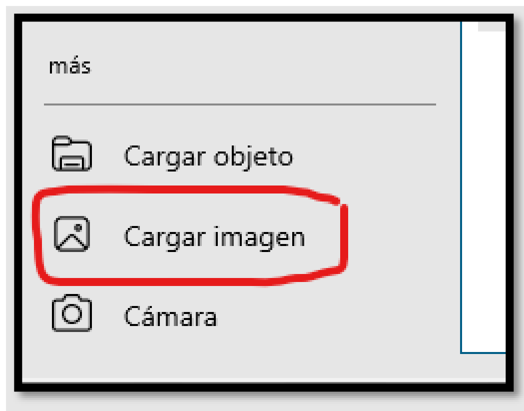
Para todo esto, hay que tener en cuenta no modificar el puerto que se expone al crear el túnel una vez configurado, o, si se cambian, tenerlo en cuenta para la configuración de sincronización como pueden ser los puertos de otros servicios y los scripts de ROS.

Apéndice B

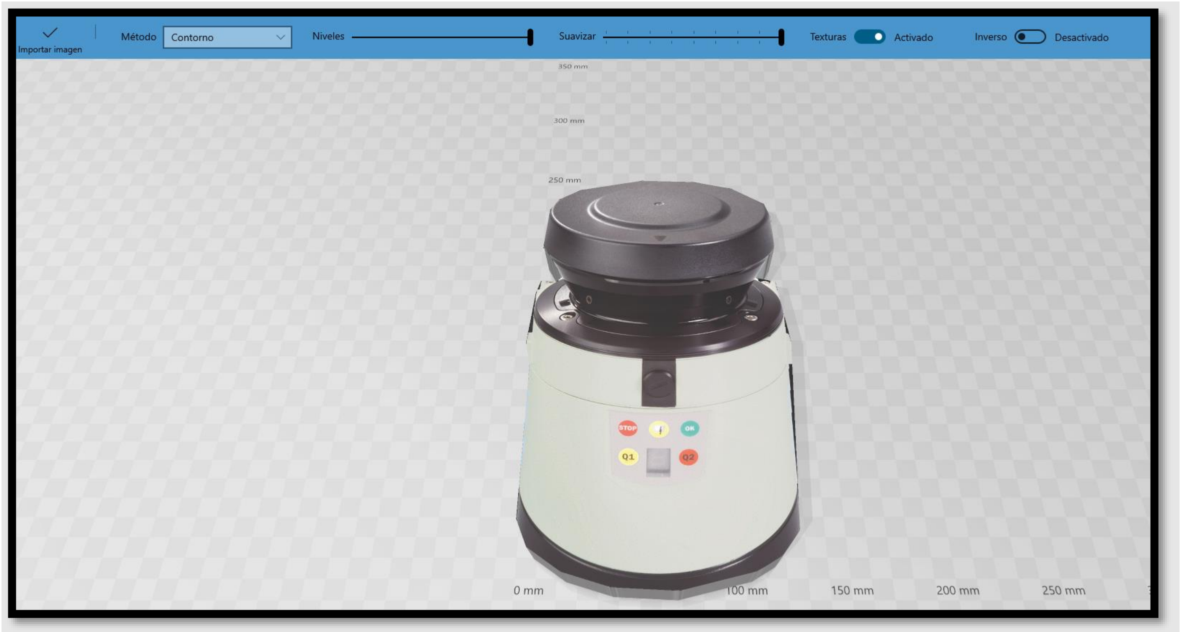
Creación de objetos 3D en Blender

Para crear un objeto en 3D a partir de una imagen, se necesita realizar los siguientes pasos:

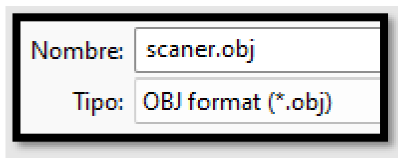
1. Obtener la imagen en formato .PNG y sin fondo del objeto que se quiera convertir a 3D.
2. Abrir el programa 3D BUILDER y cargar la imagen que se quiera trabajar.



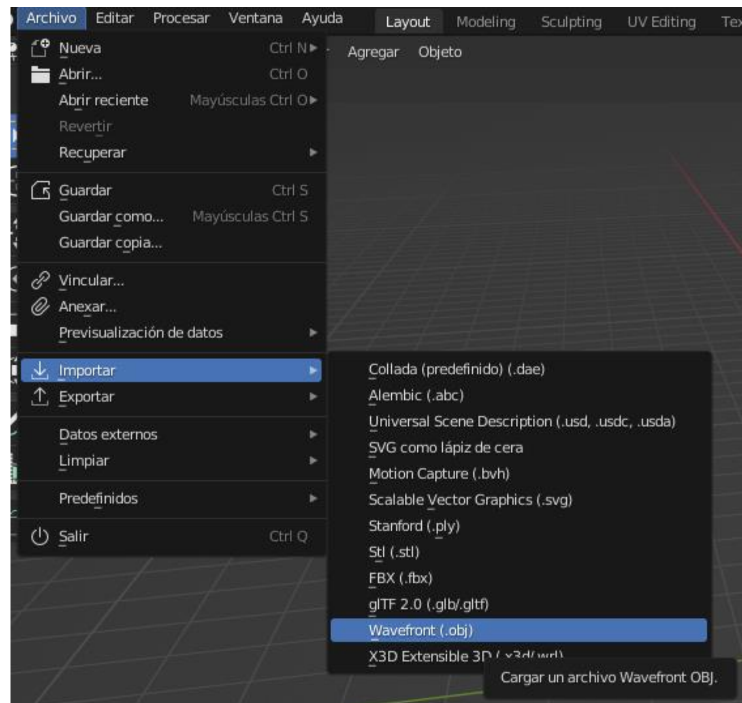
3. Una vez cargada la imagen, hay que ajustar de esta manera los parámetros de "niveles" y "suavizar" hasta que el modelo 3D quede de una forma óptima. Una vez ajustado, se pulsa en "importar".



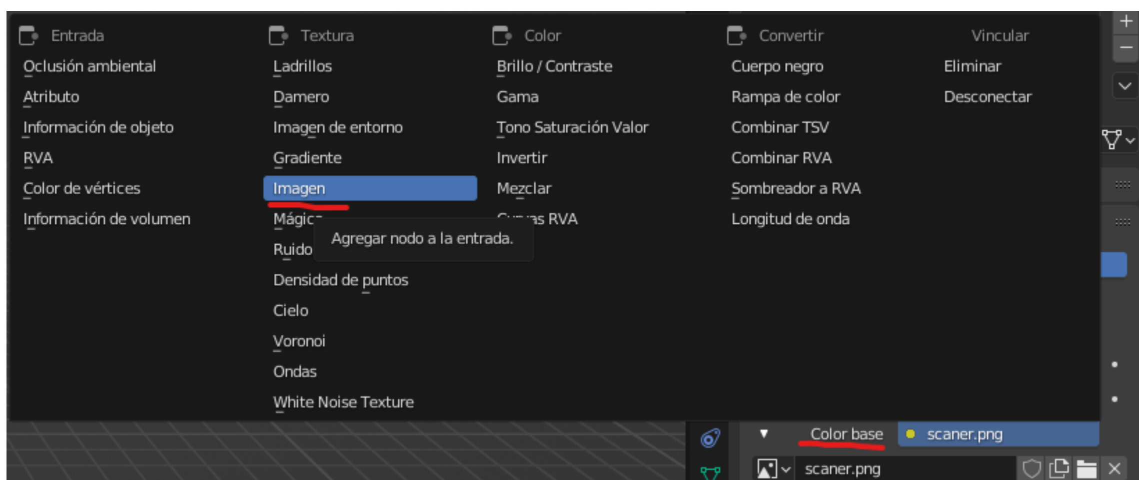
4. Ya ajustado todo, se guarda el objeto creado en 3D con la extensión .obj para poder trabajarlo posteriormente en BLENDER.



5. Se abre el programa BLENDER y se importa el objeto con la extensión .obj creado previamente.

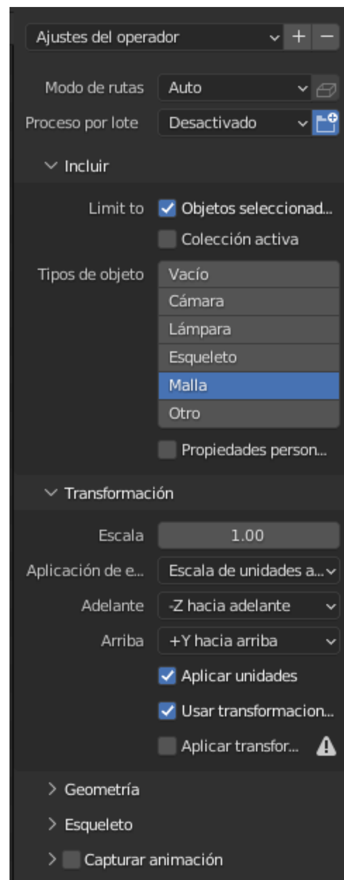


6. Por defecto, aparece el objeto en 3D pero sin textura alguna, totalmente solido (gris), ya que no trae el material. Para que el material sea igual que el del objeto que se quiere, hay que dirigirse a “Propiedades de materiales” y en “Color base” elegir la opción “Textura -> Imagen” y seleccionar el archivo .PNG de la imagen principal del objeto a crear.



7. Con toda esta configuración quedaría el objeto tal que así.

8. Para exportarlo, se selecciona el objeto y hay que dirigirse a “Archivo -> exportar -> FBX (.fbx)”, donde aparecerá una ventana para ajustar la salida del archivo y en la zona de ajustes de operador tiene que tener las opciones tal que así para que se exporten de manera correcta para poder visualizarlo en la plataforma de Unity:



Luego pulsamos en “Exportar FBX” y listo, ya tendríamos nuestro nuevo objeto preparado para importarlo en Unity y poder trabajar con él.

UNIVERSIDAD DE MÁLAGA | uma.es

E.T.S. DE INGENIERÍA INFORMÁTICA

E.T.S de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga