ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADO EN INGENIERÍA INFORMÁTICA

Estudio y aplicación de la virtualización ligera para dispositivos de internet de las cosas (IoT)

Study and implementation of lightweight virtualization for internet of things devices.

Realizado por
Pablo García Acosta

Tutorizado por
Mercedes Amor Pinilla

Departamento
Lenguajes y ciencias de la computación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, JUNIO DE 2019

Fecha defensa: Julio de 2019

Fdo. El/la Secretario/a del Tribunal

Resumen

Este trabajo cuenta con 2 partes, la primera es un trabajo de investigación para comprobar cuales son las tecnologías de virtualización ligera en la actualidad. En este trabajo de investigación se revisarán que tecnologías existen, describir sus características principales y mostrar los resultados mediante una comparativa.

La segunda, que es la ha ocupado la mayor parte del tiempo, consiste en el desarrollo de un sistema que incluye un cluster, formado por 2 raspberrys pi 3, que muestra cómo hacer uso de la tecnología de virtualización ligera en combinación con el uso de microservicios. El sistema recoge datos de un sensor de humedad/temperatura, un resistor fotovoltaico y un sensor de movimiento; estos datos son enviados por nodos NodeMCU a través del protocolo MQTT a unos microservicios que se están ejecutando en el cluster en contenedores Docker, orquestados por Docker Swarm. Cada microservicio se encarga de gestionar los datos de un sensor y de la comunicación entre la página web, a través de la cual se muestran al usuario los datos tomados por los sensores.

El usuario, a través de un servicio, puede configurar la frecuencia a la que se toman datos de los sensores.

Palabras clave:

IoT, Docker, Docker Swarm, MQTT

Abstract

This work has 2 parts, the first is a research work to check what are the technologies of lightweight virtualization nowadays. In this research work we will review what technologies exist, describe their main characteristics and show the results through a comparison.

The second part, which is the one that has occupied most of the time, consists in the development of a system that includes a cluster, composed of 2 raspberrys pi 3, which shows how to make use of light virtualization technology in combination with the use of microservices. The system collects data from a humidity / temperature sensor, a photovoltaic resistor and a motion sensor connected to different NodeMCU nodes. Once collected, this data is sent through the MQTT protocol to some microservices that are running in the cluster in Docker containers, orchestrated by Docker Swarm. Each microservice is responsible for managing the data of a sensor and to provide sensed data to a web page, which allows the user to consult the data sent by the sensors.

The user, through a web service, can configure the rate at which data is taken from the sensors.

Key words

IoT, Docker, Docker swarm, MQTT

Índice

Índice de figuras

Capítulo 1

Introducción

1.1 Motivación

Cada día nos encontramos con más dispositivos conectados y la tendencia es que siga subiendo con la llegada de las redes 5G. Esta gran cantidad de dispositivos conectados pueden suponer un problema a la hora de gestionarlo ya bien sea por la cantidad de datos que hay que manejar, porque la red esté saturada y no pueda manejar toda la información en el tiempo que se espera o directamente porque la red no esté disponible en ese momento.

Por ello se ha querido desarrollar un trabajo de fin de grado en el que se desarrolle una pequeña nube privada que sea capaz de gestionar los datos generados por una casa conectada o por cualquier otro tipo de sistema que requiera de dispositivos conectados. De esta forma poder distribuir la carga que tendría que soportar la red si tuviera que transportar los datos desde donde se generan hasta un centro de datos (*datacenter*), manejándose en local la mayoría de los datos generados.

Además, se ha añadido la dificultad de aprender a utilizar tecnologías de virtualización ligera como Docker en este caso y tecnologías de orquestación de contenedores como es Kubernetes o Docker Swarm para este caso. Incluyendo también el aprendizaje del uso de los sensores utilizados y su comunicación.

1.2 Objetivos

Los objetivos son dos:

Primero, realizar un trabajo de investigación sobre cuáles son las tecnologías de virtualización ligera existentes hoy en día, dar una breve explicación de cada una de ellas y compararlas para ver qué ventajas ofrecen cada una de ellas. Con este apartado intentaremos conseguir una mejor visión de cuáles son las posibilidades que tenemos disponibles actualmente.

EL segundo objetivo es construir un cluster con 2 raspberry pi 3 en el que se ejecute una serie de microservicios virtualizados con Docker y orquestado por Docker Swarm en el cual podamos observar cómo se orquestarían los contenedores y ver que hasta qué punto sería viable y rentable o no construir un cluster para tener una nube privada.

Sobre la placa esp8266 se escribirá un programa que lea los sensores y reenvíe es información al cluster.

Sobre las rapsberrys, como ya se ha mencionado anteriormente, la construcción de un cluster que soporte sistemas de virtualización ligera. En el cluster habrá un broker de MQTT (protocolo de comunicación de la IoT) que gestione todo el paso de mensajes, un sitio web que permita acceder a la información recogida por los sensores, un microservicio que sirva de capa intermedia de tratamiento de los datos y comunicación entre los sensores y la aplicación web que muestra los datos.

1.3 Descripción de la estructura de la memoria

Esta memoria está organizada en 5 capítulos. A continuación, se mostrará un breve resumen de cada capítulo:

- **Capítulo 1 – Introducción:** este capítulo trata de una pequeña introducción sobre de que trata el trabajo, los objetivos y las motivaciones que han llevado a realizarlo.
- **Capítulo 2 – Análisis de las Tecnologías utilizadas:** se enumerarán cada una de las tecnologías utilizadas, tanto hardware como software, necesarias para le realización de este trabajo. Además de una pequeña explicación de cada componente.
- **Capítulo 3 – Trabajo de investigación:** este capítulo integrará la primera parte de este trabajo de fin de grado. Una investigación de las tecnologías de virtualización ligeras que actualmente encontramos en el mercado. Como este trabajo fue desarrollado a parte contará con su propia introducción y conclusión.
- **Capítulo 4 – Diseño e implementación:** este cuarto capítulo abordará la gran mayoría de la parte práctica de este trabajo, viendo desde cuales

han sido las conexiones físicas realizadas hasta el código utilizado para los microservicios, y los problemas encontrados en la etapa de desarrollo.

- **Capítulo 5 – Conclusiones:** este capítulo final tratará de las conclusiones alcanzadas tras la realización del trabajo y de la proposición de unas posibles líneas de trabajo futuras.

1.4 Metodología de trabajo

Para la realización de este trabajo de fin de grado se ha elegido utilizar la metodología ágil Kanban, ya que es una de las metodologías que menos documentación genera además que tanto la fase diseño y la de implementación se puede modularizar con una gran facilidad haciéndolo perfecto para esta metodología.

Capítulo 2

Trabajo de investigación

2.1 Introducción

Hoy en día el Internet de las Cosas (desde ahora IoT por sus siglas en inglés *Internet of Things*) está creciendo rápidamente y cada vez hay más dispositivos conectados. Esto crea un reto a la hora de procesar toda la información que se genera constantemente sobre todo si se desea una respuesta a tiempo real. Por ejemplo, se estima que un coche autónomo genere 1 Gigabyte de datos por segundo, de forma que si es necesario dar una respuesta a tiempo real muchas veces esto será imposible bien sea porque hay una congestión en el acceso al Cloud o en ese momento el ancho de banda no sea el suficiente. Por ello una alternativa al proceso de datos en el Cloud (Nube) es lo que se denomina edge computing (computación en el borde o frontera).

Edge computing se refiere a las tecnologías que permiten procesar la información donde es generada, en el borde (edge en inglés) de la red, disminuyendo de esta forma la cantidad de datos que tienen que ser enviados al cloud para ser procesados. Definimos como borde de la red a cualquier recurso de cómputo o de red que se encuentre a lo largo del punto donde se generan los datos y el “cloud data center”.

Aquí es donde la virtualización ligera entra en juego, ya que a diferencia de los softwares de virtualización tradicional, también denominados *hipervisores*, como KVM, VMware o Virtualbox, esta requiere menos recursos y se despliega más rápido, puesto que solo virtualiza la aplicación y no el sistema operativo completo.

2.2 Qué es un contenedor

Un contenedor es parecido a una aplicación que se ejecuta como un proceso normal del sistema operativo que se aísla del resto usando su propio espacio de

direcciones. Concretamente es algo más que una aplicación normal. ya que además del ejecutable incluye todo el software de sistema necesario para que se pueda ejecutar, como librerías, dependencias etc. Entre las características de los contenedores hay dos que hacen posible que dos contenedores se estén ejecutando en la misma máquina física, los “Cgroups” y los “namespaces”.

Los “Cgroups”, también conocidos como “Control groups”, es una característica del kernel de linux que controla los recursos asignados. Permite al administrador del sistema asignar recursos como CPU, memoria, red o alguna combinación de ellos para ejecutar los contenedores. Los recursos pueden ser asignados dinámicamente y cada contenedor no puede usar más de lo que se le haya asignado en los “Cgroups”.

Los “namespaces” proporcionan una abstracción de los recursos del kernel, como el ID del proceso, interfaces de red o nombres de host para que de esta forma cada contenedor parezca tener sus propios recursos reservados. Por ejemplo, un contenedor no entrará en conflicto con el resto incluso cuando estos tengan el mismo ID de proceso.

2.3 Diferencia entre una máquina virtual tradicional y un contenedor

Hay 3 grandes diferencias entre estas dos tecnologías.

1. Un contenedor es *más ligero* que una máquina virtual. Un contenedor solo incluye los ejecutables y las librerías y dependencias que necesite para su ejecución. Mientras que una máquina virtual contiene hasta el sistema operativo: diferentes máquinas virtuales tienen diferentes sistemas operativos; los contenedores que se encuentren en una misma máquina comparten el mismo sistema operativo. Una máquina virtual puede ejecutar un sistema operativo diferente al de la máquina anfitriona, sin embargo, un contenedor tiene que utilizar el mismo sistema operativo.

2. En un hipervisor, o monitor de máquina virtual, como puede ser VMware o KVM, es necesario un entorno de virtualización que no es necesario con los contenedores. Una máquina virtual tiene que poder actuar como una máquina independiente, manteniendoel control de todos sus recursos. Por esto, cada instrucción de una máquina virtual tiene que ser traducida para que pueda ser ejecutada por la máquina anfitriona. Por otro lado, un contenedor se comunica directamente con la máquina anfitriona con lo que no es necesario esa capa de traducciones. En la siguiente figura se puede observar esta diferencia.

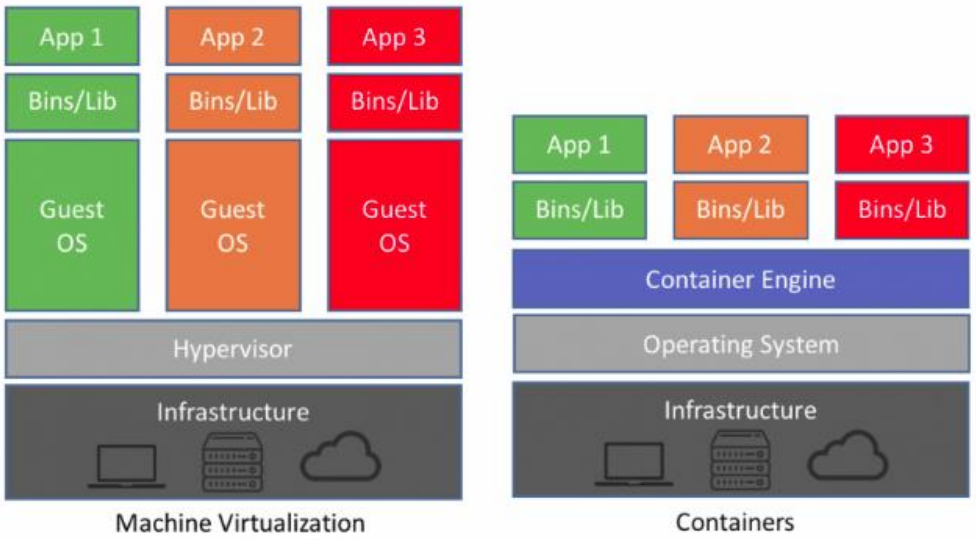


Figura 1 - Comparación entre contenedor y máquina virtual

3. Cada máquina virtual tiene su propia imagen mientras que los contenedores pueden compartir la misma imagen. Las imágenes de los contenedores están creadas por capas, de forma que en vez de crear otra imagen se creará una capa con los cambios y se superpone con la imagen original.

2.4 Unikernels

Son la forma de virtualización más nueva, se trata de imágenes de máquina con un único propósito, ejecutar una aplicación en concreto. Se construyen con librerías de sistema operativo, es un método para construir sistemas operativos *mínimos* donde el kernel y la aplicación se ejecutan en el mismo espacio de direcciones, y a través de una compilación cruzada con el código de la aplicación y la configuración. De esta forma se crea una aplicación totalmente

independiente capaz de ejecutarse completamente por sí misma. Además, a la hora de creación solo se seleccionan aquella librerías y rutinas necesarias eliminando todo lo innecesario.

Con esto podemos ver que los unikernels:

- Son estructuras inmutables, una vez creadas no se pueden modificar.
- Solo hay un espacio de direcciones en el que se encuentran el kernel y la aplicación.

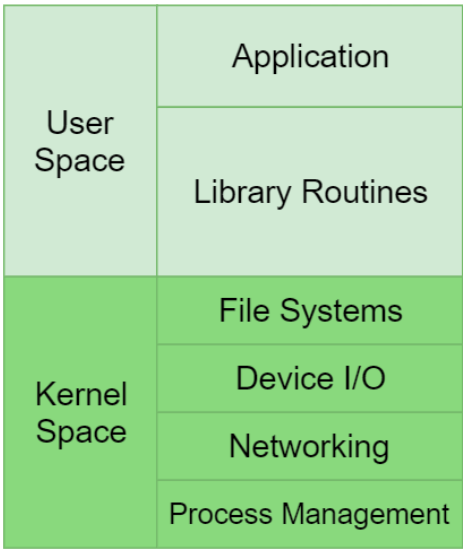


Figura 2 - pila de aplicación en un sistema operativo (Russel C. Pavlicek)

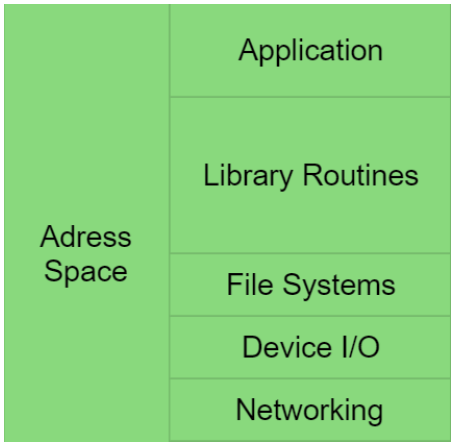


Figura 3 - pila de aplicación en unikernels (Russel C. Pavlicek)

Las imágenes resultantes son de un tamaño muy pequeño y requieren de una cantidad baja de recursos para su ejecución.

2.5 Unikernels vs Máquinas virtuales vs Contenedores

Una gran diferencia nos la encontramos en la infraestructura de cada una cómo se puede ver en la siguiente figura:

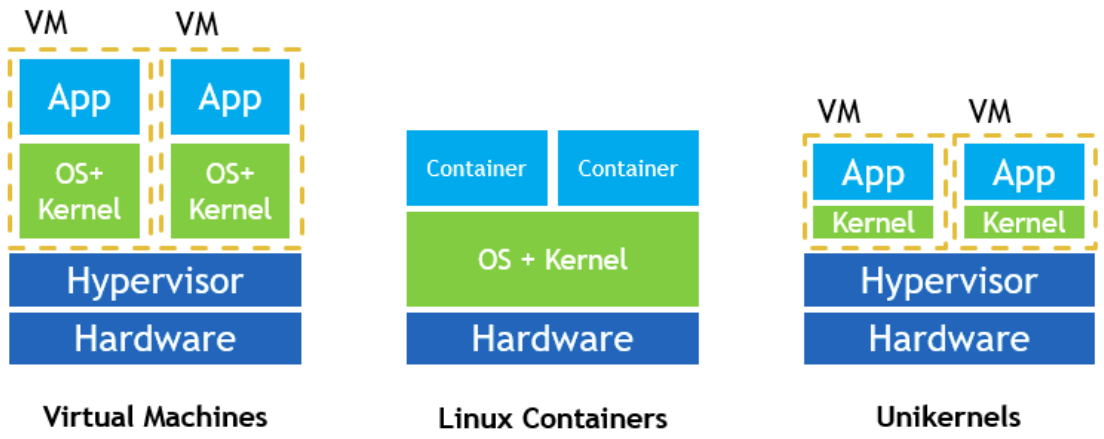


Figura 4 - Diferencia entre máquina virtual, contendor y unikernel

Se puede observar que las máquinas virtuales son las que requieren de una mayor infraestructura que los contenedores y los unikernels.

	Tiempo de inicialización	Tamaño de la imagen	Nivel de aislamiento*	Alta disponibilidad
Máquinas virtuales	~5/10 secs	~1000 MBs	hipervisor (En los bare-metal)	Si
Contenedores	~800/100 msecs	~50 MBs	Host Kernel	En general no*
Unikernels	~< 50 msecs	~< 5 MBs	hipervisor	No

*En este caso los contenedores comparten el mismo Kernel (el del host), por lo que si el kernel se ve comprometido se comprometen todos los contenedores. En este caso el hipervisor ofrece un mayor nivel de aislamiento

* Algunas tecnologías ofrecen algunos mecanismos de alta disponibilidad.

	Ventajas	Desventajas
Máquinas virtuales	Mayor aislamiento del host Existen soluciones de orquestación Única opción para desplegar sistemas windows	La opción menos deseable para IoT Cada instancia necesita un sistema operativo completo Necesidad de hardware más potente
Contenedores	Existen opciones de orquestación Asignación dinámica de recursos	Dependen del kernel del host Aun existiendo diferentes tecnologías con mucha diferencia es Docker la que tiene más documentación
Unikernels	Depende del conocimiento de un lenguaje de programación Mayor seguridad debido a que la superficie de ataque es mínima Mayor aislamiento del host	Está en una fase temprana de desarrollo Al ser distintos entre sí cada unikernel debe ser tratado de forma individual en términos de seguridad

2.6 Tecnologías de “contenerización”

Linux Containers (LXC)

Es un sistema de contenedores que intenta “simular” el funcionamiento de una máquina virtual convencional, pero sin tener que ejecutar otro kernel y tener virtualizar un hardware completo. Este funcionamiento se consigue gracias al uso de características como los “cgroups” o el uso de “namespaces” mencionados anteriormente.

Si ponemos un ejemplo de una aplicación web en la cual hay una base de datos con 2 aplicaciones diferentes, en docker habría un contenedor para cada uno mientras que en LXC lo más normal sería un contenedor para la base de datos y otro contenedor para las 2 aplicaciones.

OpenVZ

OpenVZ es un sistema de contenedores para Linux. OpenVZ crea múltiples contenedores de Linux seguros y aislados (también conocidos como VES o VPS) en un único servidor físico, lo que permite una mejor utilización del servidor y garantiza que las aplicaciones no entren en conflicto. Cada contenedor realiza y ejecuta exactamente como un servidor independiente; un contenedor puede reiniciarse de forma independiente y tener acceso de root, usuarios, direcciones IP, memoria, procesos, archivos, aplicaciones, bibliotecas del sistema y archivos de configuración.

Docker

Docker es un proyecto de código abierto que proporciona una manera sistemática de automatizar el despliegue más rápido de las aplicaciones de Linux dentro de contenedores. Básicamente, Docker es una ampliación LXC con una API de nivel de kernel y aplicación que juntos ejecutan procesos de forma aislada: CPU, memoria, E/S, red, etc. Docker también usa espacios de nombres para aislar completamente la vista de una aplicación del entorno operativo,

incluidos los árboles de procesos, la red, las ID de usuario y los sistemas de archivos.

Rocket (rkt)

rkt es un motor de contenedor de aplicaciones desarrollado para entornos de producción modernos nativos de la nube. Cuenta con un enfoque de pod nativo, un entorno de ejecución conectable y un área de superficie bien definida que lo hace ideal para la integración con otros sistemas.

	Docker	Rocket (rkt)	Linux Containers (LXC)	OpenVz
Pensado para ejecutar una aplicación	Si	Si	No	No
Limitación/reducción de recursos**	Si	No bien definida	Si	Si
Manejo vía acceso remoto	No	No	Si	Si
Comunidad	Alta	Baja	Baja	Baja
Facilidad de uso**	Alta	Baja	No comprobada	No comprobada

****Limitación/reducción de recursos:** técnicamente es posible pero las opciones para realizarlo son un poco confusas y no es una tarea tan fácil como en las demás opciones.

****Facilidad de uso:** Ponemos esta puntuación basado en una prueba de uso que se ha realizado. Esta prueba de uso podemos encontrarla en el Anexo III.

2.7 Orquestación de contenedores

La orquestación de contenedores trata de facilitar el manejo de los contenedores, sobre todo en entornos cluster, a lo largo de su ciclo de vida automatizando muchas tareas como, por ejemplo:

- Despliegue de contenedores
- Monitorización de los recursos de cada nodo
- Monitorización del estado de los contenedores y de los nodos
- Escalar hacia arriba o hacia abajo a lo largo de la infraestructura dependiendo de la carga de trabajo en ese momento
- Traslados de los contenedores de un nodo a otro bien porque uno de los nodos se encuentra escaso de recursos o porque un nodo se caiga
- Balanceo de carga

La orquestación de contenedores además proporciona mecanismos para proporcionar a las infraestructuras alta disponibilidad, mecanismos que no proporcionan los contenedores por sí mismos.

Aunque existen más tecnologías para la orquestación de contenedores nos centraremos en Kubernetes y Docker Swarm.

Kubernetes

Esta tecnología de orquestación de contenedores de código abierto fue desarrollada por Google en 2014, pero en la actualidad su mantenimiento y desarrollo es llevado a cabo por la “Cloud Native Computing Foundation” (Fundación de Computación Nativa en la Nube).

Este orquestador cuenta con dos tipos de nodos, nodos “master” que son los encargados de organizar la carga de trabajo, comprobar el estado de los nodos, entre otras tareas; y los nodos “worker”, que se encargan de ejecutar los contenedores.

Kubertnetes cuenta con una serie de objetos básicos que son:

- **Pod**: la unidad básica de trabajo: un pod es un conjunto de uno o más contenedores, aunque en la práctica lo más normal es que un pod esté compuesto por un solo contenedor.

- **Servicios**: se utilizan para exponer a la red externa un conjunto de pods ya que kubernetes provee a cada conjunto de pods una dirección IP propia.
- **Volúmenes**: es un sistema que tiene Kubernetes para asegurar que la información que contienen los pods perdure, aunque el pod tenga que ser reiniciado.
- **Espacios de nombres**: es un sistema para delimitar el espacio de trabajo en la memoria física en proyectos donde hay muchos usuarios y diferentes equipos. Si un proyecto es muy pequeño no es necesario utilizarlos.

Docker Swarm

Es un modo de funcionamiento que incluye Docker para que un conjunto de máquinas, que pueden ejecutar Docker, funcione en un cluster. Una gran ventaja que tiene Docker Swarm es que en el cluster seguiremos utilizando los mismos comandos que con Docker.

Al igual que en Kubernetes tenemos 2 tipos de contenedores: los nodos “swarm manager” y los nodos “workers”. Los primeros son los nodos que pueden administrar el cluster, balancear la carga, ejecutar comandos o permitir que a otros nodos unirse al cluster. Los segundos sólo aportan capacidad de cómputo, no tienen ningún privilegio.

Este orquestador es muy útil a las fases de desarrollo donde normalmente no se cuenta con el total del hardware que habrá en su puesta en producción debido a que dispone de la capacidad de crear máquinas virtuales para simular nodos que falten. Además, en comparación con Kubernetes tanto su instalación como uso es mucho más sencilla.

A continuación, nos vemos una tabla comparativa entre Kubernetes y Docker Swarm.

	Kubernetes	Docker Swarm
Instalacion	Compleja	Sencilla
Escalabilidad	5000 nodos y 300000 contenedores	No especificado
Escalado automatico de la aplicacion	Si	No
Balanceo de carga	Si	Si
Monitorizacion**	Si	No directamente
Alta disponibilidad	Si	Si
Soporte de diferentes tipos de contenedores	Si	No
Facilidad de uso e instalación**	Baja	Alta

****Monitorización:** Docker swarm no dispone de forma nativa herramientas para la monitorización, pero es posible a través de herramientas de terceros.

****Facilidad de uso e instalación:** se puede comprobar más adelante la diferencia entre uno y otro.

2.8 Conclusión

Tras la realización de este trabajo de investigación hemos aprendido mucho acerca de que son las tecnologías de virtualización ligera, cuales hay y en que se diferencian. También hemos podido llegar a la conclusión de que el área de la virtualización es mucho más extensa y compleja de lo que puede parecer en

un principio, y que hay una gran variedad de alternativas que pueden encajar mejor dependiendo de las necesidades del proyecto. También ha resultado interesante conocer la tecnología de los unikernels, ya que, aun estando en una fase temprana de su desarrollo, plantea unas soluciones muy interesantes para los dispositivos del Internet de las Cosas debido a su bajo requerimiento de recursos.

Capítulo 3

Tecnologías utilizadas

3.1 Hardware

3.1.1 Raspberry Pi 3

Es un ordenador de placa simple que la fundación Raspberry Pi diseñó principalmente con motivos de impulso académico y para el prototipado. Se ha elegido esta placa como nodos para el cluster ya que dispone de entrada de Ethernet, para la comunicación entre nodos del cluster, y módulo WIFI para la comunicación entre el cluster y los sensores.

Figura 5 - Raspberry Pi 3

3.1.2 NodeMCU ESP8266

Es una placa programable que monta el chip ESP8266 fabricada en este caso por la empresa Lolin. Esta será la que se comunique directamente con los sensores. Se ha elegido esta placa porque cuenta con un módulo WIFI y porque es posible trabajar fácilmente con ella en el entorno de Arduino IDE. Cuenta con pines de salida digital y un pin de salida analógica.

Figura 6 - Placa NodeMCU

3.1.3 Sensor DHT11

Este sensor mide la temperatura y la humedad. Funciona con una alimentación de 3.3V a 5V, con un consumo de 2.5 mA y la señal de salida es digital. A continuación, se muestra una tabla con dos especificaciones relevantes:

	Temperatura	Humedad
Rango de funcionamiento	De 0ºC a 50ºC	De 20% HR a 90% HR
Precisión	± 2ºC a 50ºC	± 5% HR entre 0ºC y 50ºC

Fundamentalmente hay 2 tipos de sensores uno con PCB y otro sin ella. La diferencia entre ambos es que el modelo con PCB lleva una resistencia de 5KΩ integrada además del número de patillas como se puede ver en la siguiente figura.

Figura 8 - sensor DHT11 con pcb

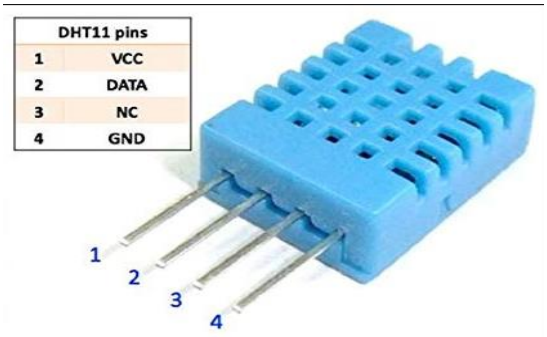


Figura 7 - sensor DHT11 sin pcb

3.1.4 Fotorresistor

El fotorresistor o LDR (Light Dependent Resistor) en inglés es una resistencia que reacciona con la luz. La salida es analógica.

Figura 9 - fotorresistencia

Los fotorresistores están compuestos por sulfuro de cadmio el cual reacciona con la luz. Dependiendo de cómo conectemos el fotorresistor podemos conseguir que a mayor luz mayor voltaje o el efecto contrario, a mayor luz menor voltaje. Las siguientes figuras muestran cómo conseguir cada uno de los efectos descritos.

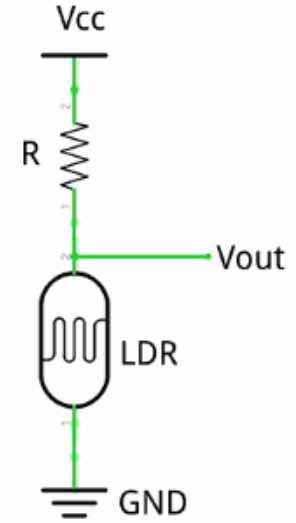


Figura 10 - Esquema1 fotorresistencia

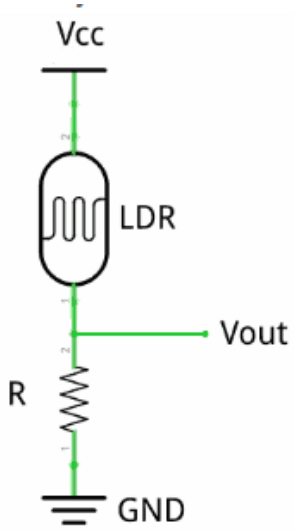


Figura 11 - Esquema 2 fotrresitencia

Para el caso de este trabajo usaremos la opción de a mayor luz mayor voltaje.

3.1.5 Sensor de movimiento PIR

Es un sensor de movimiento que detecta la presencia de una persona cuando entra en su rango de acción. Funciona tanto a 3.3V como a 5V. Su salida es digital

Figura 12 - Sensor de movimiento PIR

3.1.6 Diodo LED

Es una fuente de luz con una unió p-n. La salida es digital. Su principal ventaja es que consume muy poca energía para funcionar.

Figura 13 - Diodo LED

3.1.7 Arduino UNO

Placa programable de código abierto desarrollada por Arduino. La característica por la que la hemos escogido es porque es capaz de alimentar a los sensores con una potencia de 5 voltios.

Figura 14 - Arduino UNO

3.2 Software

3.2.1 Raspbian Strecht lite

Raspbian es el sistema operativo oficial soportado por la fundación Raspberry. Es un sistema operativo Linux basado en Debian y optimizado específicamente para las placas Raspberrys. Este sistema fue lanzado por primera vez en 2012 como un proyecto independiente y en 2015 la fundación Raspberry lo adopto como sistema operativo principal. Strecht es la versión de Debian sobre la que se basa el sistema operativo y lite indica que variante del sistema es. A la hora de descargar Raspbian nos encontramos 3 variantes, todas ellas comparten la misma versión base de Debian y mismo kernel, que si las ordenamos por tamaño nos queda de la siguiente forma:

1. Raspbian Desktop con software recomendado: esta variante incluye como añadidos una interfaz gráfica de usuario y una serie de software básico para la programación que ha considerado la fundación Raspberry. Podemos ver lo que incluye en el siguiente enlace http://downloads.raspberrypi.org/raspbian/release_notes.txt

2. Raspbian Desktop: esta variante solo incluye como añadido la interfaz gráfica de usuario.
3. Raspbian Lite: esta variante no tiene ningún añadido, es raspbian el cual se maneja desde una terminal como podría hacerse en un Ubuntu server

Para este trabajo hemos escogido la variante Lite puesto que es la que consume menos recursos y posteriormente añadiremos el software que se considere necesario.

3.3 Tecnologías y protocolos

3.3.1 Contenedores

Esta tecnología, de la que hemos hablado en el estudio del capítulo anterior, la hemos seleccionado, frente a la tecnología de máquinas virtuales y unikernels, porque necesita menos recursos para su funcionamiento además de tener unos tiempos de arranque mucho menores en comparación con una máquina virtual, y porque tiene la tecnología tiene la suficiente madurez para ser utilizada en comparación con los unikernels.

3.3.2 Docker

Esta tecnología, descrita en el capítulo anterior, la hemos seleccionado porque, tal y como hemos visto en el estudio previo, es la tecnología que ofrece una mayor facilidad de uso además de ser una de las opciones con mayor comunidad a sus espaldas. Será la tecnología que utilicemos para virtualizar los microservicios.

3.3.3 Docker Swarm

Es uno de los orquestadores de contenedores que hemos hablado anteriormente. En base al estudio anterior, hemos elegido esta por la facilidad tanto de instalación como de uso, y por una dificultad que impedía el uso de Kubernetes en este trabajo, de la cual hablamos más en profundidad en el capítulo 5 en la sección "*Problemas encontrados"*. Este orquestador lo utilizaremos para que ambos nodos puedan compartir potencia de cálculo y para que los contenedores se distribuyan por el cluster en función de la carga de trabajo de cada nodo.

3.3.4 MQTT

MQTT (de las siglas en inglés *Message Queuing Telemetry Transport*), es un protocolo de paso de mensajes mediante publicación/subscripción destinado principalmente para la comunicación de sensores en el internet de las cosas. Funciona sobre el protocolo TCP/IP y está pensado para que los mensajes sean los más pequeños posibles ya que en muchas ocasiones los sensores pueden estar localizados en zonas remotas donde el ancho de banda es muy limitado.

En este protocolo existen 2 figuras:

1. El cliente, el cual se puede subscribir a uno o más temas por lo que recibirá los mensajes publicados en esos temas y publicar en uno o mas temas para de esta forma enviar mensajes por esos temas en concreto.
2. El broker, solo hay uno y es el encargado de recibir todos los mensajes publicados. Cuando recibe un mensaje lee cual es el tema y lo reenvía a todos los clientes que estén subscritos a dicho tema.

En la siguiente figura se observa de forma esquemática lo anteriormente explicado:

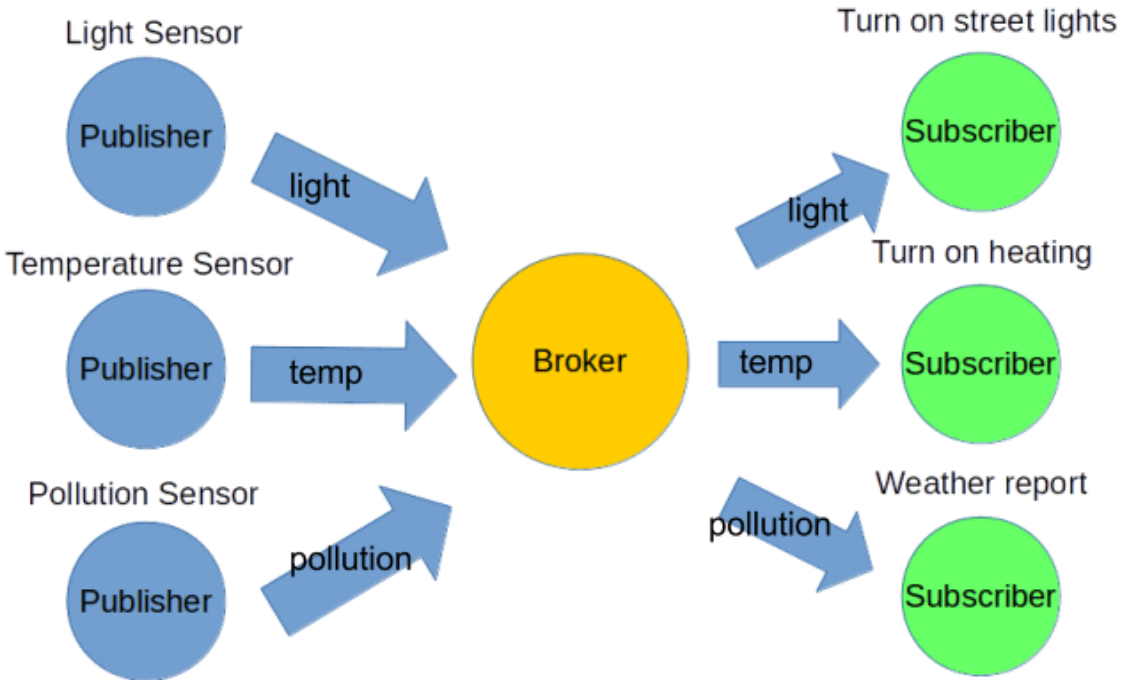


Figura 15 - Esquema funcionamiento MQTT

Este protocolo además ofrece 3 niveles de calidad de servicio diferentes:

- Calidad 0: el mensaje se envía una vez, pero el broker no establece ningún mecanismo para asegurar que el mensaje llegue al receptor
- Calidad 1: el broker establece mecanismos para que el receptor reciba el mensaje al menos una vez, pero no establece ningún mecanismo para asegurarse que el mismo mensaje llegue más de una vez
- Calidad 2: el broker establece mecanismos para que el mensaje llegue al receptor y sin repeticiones.

A mayor calidad de servicio más lentas son las comunicaciones por lo que si se quiere establecer una comunicación a tiempo real no sería recomendable escoger la calidad de servicio 2.

3.3.5 Microservicios

Los microservicios son un estilo de arquitectura y una forma de programar. El objetivo de los microservicios es que las aplicaciones sean modulares a diferencia del estilo de programación clásica en la que hay una aplicación grande que lo hace todo. Cuando se crea un microservicio la intención es que ese microservicio realice una y sola una tarea sencilla, y la composición de diferentes microservicios darán lugar a una aplicación completa. La separación de una aplicación en módulos más pequeños tiene una serie de ventajas:

- **Mantenimiento**: es más simple puesto que las secciones de código de un microservicio son más pequeñas y la depuración del código es mucho más rápida.
- **Escalabilidad**: si se quiere añadir funcionalidades a una aplicación o hacera compatible más sistemas simplemente hay que añadir microservicios.
- **Distribuida**: podemos hacer que la aplicación se ejecute en diferentes equipos localizados en diferentes lugares ya que normalmente no es necesario que los microservicios tengan que estar ejecutándose en la misma máquina.
- **Testeo**: los microservicios pueden probarse de forma individual permitiendo que las pruebas necesarias para comprobar el buen funcionamiento de este sean más sencillas

- **Reutilización**: si vemos los microservicios como una pequeña función nos damos cuenta de que los que se hayan realizado para una aplicación pueden ser válidos para otra, y su reutilización puede ser directa sin necesidad de tener que cambiar nada de código.

3.3.6 Eclipse Mosqitto

Es un broker (intermediario) de código abierto que implementa el protocolo MQTT en sus versiones 5.0, 3.1.1 y 3.0. Además, este bróker necesita pocos recursos para su funcionamiento, haciéndolo perfecto para su despliegue y ejecución en una Raspberry. Este software lo usaremos para la comunicación de los elementos del sistema a través de MQTT.

3.3.7 Rufus

Es un sencillo programa creado para el sistema Windows para crear unidades USB de arranque con imágenes ISO. Se ha elegido este programa porque no necesita ningún tipo de instalación para poder funcionar y por su sencillez a la hora de usarlo. Lo usaremos para grabar las imágenes del sistema operativo Raspbian en tarjetas microSD.

3.4 Entornos de desarrollo

3.4.1 Microsoft Visual Studio Code

Es un editor de código desarrollado por Microsoft que soporta diferentes lenguajes de programación. También se puede depurar el código escrito y cuenta con integración con la herramienta de control de versiones Git. Mediante la instalación de diferentes extensiones, que pueden descargarse desde el propio entorno, podemos obtener diferentes mejoras, como la extensión “Import Cost” que nos marca al lado de cada paquete que importemos cuanta memoria de disco consume.

3.4.2 Arduino IDE

Es un entorno de desarrollo de código abierto desarrollado por Arduino para las plataformas de Windows, Mac y Linux. Tiene soporte para todas las placas de Arduino, pero con ayuda de librerías de terceras partes es posible de programar placas de otros fabricantes. Utiliza el lenguaje de C++

3.5 Lenguajes de programación

3.5.1 C++

Es un lenguaje de programación de alto nivel creado en 1985 como una evolución del lenguaje de programación C. Tiene herramientas para la programación orientada a objetos, pero también ofrece la posibilidad de programar a bajo nivel. Por esta razón hemos escogido este lenguaje de programación para la programación de las placas NodeMCU.

3.5.2 Python

Este lenguaje de alto nivel fue creado en 1991 con el propósito de que cuando se programase el código resultante fuese lo más legible posible. Por ello una de las principales características es la facilidad de aprendizaje de este, además de ser uno de los lenguajes de programación más de moda en estos días contando con una comunidad muy grande a sus espaldas. Así que hemos elegido este lenguaje de programación para la programación de los microservicios y de la página web.

Capítulo 4

Diseño e implementación

4.1 Diagramas

4.1.1 Diagrama caso de uso

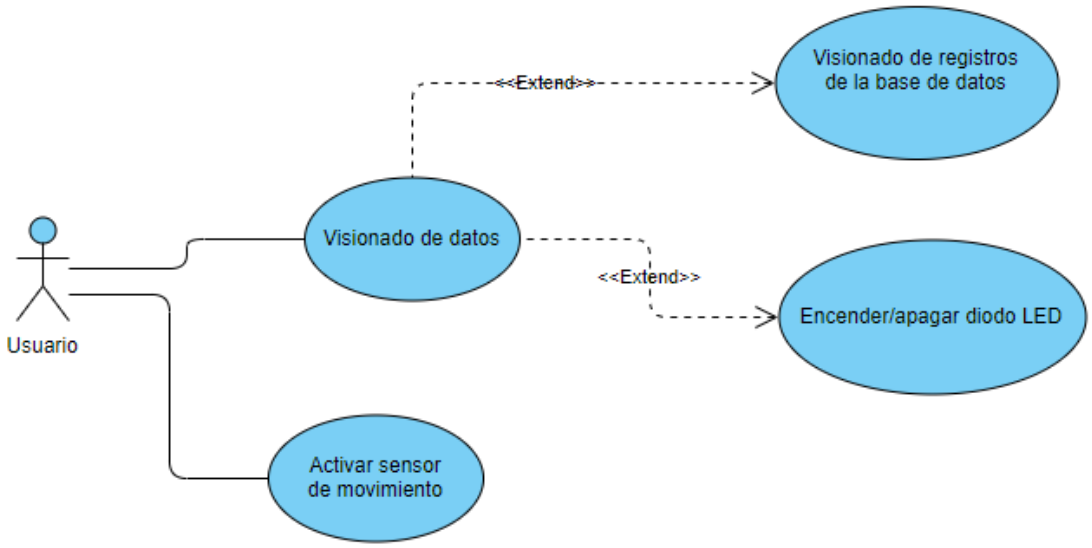


Figura 16 - Caso de uso

El actor es el usuario que podrá o bien activar el sensor de movimiento entrando en su rango de acción o accediendo a la página web para el visionado de datos. Desde el visionado de datos, además puede acceder al registro de mediciones guardadas en la base de datos o encender y apagar el diodo LED.

4.1.2 Diagramas de secuencia

En el siguiente diagrama podemos ver la secuencia para acceder a la consulta de datos.

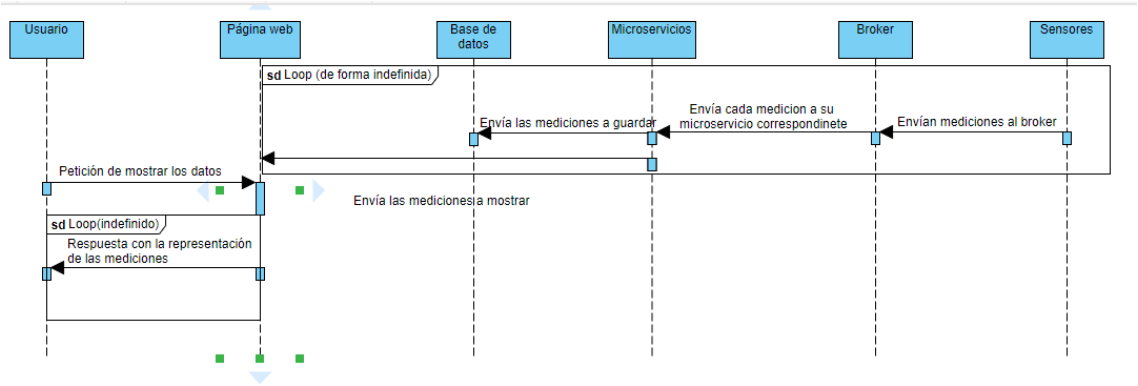


Figura 17 - Diagrama de secuencia 1: Consulta de Datos.

Para acceder a los registros guardados en la base de datos, además de realizar lo descrito en la figura anterior, hay que añadir la siguiente secuencia de pasos en los diferentes componentes mostrados:

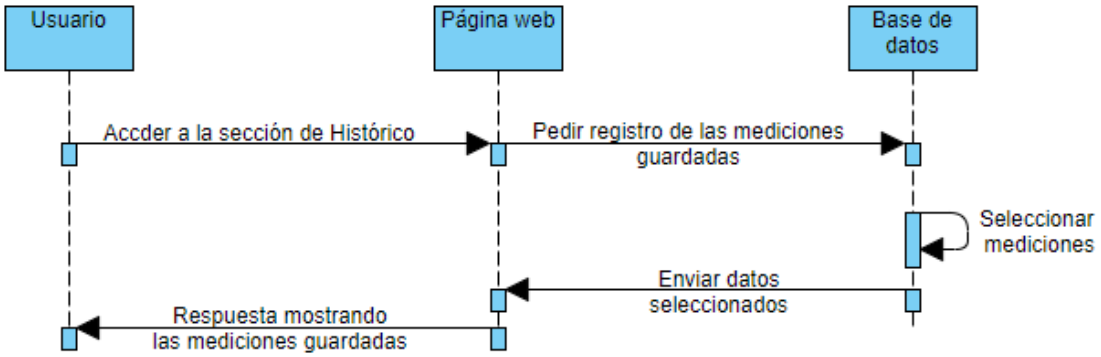


Figura 18 - Diagrama de secuencia 2: Acceso a Base de Datos.

Secuencia de pasos para encender o apagar el diodo LED, previamente hay que realizar los pasos descritos en la figura 17.

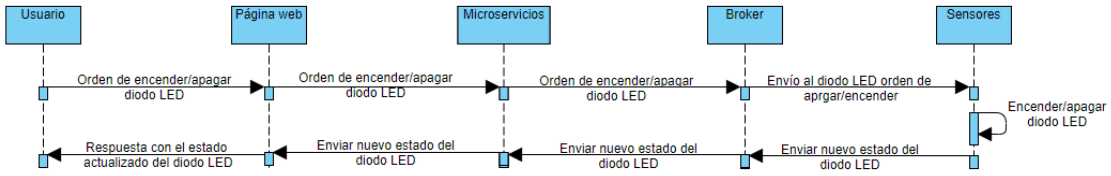


Figura 19 - Diagrama de secuencia 3: Actuación sobre diodo LED.

Secuencia de pasos para activar el sensor de movimiento PIR

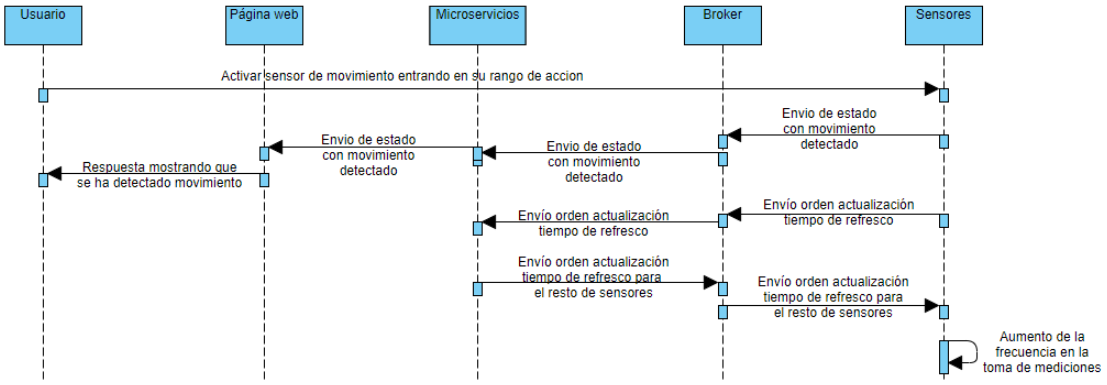


Figura 20 - Diagrama de secuencia 3: Activación Sensor PIR

4.2 Conexiones físicas

En este apartado vamos a mostrar como son las conexiones físicas de los sensores a las placas NodeMCU. Como disponemos de 2 placas NodeMCU las utilizaremos de la siguiente forma:

- La primera para conectar el sensor de temperatura/humedad, la fotorresistencia y el diodo led que emulara a una bombilla.
- La segunda placa conectaremos el sensor de movimiento y un led para darnos una indicación de cuando detecta movimiento de una forma sencilla.

Hemos elegido esta distribución de los sensores porque el sensor de movimiento hará de actuador y el resto de los sensores tomarán medidas de forma pasiva, siendo necesaria una comunicación entre los sensores.

Ahora pasaremos a ver de forma detallada las conexiones de cada una de las placas.

4.2.1 Conexión del sensor de Humedad/Temperatura. Placa 1

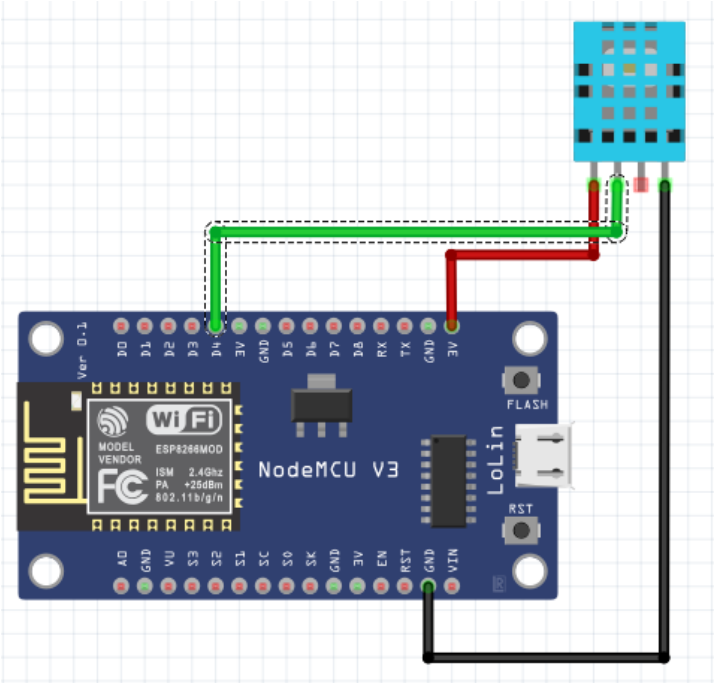


Figura 21 - conexión sensor DHT11 a NodeMCU

4.2.2 Conexión fotorresistencia. Placa 1

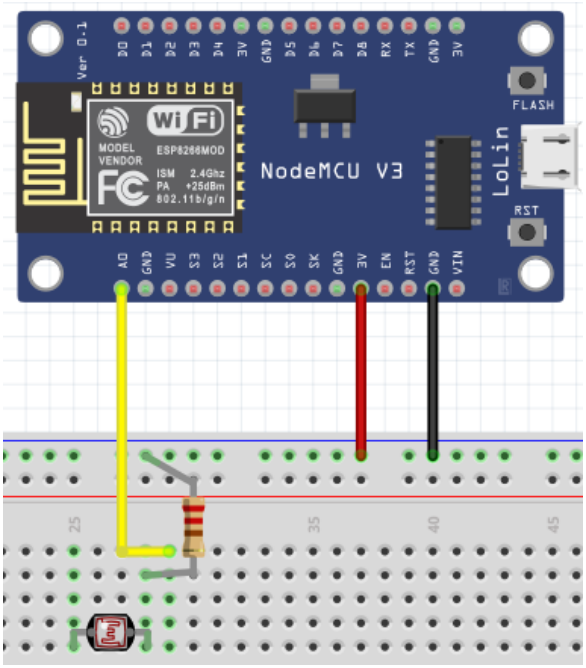


Figura 22 - Conexión fotorresistencia a NodeMCU

4.2.3 Conexión led. Placa 1

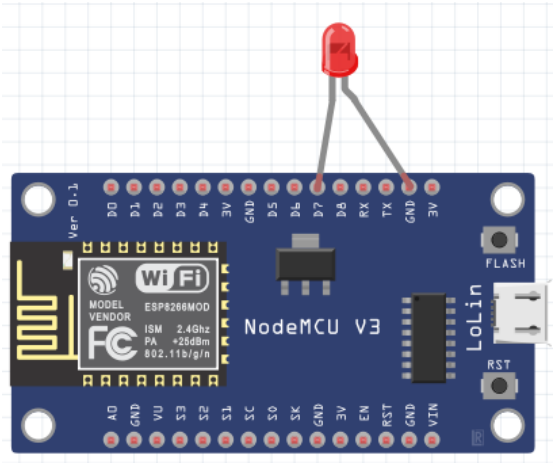


Figura 23 - Conexión diodo LED a NodeMCU

4.2.3 Conexión placa completa

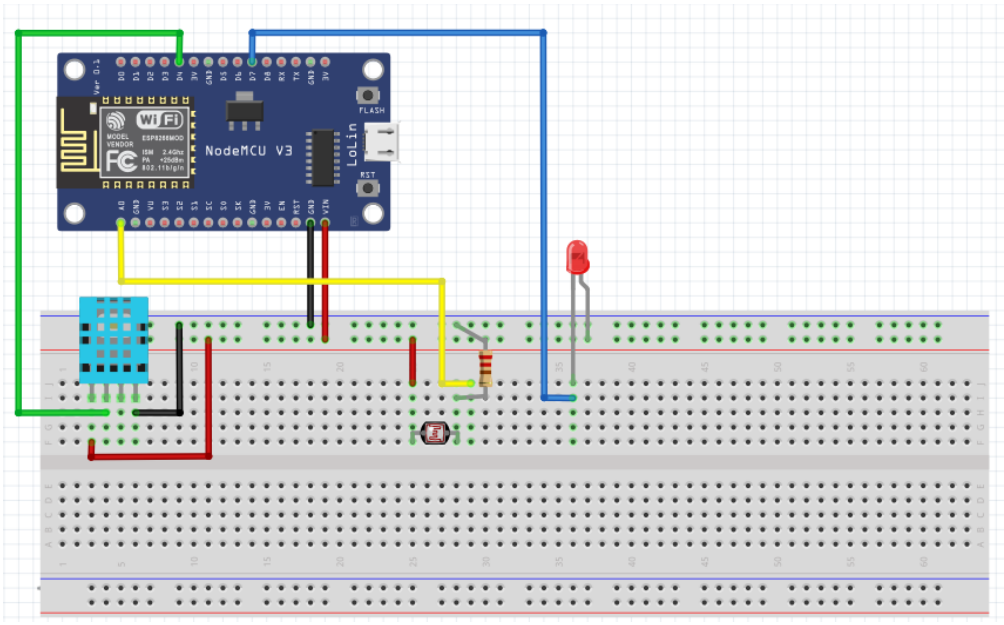


Figura 24 - Conexión completa de placa NodeMCU 1

4.2.4 Conexión sensor movimiento y led. Placa 2

En este apartado mostraremos directamente la conexión completa porque solo se conecta un sensor. En este caso la placa de Arduino la utilizamos como fuente de alimentación, podría ser sustituida por cualquier otra fuente que proporcionase 5V

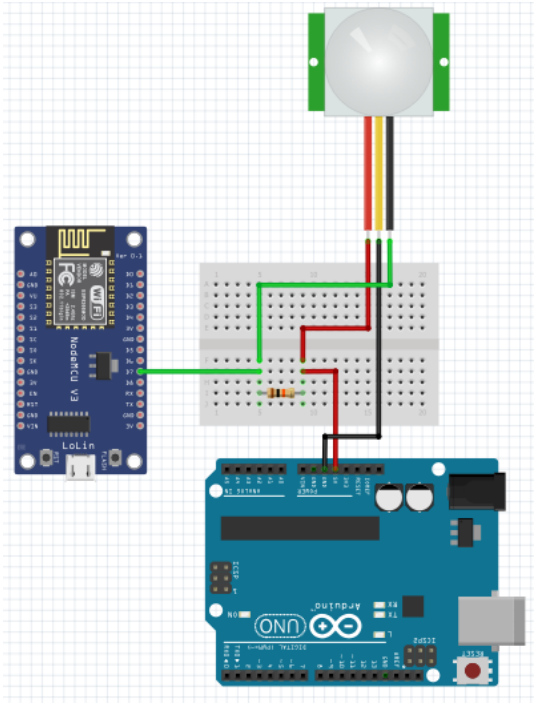


Figura 25 - Conexión completa placa NodeMCU 2

4.3 Conexiones lógicas

En esta sección veremos cómo están organizadas las conexiones con el broker de MQTT. A continuación, vemos un esquema con los temas a los que están subscritos y cuál es el tema por el que publican información cada elemento del sistema.

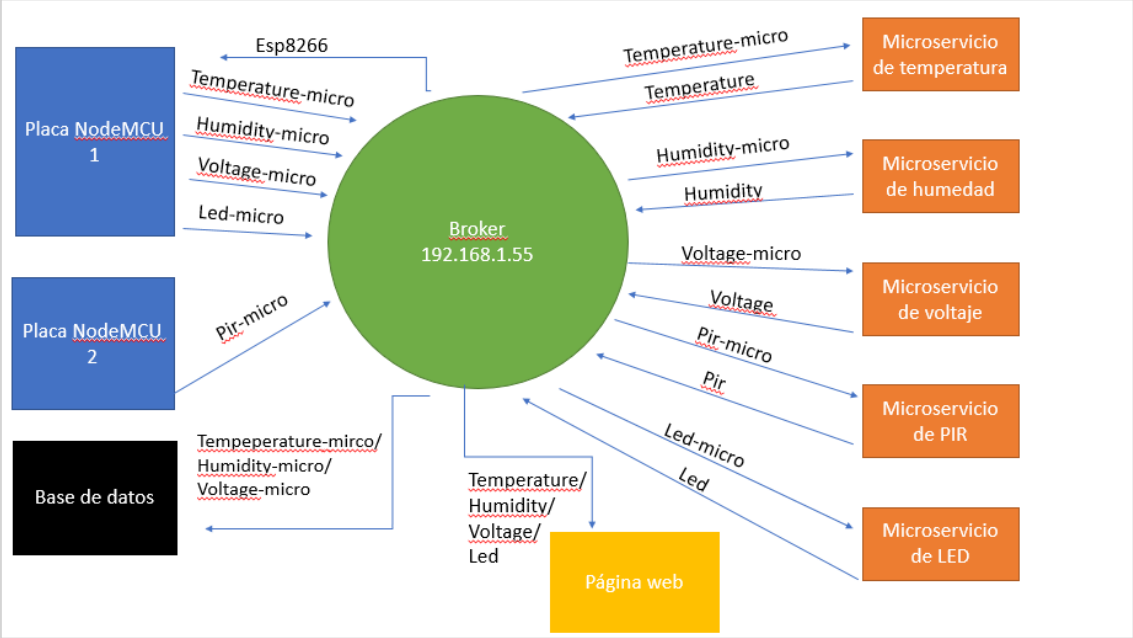


Figura 26 - Esquema conexión lógica

4.3.1 Placa NodeMCU 1

Como podemos observar en la figura anterior la placa NodeMCU 1 publica por los temas “Temperature-micro”, “Humidity-micro”, “Voltage-mciro” y “Led-micro”. Por cada uno de estos temas publica la medición tomada por cada uno de los sensores que tiene conectados excepto por el tema de “Led-micro” que por este tema publica si el diodo LED se encuentra encendido o apagado. Además, recibe mensajes por el tema “Esp8266” para actualizar el estado del diodo LED y la frecuencia de muestreo.

4.3.2 Placa NodeMCU 2

Esta placa al solo tener el sensor de movimiento publica por un solo tema llamado “Pir-micro”.

4.3.3 Microservicios

Los microservicios los tratamos de forma conjunta ya que todos se comportan de la misma manera. Todos reciben la información por el tema acabado en “-

micro” y la reenvían por el mismo tema que no tiene la terminación anterior. Por ejemplo, el microservicio encargado de las mediciones de la temperatura recibe la información a través del tema “Temperature-micro” y la reenvía por el tema “Temperature”.

4.3.4 Base de datos

La base de datos solo recibe información de los sensores. Reenviará la información que a través otra manera que ya veremos más adelante.

4.3.5 Página web

esta recibe la información de los microservicios a través de los temas “Temperature”, “Humidity”, “Voltage” y la información del estado del led por el tema “Led”

4.4 Programación placas NodeMCU

Una vez que conocemos cuales son las conexiones físicas y lógicas que va a tener nuestro sistema vamos a comenzar a programar las placas NodeMCU. Usaremos el entorno de Arduino IDE el cual hay que realizar unos pasos previos antes de comenzar a programar debido a que las placas que estamos usando no son de Arduino sino de otro fabricante.

4.4.1 Preparación del entorno de Arduino IDE

Una vez que abrimos el entorno nos vamos a la pestaña de “Archivo” y ahí buscamos una pestaña llamada “Preferencias”. En la ventana que se abre vamos hasta la opción que pone “Gestor de URLs Adicionales de Tarjetas" y en el cuadro de texto ponemos la siguiente dirección:

http://arduino.esp8266.com/stable/package_esp8266com_index.json

Una vez añadida la dirección cerramos la ventana y vamos a la pestaña de “Herramientas”, ahí buscamos otra que tenga escrito “Placas” y en el desplegable que se abre buscamos la pestaña de “Gestor de Tarjetas…”. En la ventana que se abre escribimos en el filtro de búsqueda **esp8266** e instalamos la opción que vemos en la siguiente figura:

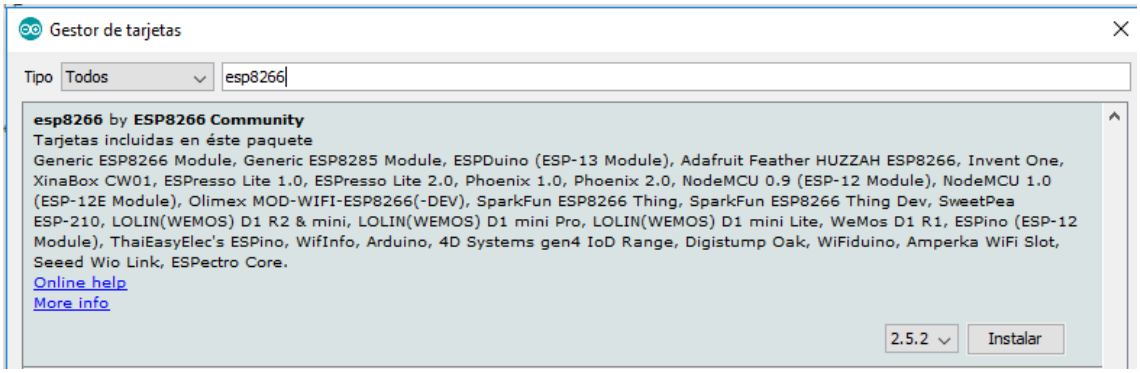


Figura 27 - Librería ESP8266 Arduino IDE

Con esto el entorno de Arduino ya está preparado para poder programar nuestras placas.

4.4.2 Programación placa NodeMCU 1

En un breve resumen lo que vamos a programar la placa para que sea capaz de realizar lo siguiente:

1. Leer los datos del sensor de humedad/temperatura, fotorresistencia además de captar el estado del diodo LED.
2. Conectarse vía wifi para enviar los datos al broker.
3. Recibir información del broker con la que cambiar la frecuencia a la que se toman los datos además de poder encender o apagar el diodo LED.

A continuación, analizaremos el código parte por parte:

```
#include <ESP8266WiFi.h>
#include <PubSubClient.h>
#include <DHT.h>

#define DHTPIN 2
#define DHTTYPE DHT11

int ledPin = 13;
int ledState = LOW;

int del = 5000;

DHT dht(DHTPIN, DHTTYPE);

const char* ssid = "MOVISTAR_7728";
const char* password =  "contraseña-wifi";
const char* mqttServer = "192.168.1.55";
const int mqttPort = 1883;
const char* mqttUser = "mqttusr";
const char* mqttPassword = "mqttrpi";

WiFiClient espClient;
PubSubClient client(espClient);

```

Figura 28 - Código NodeMCU 1

En esta parte vemos las librerías importadas, explicadas ya en apartados anteriores. Definimos que en que pin está conectado el sensor de temperatura/humedad y cual es el tipo del sensor. En la definición del pin del sensor hemos puesto el 2, que es el equivalente al pin D4 que vemos en el sensor. Dejamos un anexo al final de este trabajo con una figura que muestra las equivalencias de pines de esta placa.

Más adelante definimos el pin donde está el diodo LED y su estado inicial que será apagado. La variable “del” se usará para establecer la frecuencia de muestreo más adelante, por defecto está a 5 segundos.

En la línea 13 vemos la inicialización del sensor de temperatura/humedad pasándole como argumentos el pin donde está conectado y el tipo del sensor.

En el siguiente grupo de constantes las 2 primeras son usadas para conectar la placa al wifi la cuales son el nombre de la red wifi y su contraseña. El resto de las constantes son parámetros para establecer la conexión con el broker.

Por último, vemos 2 variables para la creación de la conexión con el broker.

```
25  void setup() {
26    Serial.begin(115200);
27    pinMode(ledPin, OUTPUT);
28    digitalWrite(ledPin, ledState);
29    WiFi.begin(ssid, password);
30
31    while (WiFi.status() != WL_CONNECTED) {
32      delay(500);
33      Serial.println("Connecting to WiFi..");
34    }
35
36    Serial.println("Connected to the WiFi network");
37    client.setServer(mqttServer, mqttPort);
38    client.setCallback(callback);
39
40    while (!client.connected()) {
41      Serial.println("Connecting to MQTT...");
42      if (client.connect("ESP8266Client", mqttUser, mqttPassword )) {
43        Serial.println("connected");
44        client.subscribe("esp8266");
45      } else {
46        Serial.print("failed with state ");
47        Serial.print(client.state());
48        delay(2000);
49      }
50    }
51  }
52
```

Figura 29 - Código NodeMCU 2

En la función setup inicializamos la salida por pantalla en la frecuencia 115200, este paso es útil para más adelante poder depurar el código en el caso de que hubiese problemas. Establecemos el pin donde está el diodo LED como pin de salida y escribimos su estado inicial. Seguidamente nos encontramos una llamada a la librería del wifi para iniciar la conexión wifi y con el primer bucle while esperamos a que se establezca la conexión.

El siguiente bucle while es para esperar hasta que se consiga establecer conexión con el broker. En la línea 42 en la función connect vemos que pasamos como argumento el usuario y contraseña para conectarse al broker y una cadena de caracteres. Esta cadena es el ID del cliente, si no le pasásemos esta cadena el protocolo MQTT automáticamente le daría uno, pero en este caso es obligatorio debido a que la librería no cuenta con un método connect sin este argumento.

En la línea 46 observamos que hemos conectado al cliente a un tema llamado “esp8266”.

```
65  void callback(char* topic, byte* payload, unsigned int length) {
66
67    for (int i = 0; i < length; i++) {
68      Serial.print((char)payload[i]);
69    }
70
71    if(length==1){
72      int x = (int)payload[0];
73      changeFreqOrLed(x);
74    }
75
76  }
```

Figura 30 - Código NodeMCU 3

La función *callback* será la que se ejecute cada vez que llegue un mensaje por el tema al que nos hemos subscrito. Si el mensaje es de longitud1 transformará a un número entero el código recibido y llamará a la función que hará el cambio pertinente según el código recibido.

```
void changeFreqOrLed(int state){
  if(state==48){ //48 en bytes equivale a recibir un 0
    ledState=LOW;
    digitalWrite(ledPin, ledState);
    char led [1]={'0'};
    client.publish("led-micro",led);
  }else if(state==49){ //49 en bytes equivale a recibir un 1
    ledState=HIGH;
    digitalWrite(ledPin, ledState);
    char led [1]={'1'};
    client.publish("led-micro",led);
  }else if(state==50){
    //Se detecta movimiento por tanto se aumenta la frecuencia de muestreo
    del=2000;
    Serial.println("Change Fregquncy Hight");
  }else if(state==51){
    //Se detecta ausencia de movimiento por tanto se baja la frecuencia de muestreo
    del=7000;
    Serial.println("Change Fregquncy Low");
  }else{
    Serial.print("***ERROR state do not recognize ");
    Serial.println(state);
  }
  Serial.println("----------------------");
}

```

Figura 31 - Código NodeMCU 4

Esta es la función que cambia la frecuencia del muestreo o el estado del diodo LED dependiendo de cuál es el código que haya recibido.

```
131  void loop() {
132      char volt [10];
133      char temp [10];
134      char humd [10];
135
136      sprintf(volt,"%g",readVoltage());
137      sprintf(temp,"%g",readDHTTemperature());
138      sprintf(humd,"%g",readDHTHumidity());
139
140      Serial.println("-----------------------");
141
142      client.publish("voltage-micro", volt);
143      client.publish("temperature-micro", temp);
144      client.publish("humidity-micro",humd);
145
146      delay(del);
147      client.loop();
148  }
```

Figura 32 - Código NodeMCU 5

La función *loop* lee los datos de los sensores con las funciones respectivas. Estas funciones no son de ninguna librería, leen el dato comprueban si el es un número y si no lo es devuelven un -999. La función publish tiene 2 argumentos, el tema por el que se publica y el mensaje en cuestión. Esta función obliga a que el mensaje sea un array de caracteres por eso se transforma el dato que se obtiene de la función de lectura.

La función client.loop es una función que tiene que ser llamada regularmente según la documentación de la librería. Es la que permite al cliente procesar los mensajes de entrada y mantener la conexión con el broker

4.4.3 Programación placa NodeMCU 2

```
47  void loop() {
48    int state = digitalRead(sensor);
49    if (state == LOW) {
50      Serial.println("Motion detected!");
51      digitalWrite(LED_BUILTIN, LOW);
52      char msg [1]={'2'};
53      client.publish("pir-micro", msg);
54      delay(10000);
55    }else {
56      Serial.println("No motion detected.");
57      digitalWrite(LED_BUILTIN, HIGH);
58      char msg [3] ={'3'};
59      client.publish("pir-micro", msg);
60      delay(10000);
61    }
62    Serial.println("------------------");
63  }
```

Figura 33 - Código NodeMCU placa 2

Lo que hace la placa es revisar cuando el sensor detecta movimiento, si lo detecta envía un 2, que es el código para indicar movimiento, y se espera 10 segundos. En caso contrario enviará un 3, código para indicar que hay movimiento, y se esperará 10 segundos. Además, utilizamos un led que se enciende cuando detecta movimiento para ayudarnos a saber el estado del sensor sin necesidad de tener que abrir una terminal en el ordenador.

4.5 Preparación Microservicios

Ahora veremos la implementación realizada para los microservicios mostrando las partes que son más interesantes.

Para los microservicios de la temperatura, humedad y voltaje como son muy parecidos entre si los comentaremos de forma conjunta explicando solo uno de ellos.

4.5.1 Microservicios humedad, temperatura y voltaje

```python
import paho.mqtt.client as mqtt

mqtt_username = "mqttusr"
mqtt_password = "mqttrpi"
mqtt_topic_sub = "humidity-micro"
mqtt_topic_pub = "humidity"
mqtt_broker_ip = "192.168.1.55"


def on_connect(client, userdata, flag, rc):
    print "Connected!", str(rc)
    client.subscribe(mqtt_topic_sub)

def on_message(client, userdata, msg):
    print "Topic: ", msg.topic + "\nMessage: " + str(msg.payload)
    client.publish(mqtt_topic_pub, str(msg.payload))

client = mqtt.Client()
client.username_pw_set(mqtt_username, mqtt_password)

client.on_connect = on_connect
client.on_message = on_message

client.connect(mqtt_broker_ip, 1883)

client.loop_forever()
client.disconnect()

```

Figura 34 - Código microservicio humedad

Lo primero que observamos es que se importa la librería de MQTT y unas variables para la conexión con el broker. Si nos fijamos podemos ver que este microservicio está subscrito al tema llamado “humidity-micto” y publica por otro llamado “humidity”.

Luego encontramos las funciones on_connect y on_message. Estas funciones modelan el comportamiento del microservicio cuando se conecta al broker y cuando le llega un mensaje. En la función on_connect vemos que al conectarse se subscribe al tema “humidity-micro” y en la función on_message reenvía el mensaje recibido por el tema “humidity”

4.6 Base de datos

Hemos creado una base de datos en SQLite con una tabla que contiene la siguiente información:

Nombre columna	tipo
id	Integer
TopicName	Text
Value	Real
Date	Text
Hour	Text

El id es la clave primaria además de ser un valor que se incrementa automáticamente. El resto de los valores pueden ser nulos, pero nunca van a serlos puesto que el único método que inserta en esta tabla siempre inserta filas con todos los datos.

Esta tabla se ha creado usando el lenguaje de programación Python que en su imagen base trae ya incluida unas librerías básicas pero que para este trabajo son más que suficientes para crear y manejar tablas en SQLite.

En esta tabla guardaremos la media de las mediciones tomadas por los sensores a cada hora del día.

Ahora analizaremos el microservicio encargado de almacenar los objetos en esta tabla.

```python
import paho.mqtt.client as mqtt
from bbddhandler import Data_Handler as bd
import datetime

mqtt_username = "mqttusr"
mqtt_password = "mqttrpi"
mqtt_topic_sub = ["temperature-micro","humidity-micro","voltage-micro"]
mqtt_broker_ip = "192.168.1.55"

stack_values = [0,0,0]
stack_number = [0,0,0]
hours = [0,0,0]
```

Figura 35 - Código base de datos 1

En esta parte vemos, de nuevo las variables para la conexión con el broker, tres de listas. Estas son:

- **stack_values**: guarda la suma de las mediciones que va recibiendo
- **stack_number:** el número de mediciones que ha recibido
- **hours:** la hora a la que se han recibido estas mediciones. Este valor es el mismo en todas las posiciones, pero explicaremos el motivo más adelante.

El tamaño de las listas es igual al número de temas de los que se quiere guardar sus valores y en este caso como estamos subscritos a 3 temas nuestras listas son de tamaño 3. Si quisiéramos añadir o quitar mediciones a guardar solo habría que modificar el tamaño de estas listas y los temas a los que estamos subscritos.

```
15  def on_connect(client, userdata, flag, rc):
16      global hours
17      print "Connected!", str(rc)
18      hour = datetime.datetime.today().hour
19      i=0
20      for x in mqtt_topic_sub:
21          client.subscribe(x)
22          hours[i] = hour
23          i+=1
24
25
26  def on_message(client, userdata, msg):
27      print "Topic: ", msg.topic + "\nMessage: " + str(msg.payload)
28      i = 0
29      match = False
30      while match == False:
31          if msg.topic == mqtt_topic_sub[i]:
32              match = True
33          else:
34              i += 1
35      save_data(float(msg.payload),i)
36
```

Figura 36 - Código base de datos 2

En la función *on_connect* nos subscribimos a todos los temas e inicializamos la lista de horas con la hora actual ya que será a partir de este momento cuando empiece la base de datos a recibir mediciones.

En la función *on_message* cada vez que llega el mensaje buscamos de que tema se trata. Una vez encontrado, se invoca la función *save_data,* pasándole como argumento el mensaje recibido en formato real y la posición de las listas en la que hay que insertar el dato.

```
38  def save_data(data,pos):
39      global stack_values
40      global stack_number
41      global hours
42      if data > -999.0:
43          print stack_values
44          stack_values[pos] = stack_values[pos] + data
45          stack_number[pos] += 1
46         # store_in_bbdd(pos)  #THIS LINE IS ONLY FOR TESTING
47      if hours[pos]<datetime.datetime.today().hour:
48          store_in_bbdd(pos)
49
50
```

Figura 37 - Código base de datos 3

En esta función comprobamos que el dato recibido sea una medición correcta. Si es así actualizaremos los datos de las listas y comprobamos si la hora que tenemos guardada en la lista de horas es menor de la hora del sistema llamaremos a la función *store_in_bbd* que hará la media de las mediciones con las variables *stack_values* y *stack_number* y actualizará la hora.

Es en este punto donde podemos ver porque es necesario tener una lista de horas que tiene la misma hora en todas sus posiciones. Cada microservicio envía las mediciones de forma independiente, sin tener en cuenta si hay más microservicios enviando información. Pero esta información se procesa una por una a la hora de guardarla en la base de datos. De esta forma tenemos varios microservicios enviando información de forma concurrente, y un servicio para la base de datos que los procesa de forma lineal. Si tuviésemos solo una variable que controlara la hora en lugar de la lista, el primero de los datos que se guardara actualizaría la variable con la hora, y el resto de los datos que sí queremos guardar no se guardarían, porque verían que la hora esta actualizada a la más reciente.

4.7 Preparación de la web de consulta

La web contará con dos páginas:

1. La página “index” donde se mostrarán los datos recogidos por los sensores, además del manejo del estado del diodo LED (que se utiliza a modo de actuador).
2. La página “records” donde se mostrarán los datos almacenados en la base de datos.

Figura 38 - Captura de la página "index"

Como se puede observar de la figura anterior los datos están organizados por columnas donde irán apareciendo a medida que se envíen las mediciones. En la columna de la derecha nos encontramos con el estado del diodo LED, un botón para cambiar ese estado, información de si se ha detectado o no movimiento y un botón para ir a la página de “records”.

Figura 39 - Captura página "records"

En esta página nos encontramos solo los datos que se almacenan en la base de datos, que en este caso son la temperatura, humedad y luminosidad. En la parte inferior un botón que nos lleva de nuevo a la página de “index”.

Ambas páginas se adaptan de forma automática al tamaño de la pantalla desde donde se visualizan los datos colocándose cada columna una encima de la otra, por si entramos a la página desde un teléfono o algún otro dispositivo con una pantalla pequeña.

4.8 Preparación Raspberrys

En esta sección nos limitaremos a comentar los datos más relevantes, en el anexo IV se encuentra toda la información detallada para la preparación de las raspberrys.

Una vez que se haya instalado Raspbian lo primero que haremos es conectar las raspberrys a un monitor para activar la conexión por ssh y establecer una dirección IP estática para poder trabajar más cómodamente de forma remota.

El siguiente paso será instalar el software que vamos a necesitar. Necesitaremos instalar Docker y Mosquitto-mqtt, de este último instalaremos el broker y un cliente para poder hacer pruebas desde la terminal. Todo el software los instalaremos en las últimas versiones disponibles en el momento,18.09.6 para Docker y 1.6.3 para Mosquitto-mqtt.

Cuando hayamos instalado el software iniciaremos el cluster de Docker Swarm el cual ya viene preinstalado con Docker en una de las raspberrys. Y, por último, añadiríamos al cluster la raspberry restante.

Este proceso de instalación lo hemos realizado de forma manual, aunque podría automatizarse recopilando los comandos del apartado 1 del anexo en un script y los comandos del apartado 2 en otro.

4.9 Virtualización de microservicios y web

Una vez que ya hemos instalado, configurado y desplegado todo lo que necesitamos, e inicializado el cluster, vamos a virtualizar los servicios en contenedores Dokcer.

Como la creación de estos contenedores para todos los microservicios y la página web son muy parecidos los analizaremos todos juntos explicando un ejemplo.

Primero copiamos a una carpeta diferente todo el código y archivos que vamos a querer tener en nuestro contenedor y creamos un archivo de texto que se llame “Dockerfile”. Es importante que el archivo de texto tenga ese nombre porque en caso contrario Docker no reconocerá el archivo. Vamos a ver el ejemplo de virtualización de la página web, que tiene el siguiente formato:

```
FROM python:2.7
WORKDIR /web
COPY . /web
RUN pip install flask
RUN pip install flask-mqtt
RUN pip install flask-socketio
RUN pip install -U eventlet
CMD ["python", "flaskmqtt.py"]
```

Figura 40 - Contenido de un DockerFile

La línea con la cláusula “FROM” nos indica cual es la imagen base de la que va a partir la nueva imagen que estamos creando. Por tanto, nuestra imagen va a partir de otra imagen que ya contiene Python en su versión 2.7. La cláusula “WORKDIR” es para indicar cual va a ser nuestro espacio de trabajo dentro del contenedor. Con la tercera línea estamos indicando que se copien todos los archivos que están en el directorio actual, la carpeta en la que hemos copiado todos los ficheros que queremos en el contenedor, a directorio de trabajo que acabamos de crear. Las líneas con las cláusulas RUN instalarán las librerías necesarias para nuestra aplicación web. Y por último la cláusula “CMD” ejecutará el comando “python flask-mqtt.py” cuando arranque el contenedor.

Una vez ya tenemos el fichero tenemos que subir la imagen al Docker Hub, que es el repositorio que tiene Docker para sus imágenes. Tenemos que subir las imágenes porque los nodos en un principio no tendrán las imágenes que van a ejecutar en local y al no encontrarlas automáticamente irán a Docker Hub a buscar la imagen. Para ello ejecutaremos los siguientes comandos:

$ dokcer login

$ dokcer tag (nombre imagen) (usuario Docker Hub)/(nombre imagen)

$docker push (usuario DockerHub)/(nombre imagen)

El comando “login” es para acceder a Docker Hub con nuestra cuenta. Este comando solo hay que ejecutarlo una vez. Los dos siguientes comandos hay que ejecutarlos cada vez que se quiera subir una imagen. El primero de ellos es para asignar una etiqueta a nuestra imagen y el segundo para subir la imagen. El comando para asignar la etiqueta es necesario porque en caso contrario Docker no te permitirá subir la imagen.

Repetiremos todos estos pasos para cada microservicio ya que cada uno de ellos se ejecutará en un contenedor diferente salvo la página web que se ejecutará con la base datos en el mismo contenedor.

4.10 Despliegue de los contenedores en el cluster

Ya tenemos todo lo necesario para desplegar el sistema en el cluster. Para ello ejecutamos el comando:

$ dokcer service create

Este comando creará un servicio donde se ejecutará el contenedor. El master automáticamente decidirá en que nodo ejecutará el servicio balanceando la carga de trabajo entre ambos. A este comando de creación tiene como argumento obligatorio el nombre de la imagen en el repositorio de Docker Hub, además se le puden pasar otros argumentos como “--name” el cual permite darle un nombre al servicio o el argumento “--publish published=XXX, target=XXX” que permite redirigir el tráfico que publica en un puerto del contenedor a otro puerto del host. Este último argumento es necesario a la hora de desplegar la pagina web para luego poder acceder desde otra máquina. Si no se especifica nada con el argumento “--replicas” Docker Swarm automáticamente generará una sola replica, para este trabajo todos los servicios delegados tendrán una sola replica.

Una vez creados todos los servicios ya tendríamos el cluster completamente funcional con nuestro sistema.

Con esto queda finalizado el diseño e implementación de nuestro sistema.

Capítulo 5

Conclusiones y líneas futuras

5.1 Conclusiones

Tras la realización de este trabajo hemos llegado a las siguientes conclusiones; A pesar de que el diseño y la implementación de este trabajo no es muy complejo debido a que el tamaño del cluster y el número de sensores es reducido, es suficiente para mostrar como realizar una red de sensores IoT que funcione en un cluster virtualizando los servicios puede convertirse en una tarea compleja (sobre todo cuando el tamaño de la red y el cluster son elevados) pero que la virtualización en contenedores puede ayudar mucho a su gestión porque ofrece una alta modularidad.

Por otro lado, se han usado y combinado una gran variedad de tecnologías (que en mi caso desconocía en su mayoría) y he podido poner en práctica muchos de los conocimientos adquiridos durante la carrera. Este trabajo me ha ayudado a aprender tecnologías como Docker y Docker Swarm, de las cuales sabía de su existencia por su alta popularidad, o el protocolo MQTT el cual desconocía completamente y su relación con la IoT.

El trabajo de investigación sobre las tecnologías de virtualización ligera me ha ayudado a conocer mejor como se encuentra esta área de la informática y descubrir que hay otras muchas alternativas a Docker, alguna de ellas muy interesantes como los microkernels.

En definitiva, el Internet de las Cosas es un campo con mucho futuro y teniendo en cuenta la inmensa cantidad de dispositivos conectados que se espera que haya dentro de no mucho tiempo las tecnologías de virtualización ligera pueden ser un gran aliado para solucionar los posibles problemas de gestión que surjan.

5.2 Líneas futuras

De este trabajo podemos extraer dos líneas de trabajo muy interesantes y que también están relacionado con el área de la virtualización ligera:

1. **Escalabilidad**: una de las características más interesantes de los orquestadores de contenedores es su facilidad y velocidad para escalar contenedores tanto hacia arriba como hacia abajo. En el ámbito del Internet de las Cosas este aspecto es muy interesante puesto que muchos de los dispositivos conectados estarán en movimiento, siendo necesario que los sistemas puedan escalar los servicios de una forma rápida.
2. **Seguridad**: en este trabajo no se ha abordado nada acerca de los sistemas de seguridad que tienen estas tecnologías o cuales son lo que serían necesarios. La aparición de una gran cantidad de dispositivos conectados hace que surjan nuevos retos en esta área.

5.3 Problemas encontrados

En un principio para este trabajo el software que se había seleccionado para orquestar los contenedores era Kubernetes. En este tipo de cluster nos encontramos con 2 tipos de nodos, los nodos “master” que se encargan de distribuir la cargar y los nodos “worker” que se encargan de ejecutar los contenedores. Una vez instalado el cluster nos dimos cuenta de que en Kubernetes los nodos “master” no pueden ejecutar contenedores solo se encargan de tareas de balanceo de carga y monitorización. Como nuestro cluster contiene únicamente dos nodos, no tendrían mucho sentido utilizar Kubernetes puesto que perderíamos uno de ellos por lo que no sería muy eficiente utilizar está opción.

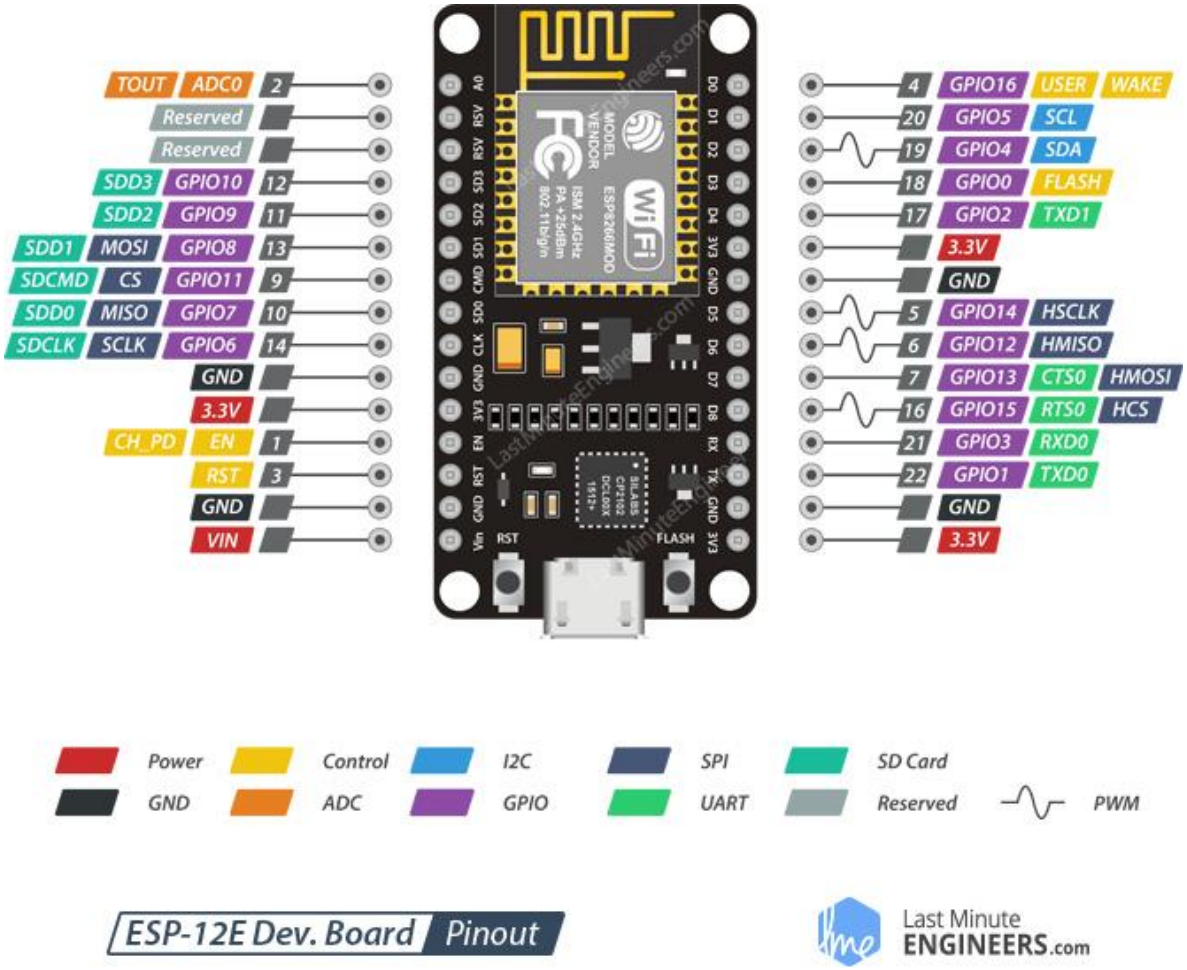
Una posible solución hubiese sido virtualizar raspberrys en un ordenador, pero descartamos está opción ya que añadiría más complejidad a un proceso (la instalación y manejo de un cluster en Kubernetes) que ya de por si no es sencillo. Al final la solución adoptada fue cambiar el software de orquestación de contenedores por Docker Swarm. Este sistema además de ser mucho más sencillo tanto en instalación como en uso permite que los nodos “master” sean “workers” al mismo tiempo.

Otro problema que ha provocado mucho retraso ha sido un error en la documentación del sensor PIR. En el sensor utilizado las marcas de colores de los cables no se correspondían con las habituales. Del sensor salen 3 cables, un

rojo otro blanco y uno negro. Este marcado lleva a pensar que el cable rojo es para la alimentación, el blanco para la salida de datos y el negro para tierra. Pero realmente el cable blanco es el de tierra y el negro para la salida de datos. Además, de que el sensor necesita 5V para funcionar correctamente. Hemos conseguido conocer esta información buscando en la página web del vendedor donde indica el marcado correcto de los cables y utilizar otra placa para suministrar los 5V al sensor.

Anexos

Anexo I: Hoja de pines ESP8266



Anexo II: instalación Kubernetes en Raspberry Pi

Además de realizar los pasos de los puntos 4.7.1 y 4.7.2 de esta memoria hay que ejecutar los siguientes comandos:

$ sudo dphys-swapfile swapoff

$ sudo dphys-swapfile uninstall

$ sudo update-rc.d dphys-swapfile remove

sudo swapon --summary

Estos comandos son para deshabilitar la memoria “swap” ya que si está activa Kubernetes puede tener problemas a la hora del funcionamiento.

Ahora hay que añadir la siguiente línea en el fichero /boot/cmdline.txt

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

Es importante que esto se añada al final del fichero en la misma línea y no en una nueva porque en caso contrario no funcionará.

Añadimos la clave de Kubernetes con el comando

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

Paso seguido instalamos “kubeadm” y “kubectl” que serán con los que arrancaremos nuestro cluster.

$ sudo apt-get install kubeadm

Una vez instalado todo iniciaremos el nodo master del cluster con el comando:

$ sudo kubeadm init --token-ttl=0

La opción “--token-ttl=0” es para que los tokens nunca expiren, esta opción es recomendable exclusivamente en los cluster de prueba.

Si la inicialización termina con existo aparecerá por pantalla un token como el parecido al que hemos visto en el apartado 4.7.3 y una serie de comandos que también tenemos que ejecutar.

$ mkdir -p $HOME/.kube

$ sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config

$ sudo chown $(id -u):$(id -g) $HOME/.kube/config

Una vez inicializado ejecutaríamos en el otro nodo el comando con el token. Con esto quedaría inicializado el cluster en Kubernetes.

Anexo III: Prueba de uso de Docker y Rocket

Docker

Comandos básicos de Docker

NOTA: todos los comandos van precedidos de **sudo docker**.

run : iniciamos un contendor de la imagen seleccionada.

build: crea una imagen a partir de un dockerfile.

start: inicia uno o más contenedores parados.

stop: para uno o más contenedores que están en ejecución.

ps -a: vemos todos los contenedores que hay, ya sean en ejecución o parados.

image ls: lista las imágenes que hay en la máquina.

rm: elimina uno o más contenedores.

rmi: elimina una o más imágenes.

Ejemplo de uso de docker

Antes de comenzar con nuestro ejemplo vamos a ver la diferencia entre un dockerfile una imagen docker (docker image) y contenedor docker (docker container). Estos tres conceptos están relacionados entre sí pero cada uno va en un orden diferente.

Todo comienza con el dockerfile que es un documento de texto plano el cual contiene los comandos necesarios que indica cómo se debe de crear la imagen docker. Una vez que el dockerfile está completo ejecutamos el comando **docker build** y se generará la imagen docker correspondiente. Por último, cuando ya tenemos la imagen al ejecutar el comando **docker run** creamos el contenedor docker que ejecuta nuestra imagen.

Por resumir un contenedor docker viene de una imagen docker y esta a su vez viene de un dockerfile.

Ahora haremos un ejemplo con el que podamos ver cómo funciona docker. El ejemplo consistirá en una pequeña aplicación web en python que nos diga hola mundo llamada app.py.

```
from flask import Flask
import os
import socket

app = Flask(__name__)
@app.route("/")

def hello():
    html = "<h3>Hello {name}!</h3> <b>Hostname:</b> {hostname}<br/>"
    return html.format(name=os.getenv("NAME", "world"), hostname=socket.gethostname())

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=4000)
```

Ahora crearemos el dockerfile correspondiente que tendrá la siguiente forma.

```
FROM python:2.7-slim
WORKDIR /app
ADD . /app
RUN pip install --trusted-host pypi.python.org Flask
ENV NAME World
CMD ["python", "app.py"]
```

FROM python:2.7-slim Esta línea indica la imagen base desde la que vamos a crear la nuestra, en este caso escogemos una que contiene los paquetes de python. Todos los dockerfile deben empezar con la cláusula FROM.

WORDIR /app aquí establecemos nuestro espacio de trabajo (working directory) llamado /app

ADD . /app añadimos los archivos y carpetas que tenemos en el directorio actual a nuestro espacio de trabajo. Este comando tiene dos argumentos ADD <ruta origen> <ruta destino> en el cual usamos el “.” para indicar que es el directorio actual. Como advertencia comprobar que en ruta de origen nunca se haya escrito el directorio raíz ya que en ese caso el comando copiará todo el contenido del disco duro al espacio de trabajo del contenedor.

RUN pip install --trusted-host pypi.python.org Flask ejecuta el comando pip install para instalar el paquete Flask. Con el comando RUN podemos ejecutar comandos como parte de la creación de la imagen.

ENV NAME World establece la variable de entorno NAME con valor World. Esta variable es usada en la aplicación.

CMD [“python”, “app.py”] esto ejecuta el interprete de python con la aplicación app.py. El comando CMD permite pasar un comando con una lista de argumentos de la forma CMD [“executable”, “arg1”,”arg2”].

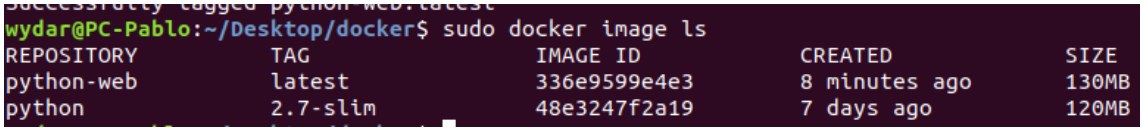
En este momento ya tenemos definido el dockerfile, ahora hay que ejecutarlo para generar la imagen con el comando

sudo docker build -t python-webb .

```
wydar@PC-Pablo:~/Desktop/docker$ sudo docker build -t python-web .
Sending build context to Docker daemon  3.072kB
Step 1/6 : FROM python:2.7-slim
2.7-slim: Pulling from library/python
27833a3ba0a5: Pull complete
8b35abcb27de: Pull complete
cd1fc6dee9fe: Pull complete
2c6a92003566: Pull complete
Digest: sha256:55c18f359c28061ddf70ee194439e9a5b2bdc205df3e7559b419dad10086fc89
Status: Downloaded newer image for python:2.7-slim
 ---> 48e3247f2a19
Step 2/6 : WORKDIR /app
 ---> Running in 9055e4860646
Removing intermediate container 9055e4860646
 ---> 7fa8301b9483
Step 3/6 : ADD . /app
 ---> 91cb5df8e036
Step 4/6 : RUN pip install --trusted-host pypi.python.org Flask
 ---> Running in 07708e168f21
DEPRECATION: Python 2.7 will reach the end of its life on January 1st, 2020. Please upgrade your Python as Python 2.7 won't be maintained after that d
ate. A future version of pip will drop support for Python 2.7.
Collecting Flask
  Downloading https://files.pythonhosted.org/packages/7f/e7/08578774ed4536d3242b14dacb4696386634607af824ea997202cd0edb4b/Flask-1.0.2-py2.py3-none-any.
whl (91kB)
Collecting itsdangerous>=0.24 (from Flask)
  Downloading https://files.pythonhosted.org/packages/76/ae/44b03b253d6fade317f32c24d100b3b35c2239807046a4c953c7b89fa49e/itsdangerous-1.1.0-py2.py3-no
ne-any.whl
Collecting Jinja2>=2.10 (from Flask)
  Downloading https://files.pythonhosted.org/packages/7f/ff/ae64bacdfc95f27a016a7bed8e8686763ba4d277a78ca76f32659220a731/Jinja2-2.10-py2.py3-none-any.
whl (126kB)
```

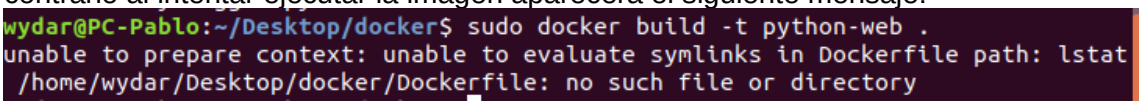
La opción -t es para ponerle una etiqueta a nuestra imagen, que en este caso es python-web. El “.” es para indicar que busque el dockerfile en el directorio actual, en caso contrario poner la ruta donde se encuentre el dockerfile.

con el comando **sudo docker image ls** vemos las imágenes creadas.

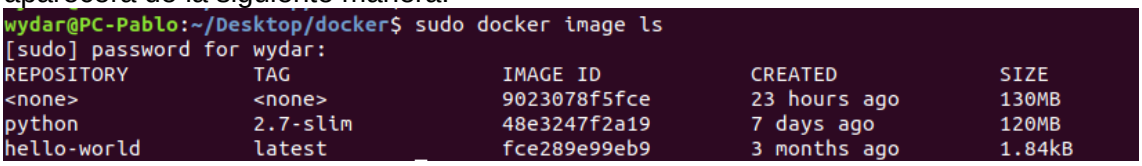


Antes de continuar, vamos realizar un par de anotaciones.

1. Asegurarse que el dockerfile se llame “dockerfile” o “Dockerfile” en caso contrario al intentar ejecutar la imagen aparecerá el siguiente mensaje:



2. si en el comando sudo docker build no ponemos la opción -t la imagen aparecerá de la siguiente manera:



Con el comando **sudo docker run -d -p 8080:4000 python-web** ejecutamos nuestra imagen y creamos un contenedor. La opción -**d** es para ejecutar el contenedor en segundo plano, la opción **-p 8080:4000** fija el puerto 8080 del host al puerto 4000 del contenedor.

Si vamos al navegador y escribimos la direccion http://127.0.0.1:8080 veremos lo siguiente:

Con el comando sudo docker ps podemos ver que contenedores hay en ejecución.

```
wydar@PC-Pablo:~/Desktop/docker$ sudo docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS                    NAMES
f152f06f96a7        python-web          "python app.py"     23 seconds ago      Up 22 seconds       0.0.0.0:8080->4000/tcp   heuristic_darwin
```

Rocket (rkt)

Guía de instalación

Actualmente no existe de forma oficial un paquete para Ubuntu. Con la ejecución de los siguientes comandos instalamos rkt.

$ gpg --recv-key 18AD5014C99EF7E3BA5F6CE950BDD3E0FC8A365E
$ wget https://github.com/rkt/rkt/releases/download/v1.29.0/rkt_1.29.0-1_amd64.deb
$ wget https://github.com/rkt/rkt/releases/download/v1.29.0/rkt_1.29.0-1_amd64.deb.asc
$ gpg --verify rkt_1.29.0-1_amd64.deb.asc
$ sudo dpkg -i rkt_1.29.0-1_amd64.deb

rkt utiliza un formato de imagen diferente al de docker, (ACI) Application Container Image. A diferencia de docker, cuando instalamos rkt solo ejecuta imagenes por tanto para poder compilar imagenes en formato ACI tenemos que instalar una herramienta llamada **acibuild** con el comando:

$ sudo apt-get install acbuild

Concepto previo

rkt utiliza **pods** como configuración de contenedores. Un pod es la unidad mínima de ejecución en rkt, es un conjunto de imágenes que se ejecutan de forma simultánea.

Comandos básicos en rkt

NOTA: todos los comandos van precedidos de **sudo rkt**.

run : iniciamos una imagen en un pod. Si le pasamos más de una imagen todas se ejecutarán en el mismo pod.

stop: para uno o más pods que esté en ejecución. Con la opción --force paramos el pod de forma forzosa.

list: lista todos los pods que hay, en rkt.

rm: elimina todos los recursos asociados a uno o más pods.

Ejemplo de uso de rkt

Del mismo modo que hemos hecho con docker ahora vamos a realizar un ejemplo básico para ver la funcionalidad de rkt.

Esta vez haremos un ejemplo parecido al anterior pero esta vez en go.

Tenemos el siguiente programa en go:

```
package main
import (
    "log"
    "net/http"
)
func main() {
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        log.Printf("request from %v\n", r.RemoteAddr)
        w.Write([]byte("hello\n"))
    })
    log.Fatal(http.ListenAndServe(":5500", nil))
}
```

Vamos a compilar los paquetes y dependencias de forma estática para poder enviar a rkt el programa sin dependencias externas. Esto lo hacemos con el siguiente comando:

$ **CGO_ENABLED=0 go build -ldflags '-extldflags "-static"'**

Ahora vamos a crear la imagen. Para ello vamos a utilizar la herramienta acbuild instalada previamente. Este paso es el equivalente a hacer el dockerfile, para ello ejecutaremos los siguientes comandos:

```
acbuild --debug begin
acbuild --debug set-name example-img
acbuild --debug copy example /bin/hello
acbuild --debug set-exec /bin/hello
acbuild --debug port add www tcp 5500
acbuild --debug write example.aci
acbuild --debug end
```

Estos comandos pueden ejecutarse uno por uno en la terminal pero de esta manera es mucho más tedioso, por lo tanto los ejecutaremos a través de un script.sh

Todos los comandos tienen la opción **--debug** para que a la hora de crear la imagen la herramienta nos escriba más información por pantalla.

acbuild --debug begin este comando es para inciar la creación de la imagen.

acbuild --debug set-name example-img aquí ponemos el nombre a la imagen.

acbuild --debug copy example /bin/hello copiamos el archivo example a la ruta /bin/hello

acbuild --debug set-exec /bin/hello establecemos un comando de ejecución.

acbuild --debug port add www tcp 5500 este comando permite mapear más fácilmente los puertos de dentro de la imagen con los del host.

acbuild --debug write example.aci escribimos el ACI en un fichero.

acbuild --debug end terminamos la creación de la imagen.

Una vez ejecutado el script no saldrá lo siguiente por pantalla:

```
wydar@PC-Pablo:~/Desktop/example$ ./app.sh
Beginning build with an empty ACI
Setting name of ACI to example-img
Copying host:example to aci:/bin/hello
Setting exec command [/bin/hello]
Writing ACI to example.aci
Ending the build
```

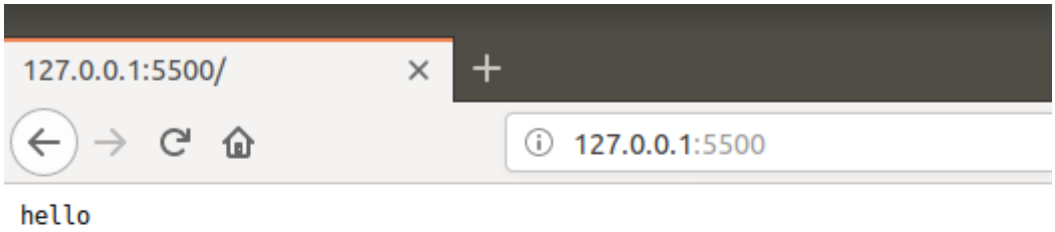
Una vez creada la imagen solo falta ejecutarla. Para ello usamos el comando:

$ sudo rkt --net=host --insecure-options=image run example.aci

con la opción --net=host indicamos que es localhost la dirección.

La opción --insecure-options=image es para poder ejecutar la imagen que hemos creado usando su nombre.

Si vamos al navegador y escribimos http://127.0.0.1:5500 vemos que efectivamente funciona.



En consola si escribimos el comando **$ sudo rkt list** vemos que nuestro pod está en ejecución.

```
wydar@PC-Pablo:~/Desktop/example$ sudo rkt list
UUID            APP             IMAGE NAME      STATE   CREATED         STARTED         NETWORKS
b2e49e39        example-img     example-img     running 19 seconds ago  19 seconds ago
```

Ejecutar un dockerfile en rkt

rkt permite ejecutar imágenes de docker. Hay una herramienta llamada dokcer2aci que transforma las imágenes docker en imágenes ACI. Vamos a ver un par de ejemplos de uso.

Primero instalamos la herramienta con el comando

$ sudo apt-get install docker2aci

Ahora vamos a usar la imagen docker que se creó en el apartado anterior como base del ejemplo.

Para poder hacer la transformación es necesario tener la imagen docker guardada en un fichero .tar, para ello usamos el comando

$ sudo docker python-web > pyapp.tar

Con el fichero .tar ejecutamos el siguiente comando

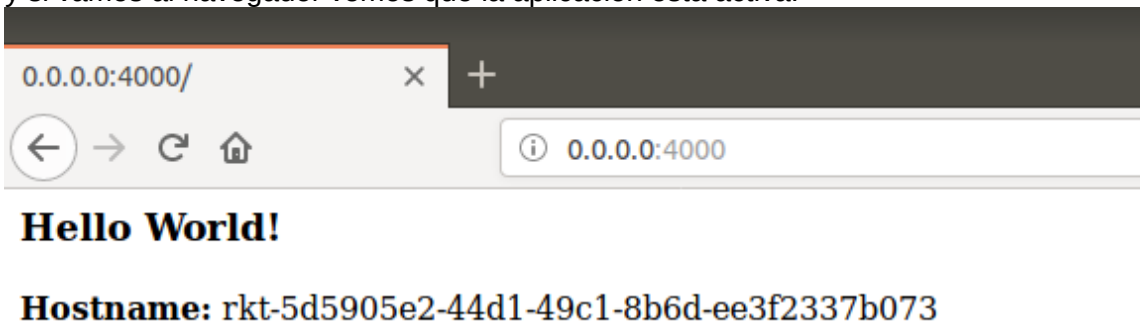
$ sudo docker2aci ./pyapp.tar

Donde ./pyapp.tar es la ruta donde tenemos el fichero .tar . Automáticamente, genera una imagen ACI llamada **python-web-latest.aci**

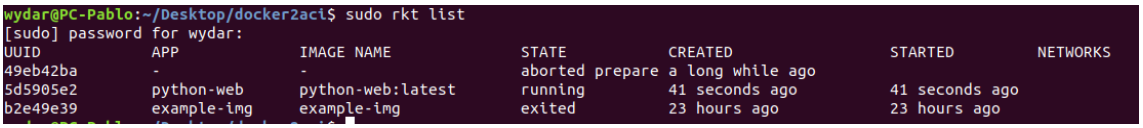
Con la imagen ya solo hay que ejecutarla en rkt

$ sudo rkt --net=host --insecure-options=image run python-web-latest.aci

y si vamos al navegador vemos que la aplicación está activa:



También podemos ver que el pod está en ejecución si lo listamos.



Aclaración final del Anexo III

Se ha intentado realizar el mismo ejemplo tanto en Docker como en Rocket pero ha resultado imposible debido a la gran dificultad de encontrar información acerca del uso esta última tecnología de forma nativa. Solo ha sido posible ejecutar el mismo ejemplo en ambas tecnologías gracias al uso de una herramienta que transforma los ficheros Docker en imágenes de Rocket.

Anexo IV: Preparación de las Raspberrys

1 Pasos previos

Estos pasos hay que realizarlos para ambas raspberrys.

Descargamos de la página de raspberry la imagen de raspbian strecth lite y creamos una unidad de arranque. En este caso usaremos el programa rufus.

Una vez que tenemos la memoria microSD encendemos la raspberry con raspbian y la conectamos por ethernet. Habilitamos el protocolo ssh para poder acceder de forma remota con los siguientes comandos:

$ sudo systemctl enable ssh

$sudo systemctl start ssh

Como en los sistemas raspbian el usuario y contraseñas por defecto son pi/raspberry por seguridad cambiaremos la contraseña con el siguiente comando:

$ sudo pass

El cual nos pedirá la nueva contraseña.

Ahora mismo la dirección ip de nuestra raspberry se asigna de forma dinámica y para el funcionamiento del sistema necesitamos que se asigne de forma estática. Para ello iremos al fichero /etc/dhcpcd.conf y al final del mismo añadiremos las siguientes líneas.

```
interface eth0
static ip_address=$ip/24
static routers=$dns
static domain_name_servers=$dns
```

Donde $ip será la dirección ip que vamos a establecer y $dns la dirección de nuestro router.

A continuación, actualizamos los repositorios para las instalaciones de software que realizaremos más adelante con los comandos

$ sudo apt-get update

$ sudo apt-get upgrade

Por último, reiniciamos la raspberry para que los cambios se hagan efectivos

2 Instalación del software

El software que vamos a necesitar instalar es Docker y Mosquitto-mqtt. Docker será necesario instalarlo en ambas raspberrys mientras que Mosquitto-mqtt solo será necesario en una de ellas.

2.1 Dokcer

Para instalar Docker ejecutamos el siguiente comando:

$ curl -sSL get.docker.com | sh

Además, como vamos a estar usando comandos Docker con frecuencia vamos a incluirlo en el grupo donde se encuentra el usuario administrador para no tener que poner el comando “sudo” cada vez que usemos un comando de Docker. Para ello ejecutamos el siguiente comando.

sudo usermod pi -aG Docker

Con esto ya tendríamos Docker instalado.

2.2 Mosquito-mqtt

Para descargarnos el programa ejecutaremos los siguientes comandos.

$ sudo apt-get install mosquitto

$ sudo apt-get install mosquitto-clients

Para añadir un poco más de seguridad vamos a establecer un usurario y contraseña para quien quiera publicar o subscribirse a algún tema en el bróker.

En el fichero /etc/mosquitto.conf eliminamos la última línea que pone

```
Include_dir /etc/mosquitto/conf.d
```

Y añadimos las siguientes líneas

```
allow_anonymous false
```

```
password_file /etc/mosuitto/pwfile
listener 1883
```

Con esto le indicamos al broker que no acepte conexiones anónimas y que este escuchando por el puerto 1883.

Para establecer cual será el usario y la contraseña que el broker tiene que aceptar lanzamos el siguiente comando:

$ sudo mosquitto_passwd -c /etc/mosquitto/pwfile (nombre del usuario)

Nos pedirá cual es la contraseña y una vez terminado reiniciamos la Raspberry y ya estaría terminado.

3 Iniciación del cluster

Una vez que tenemos las raspberrys configuradas y con el software instalado vamos a inicializar nuestro cluster en Dokcer Swarm. Para ello no hay que instalar ningún software extra debido a que Docker ya trae de serie Docker Swarm.

Para inicializarlo tenemos que ejecutar el siguiente comando:

$ docker swarm init

Una vez ejecutado nos saldrá por pantalla un mensaje con el siguiente formato:

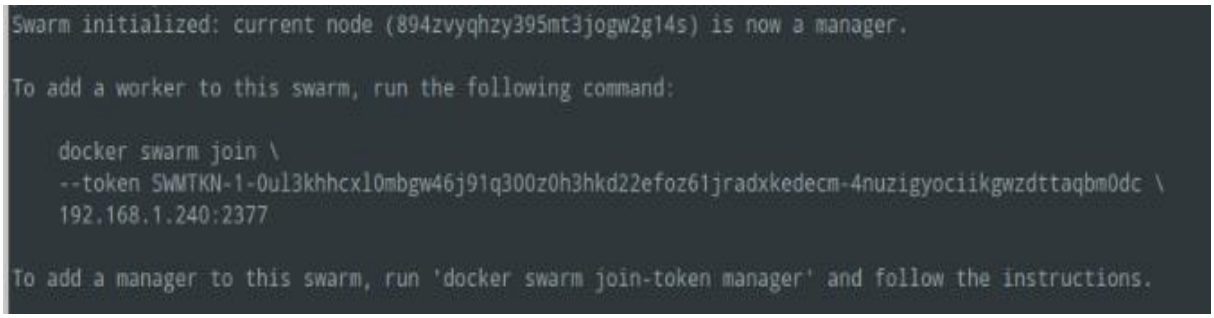


Figura 41 - Resultado init Docker Swarm

El token que aparece es importante guardarlo porque es necesario para añadir más nodos al cluster. El nodo en el que ejecutemos este comando de inicialización será el nodo master de nuestro cluster.

Ahora vamos al otro nodo y ejecutamos el comando join que aparece en la figura anterior para añadirlo al cluster.

Ya con los dos nodos en el cluster si ejecutamos el comando:

$ docker node ls

Nos aparecerá lo siguiente:

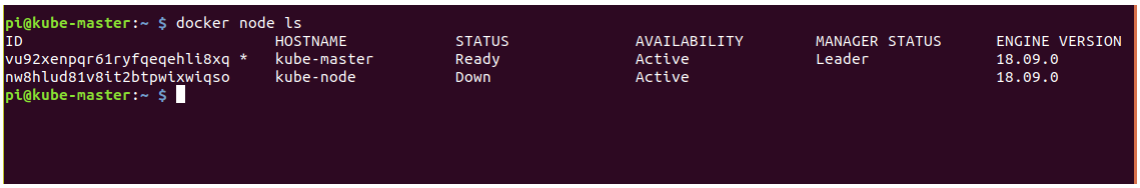


Figura 42 - Visualización nodos Docker Swarm

Como vemos en la figura anterior podemos ver cuál es el nombre del nodo, si está disponible o no, cuál de ellos es el líder entre otras cosas. Si nos fijamos en la columna “ID” la fila que tenga un asterisco nos indica en que nodo del cluster nos encontramos.

Bibliografía

- Raspberry Pi Foundation, Raspbian: https://www.raspberrypi.org/ (accedida julio/2019)
- MQTT sitio oficial, MQ Telemetry Transport http://mqtt.org/ (accedida julio/2019)
- Documentación Librería "PubSubClient.h" C++: https://pubsubclient.knolleary.net/api.html (accedida julio/2019)
- Documentación Librería "paho-mqtt" Python: https://pypi.org/project/paho-mqtt/#description (accedida julio/2019)
- Documentación Librería "flask" Python: http://flask.pocoo.org/docs/1.0/ (accedida julio/2019)
- Docker Inc, Doumentación de Docker Container: https://docs.docker.com/ (accedida julio/2019)
- Doker Inc, Documentación de Docker Swarm: https://docs.docker.com/engine/swarm/ (accedida julio/2019)
- The Linux Foundation, Documentación de Kubernetes: https://kubernetes.io/es/docs/home/ (accedida julio/2019)
- Madhavapeddy, A., Mortier, R., Rotsos, C., Scott, D., Singh, B., Gazagnaire, T., ... & Crowcroft, J. (2013). Unikernels: Library operating systems for the cloud. Acm Sigplan Notices, 48(4), 461-472.
- Bernstein, D. (2014). Containers and cloud: From lxc to docker to kubernetes. IEEE Cloud Computing, 1(3), 81-84.