

Taller de Proyecto II

2021

Informe Final de Proyecto

N10: (Desarrollo) Crear un clúster de Raspberry Pi's con Kubernetes y Containerd o Docker

Grupo de Desarrollo

- 1002/2 - Fausto Passerini
- 1593/9 - Cao Agustin Leonardo

Índice

Índice	1
Proyecto	2
Funcionalidad del Proyecto	3
Esquema Gráfico del Hardware del Proyecto	4
Documentación del Software del Proyecto	8
Archivo Dockerfile	9
Archivos .yaml	10
Casos de uso	12
Aplicación de prueba	13
Diagrama de Posibles Modos de Deployment	14
Documentación en Formato Gráfico, Video y Bitácora	16
Video	16
Bitácora	17
Investigación	17
Instalación de Docker	20
Pruebas del sensor	23
Preparativos para el cluster con K3s	25
Instalación de K3s en el nodo maestro	26
Instalación de K3s en nodos worker	27
Corriendo el script en un contenedor	29
Pagina web del sensor con Flask version 2.0.2	31
Servicio Flask en contenedores	33
Subiendo la imagen a docker hub	34
Deployment en el cluster de kubernetes	35
Propuestas de Proyectos a Futuro	40
Bibliografía	41
Anexo	43

1. Proyecto

Se busca desarrollar sobre computadoras de bajo costo, de tipo CBS, un clúster de Kubernetes¹ que permite el desarrollo y despliegue de aplicaciones utilizando herramientas modernas utilizadas en la industria, logrando un análisis a escala de los recursos utilizados y cómo afectan dichas herramientas a la CBS. Se usarán dos Raspberry Pi a modo de proof-of-concept, con la idea de que el sistema sea escalable a un número de “n” computadoras, Docker para contener el software en un ambiente virtualizado y aislado (containerized), y Kubernetes para la orquestación de los contenedores.

Para esto se emplearon dos Raspberry Pi 3B+. Dadas las bajas prestaciones de las mismas, se optó por utilizar K3S², una distribución “liviana” de kubernetes, en vez de una versión completa de dicho sistema, dado que esta es una herramienta que consume menos recursos en gran medida.

¹ Kubernetes: <https://kubernetes.io/>

² K3S: <https://k3s.io/>

2. Funcionalidad del Proyecto

El proyecto provee todas las funcionalidades inherentes a un clúster de Kubernetes y contenedores utilizando Docker y K3S + Containerd.

Expandingo, respecto a contenedores base, el mismo tiene la capacidad de soportar el despliegue de aplicaciones contenerizadas mediante el uso de Docker de forma manual. Además, se comprobó que el mismo es capaz de soportar la creación de imágenes personalizadas que sean compatibles con la arquitectura de las SBC utilizadas; en este caso, ARM.

Por el lado de Kubernetes, el clúster se encuentra manejado por el orquestador K3S, una versión liviana de Kubernetes pensada para aplicaciones de IoT y edge computing. Con el sistema desarrollado, se pueden desplegar aplicaciones dentro de pods, descargando imágenes desde los repositorios de Docker Hub³, y levantar todas las redes y los servicios que sean necesarios para dar soporte a dichas aplicaciones. Por supuesto, el sistema puede ser operado mediante la herramienta “*kubectl*”⁴, por lo que no presenta diferencias obvias a la hora de manipularlo igual que a una distribución de Kubernetes tradicional, como K8S.

En ambos casos, el usuario tiene el control total sobre lo que pueda realizarse en el clúster, con la ventaja de la automatización y escalabilidad asociada Kubernetes. El mismo puede ser accedido desde cualquier terminal o PC dentro de la misma red en la que este se encuentre conectado. Para su manipulación, se recomienda el uso del protocolo SSH.

³ Docker Hub: <https://hub.docker.com/>

⁴ Kubectl: <https://kubernetes.io/docs/reference/kubectl/overview/>

3. Esquema Gráfico del Hardware del Proyecto

El sistema fue planteado de forma en que las SBC puedan ser utilizadas en cualquier red establecida bajo el conjunto de direcciones IP 192.168.0.0/24. Para ello, se las debe conectar mediante su interfaz de cable Ethernet, ya que la misma provee la estabilidad necesaria para las aplicaciones que se buscan del sistema. La siguiente figura muestra un extracto del sistema.

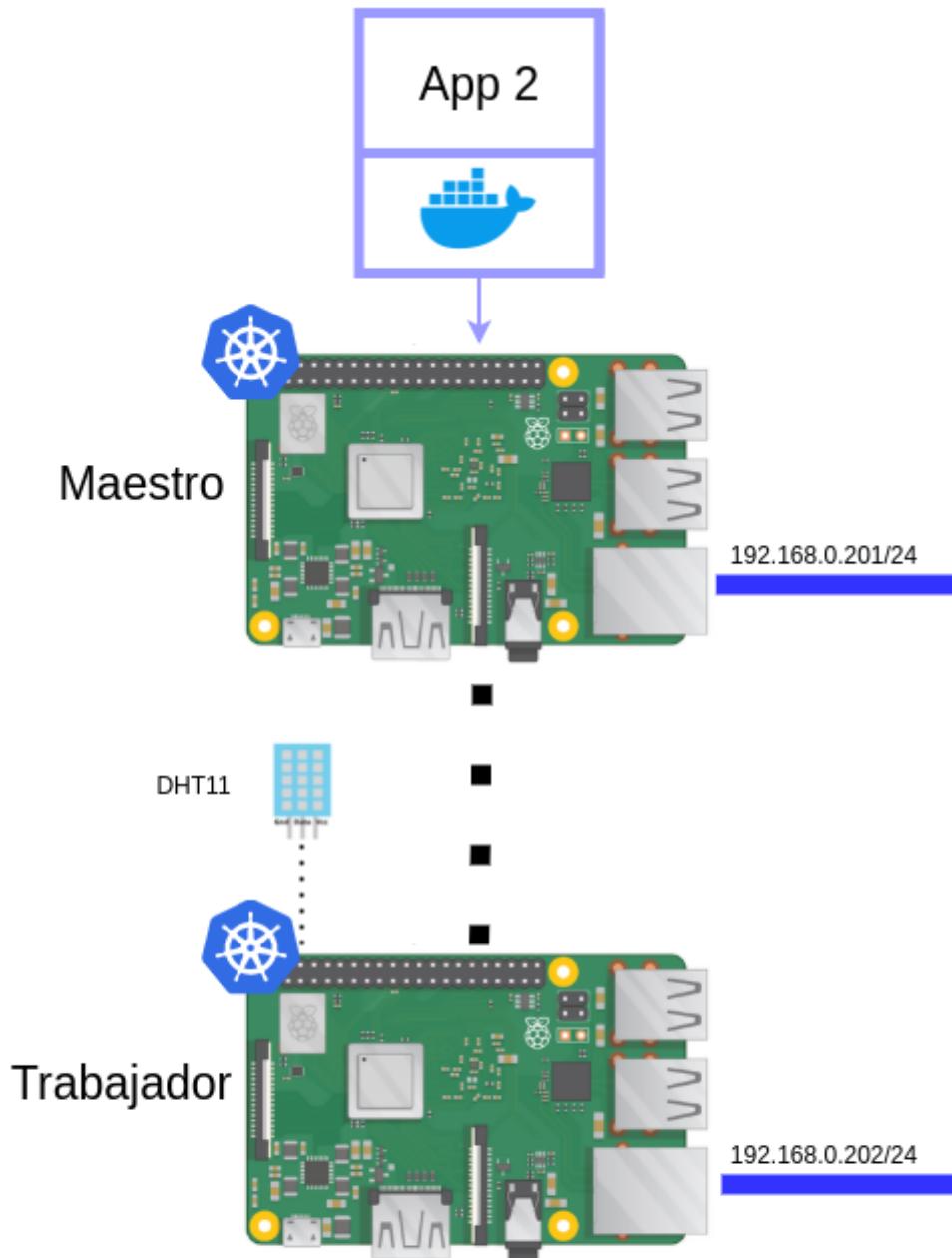


Fig. 1: Base del sistema.

Para montar el mismo, se utilizó una red hogareña que permitiese brindarle una conexión a internet manteniendo una cierta cantidad de comodidad a la hora de trabajar. Por ello, se optó por tomar un enrutador reciclado y transformarlo en un access point, el cual tuviera conectadas de forma cableada las raspberry pi's y brindará conexión a internet estando conectado a otro router de forma inalámbrica, debido a las restricciones de espacio con las que se contaba en el momento del montaje. Como se explicó antes, el mismo no influirá en el funcionamiento del sistema, ya que este es independiente de la red a la que se lo conecte, siempre y cuando dicha red trabaje con las IP's en el rango mencionado. Para acceder al sistema, uno debe conectarse mediante el protocolo SSH desde una terminal o PC dentro de la misma red. El sistema completo puede ser visualizado en la siguiente figura.

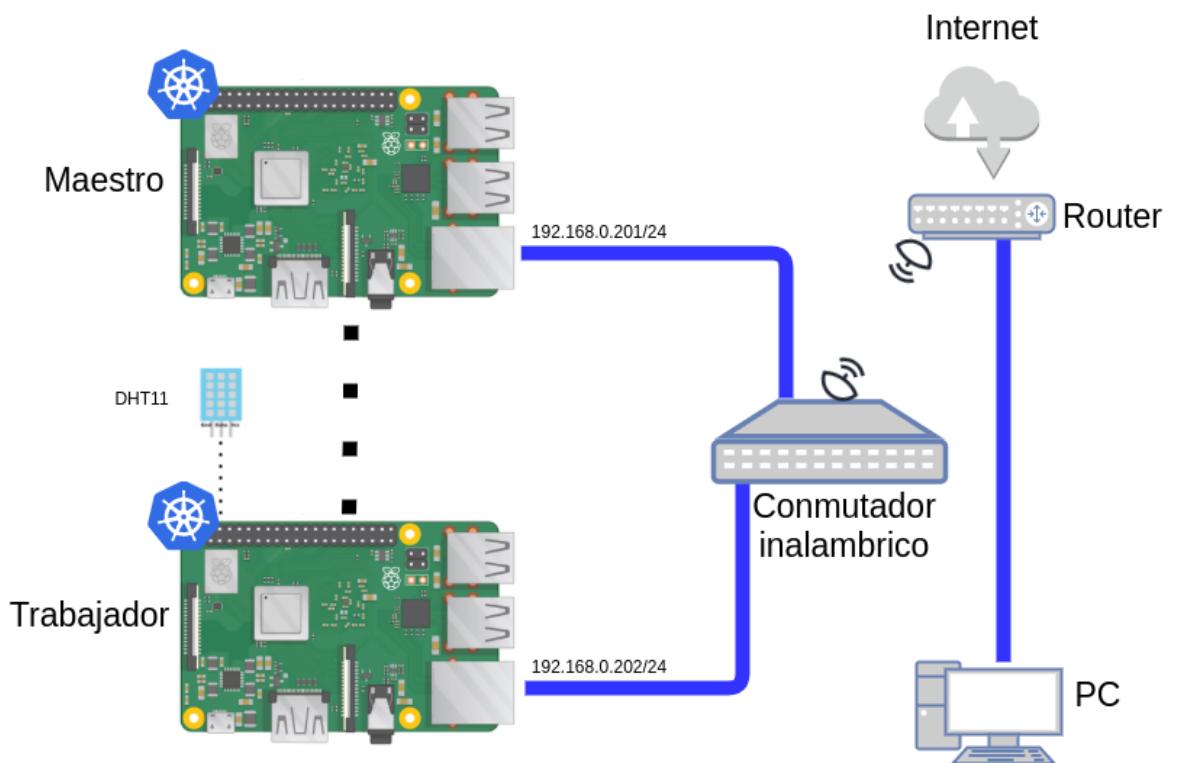


Fig. 2: Red completa utilizada para las pruebas.

En la siguiente figura se detallan los pines de la Raspberry utilizada

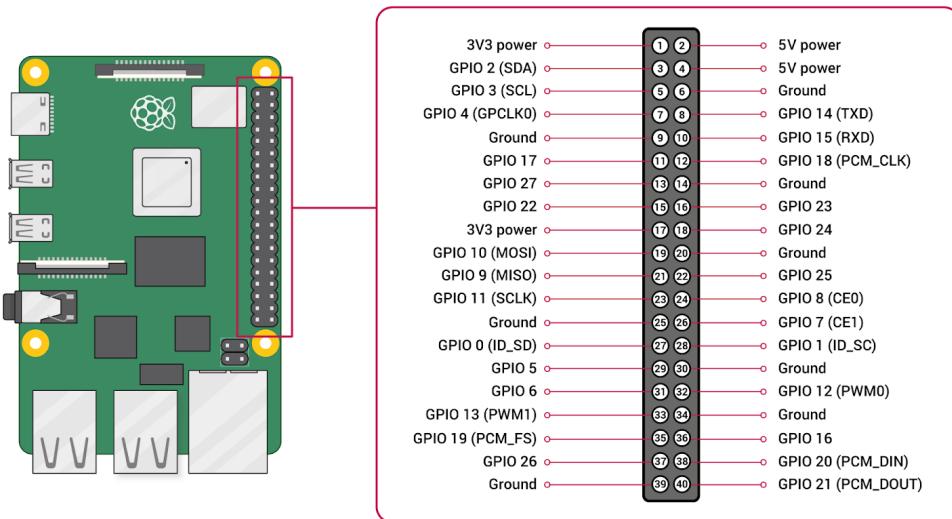


Figura 3: pines de la Raspberry Pi 3B+

A parte de las RPi, el único otro dispositivo utilizado fue un sensor de temperatura y humedad, el DHT11, conectado como se puede observar en la siguiente figura:

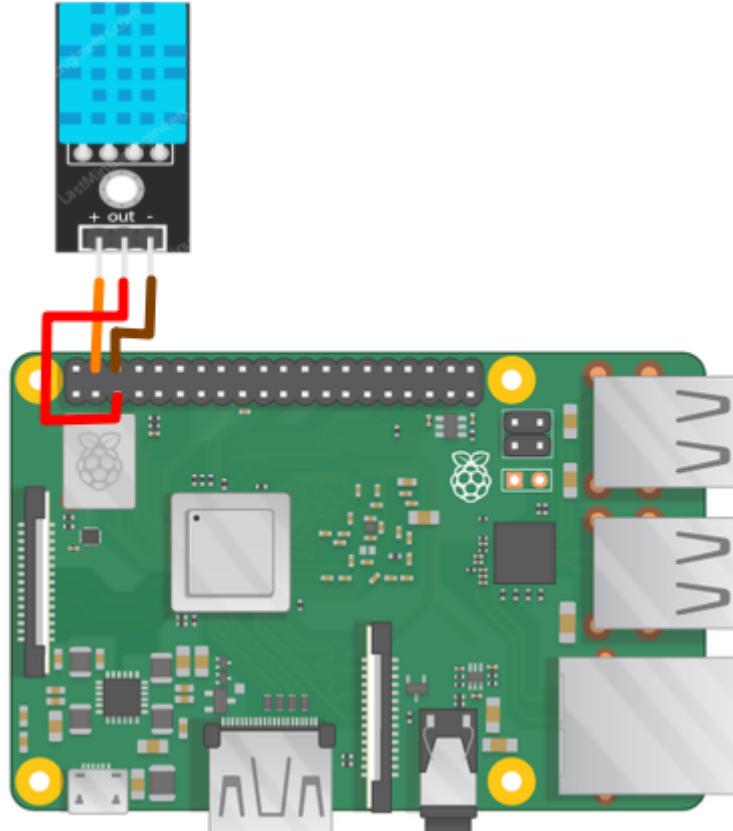


Figura 4: conexión del sensor a la computadora

Los elementos de hardware utilizados para el proyecto fueron:

Cantidad	Item	Precio unidad ¹	Valor ¹
2	Raspberry Pi 3B+ ²	\$ 7160	\$ 14,320.00
2	memoria flash USB 32GB ²	\$ 500	\$ 1,000.00
2	fuente de alimentación 3.6A USB ²	\$ 880	\$ 1,760.00
1	sensor medidor de temperatura y humedad DHT11 ²	\$ 290	\$ 290.00
2	cable micro-USB	\$ 450	\$ 900.00
2	cable UTP Cat 5e 1.2m	\$ 199	\$ 398.00
1	router TpLink TL-WR741ND v4.22 ³	\$ 1600	\$ 1,600.00
1	conector múltiple eléctrico de 4 tomas	\$ 890	\$ 890.00
Total	-	-	\$21,158.00

¹En pesos argentinos. Los valores provienen de los sitios web <https://www.mercadolibre.com.ar/> y <https://ar.mouser.com/> buscados en la fecha 13/10/2021. En caso de ser productos importados, incluyen todos los impuestos hasta la fecha. No se incluyen gastos de envío ni transporte.

² Provisto/financiado por la cátedra.

³ Reutilizado de otro proyecto.

4. Documentación del Software del Proyecto

En sí mismo, el proyecto no requirió de un desarrollo exhaustivo de programas, dado que la gran mayoría del trabajo de desarrollo e investigación se centró en la configuración de Docker y Kubernetes, el estudio de dichas herramientas y el análisis de las distintas alternativas presentes en el mercado hoy en día.

Se optó por utilizar K3S en vez de K8S debido a las bajas prestaciones del hardware del sistema que se posee. Al analizar la performance (Fig. 4.1 y 4.2), se confirmó que verdaderamente el sistema no hubiera funcionado de manera estable con un sistema más demandante en cuanto a recursos como lo es K8S.

Un detalle a recalcar respecto a K3S es que se decidió optar por la configuración recomendada por defecto del desarrollador, la cual emplea como sistema de contenedores a Containerd en vez de Docker. En la práctica y en cuanto al uso de contenedores, el cambio de sistema no afectó el transcurso del proyecto.

El sistema de muestra consiste de un archivo Dockerfile (archivo de configuración de Docker), un sitio web sencillo implementado en Flask, y dos archivos yaml. De estos, caben destacar los archivos dockerfile, que se utilizan para la creación de imágenes que contengan una aplicación la cual se desea contenerizar; y los archivos yaml, usados para configurar el deployment de un container dentro de un cluster de Kubernetes y toda la infraestructura necesaria para el funcionamiento de dicha aplicación.

Debido a la importancia que estos tienen, se hará una breve explicación de estos usando el código del proyecto como ejemplo:

Archivo Dockerfile

```
FROM arm32v7/python:3.7.12-buster

RUN apt-get update && apt-get install -y python3-pip=18.1-5 rpi.gpio=0.6.5-1

RUN python3 -m pip install --upgrade pip setuptools wheel

RUN pip3 install Adafruit_DHT==1.4.0 Flask==2.0.2

COPY flask_dht11_sensor_test.py .

CMD [ "python3", "flask_dht11_sensor_test.py" ]
```

FROM inicializa un nuevo *build stage* desde la imagen base (en este caso arm32v7/python:3.7.12-buster)
COPY copia los archivos del proyecto desde el directorio indicado
RUN ejecuta el comando indicado para preparar la imagen
CMD indica que es lo que debe ejecutar la imagen dentro del contenedor al iniciarse.

Para correr la aplicación mediante Docker, basta con utilizar la siguiente sentencia:

```
docker container run --privileged -p 4000:5000 alcaolpg/flask-dht11-rpi:latest
```

Como es de esperarse, la imagen⁵ se encuentra actualmente en un repositorio público en Docker Hub.

⁵ Imagen del proyecto: <https://hub.docker.com/repository/docker/alcaolpg/flask-dht11-rpi>

Archivos .yaml

Estos archivos son archivos de configuración o manifiesto que utilizan el lenguaje YAML⁶ (Yet Another Markdown Language o YAML Ain't Markup Language). Para una mejor comprensión del mismo, se detallan en forma de comentarios el funcionamiento de aquellas líneas más relevantes.

El sistema de prueba requiere de dos archivos yaml. Uno para realizar el deployment de la aplicación dentro de un contenedor, y otro que permita crear un servicio que redirige las solicitudes del clúster desde un determinado puerto al contenedor que está corriendo nuestra aplicación.

```
apiVersion: apps/v1 # Versión de Kubernetes para este objeto
kind: Deployment # Tipo de objeto
metadata:
  name: sensor # Nombre del objeto
  namespace: default # Nombre del espacio de trabajo
spec:
  replicas: 1 # Cantidad de pods que se crearan
  selector:
    matchLabels:
      app: sensor # Nombre del contenedor para identificación de la aplicación
  template:
    metadata:
      labels:
        app: sensor # Nombre del contenedor
    spec:
      containers:
        - name: flask-dht11-rpi # Nombre de la aplicación
          image: alcaolpg/flask-dht11-rpi:latest # Nombre de la imagen que debe
            # buscarse en docker hub
        ports:
          - containerPort: 5000 # Puerto utilizado para comunicarse con la
            # aplicación
        protocol: TCP
      securityContext:
        privileged: true # Permite que el contenedor acceda a los
          # dispositivos GPIO
    nodeSelector:
      dhtsensor: "yes" # Selecciona los nodos que poseen el dispositivo DHT11
```

⁶ YAML: <https://en.wikipedia.org/wiki/YAML>

Básicamente, el archivo lo que hará es crear una instancia de nuestra aplicación dentro de un contenedor, el cual correrá en modo privilegiado dentro de la SBC que tenga conectado el sensor de humedad y temperatura DHT11.

Cabe recalcar dos particularidades. Tanto en el uso de docker en sí, como el deployment realizado mediante kubernetes, lo estamos haciendo en modo privilegiado. En un ambiente real, dicha configuración no es para nada recomendada⁷. Esto se debe a que al correr un contenedor en modo privilegiado, se pierde la capa de protección característica de los sistemas contenerizados, y se pasa a un modo de operación del contenedor más cercano al sistema operativo del host. Básicamente, otorga a un contenedor capacidades de administrador, permitiendo acceso a todos los dispositivos del sistema host. En nuestro caso, esto es necesario para poder acceder al uso de los puertos GPIO de las Raspberry Pi, donde se encuentra conectado nuestro sensor. Debido a que nuestro sistema se encuentra operando en un ambiente cerrado y controlado, no ocasiona ningún riesgo relevante, pero es un detalle a tener en mente.

La otra característica a recalcar es el uso de etiquetas⁸. Dado que solo se cuenta con un único sensor, la aplicación debe correr sobre aquel nodo que cuente con dicho sensor conectado a él. En el caso de Docker, esto nos limita a únicamente levantar la aplicación dentro de un contenedor en aquella raspberry pi que cuente con el sensor. En cambio, al utilizar kubernetes, el nodo maestro es quien decide sobre qué nodo se va a correr la aplicación. Por ello, se estableció una etiqueta que señala al nodo que cuenta con el sensor conectado a él. Al especificar dentro del archivo .yaml que se deben emplear los nodos que cuenten con dicha etiqueta, nos aseguramos que el maestro siempre elija los nodos indicados para dicha aplicación.

En cuanto al otro archivo .yaml, a continuación se muestra un detalle del mismo. Como ya fue mencionado, el mismo permite crear un servicio que comunique la aplicación contenerizada dentro del clúster con el exterior.

⁷ Docker Privileged: <https://phoenixnap.com/kb/docker-privileged>

⁸ Assigning Pods to Nodes: <https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/>
How to use Node Selectors in Kubernetes: <https://www.howtoforge.com/use-node-selectors-in-kubernetes/>

```

apiVersion: v1
kind: Service
metadata:
  name: sensor-nodeport # Nombre del servicio
  namespace: default
spec:
  type: NodePort
  selector:
    app: sensor # Nombre del contenedor que identifica a la
    aplicación
  ports:
    - port: 5000 # Puerto utilizado para comunicarse con la
      # aplicación
    targetPort: 5000 # Puerto del contenedor de la aplicación
    nodePort: 30001 # Puerto utilizado para acceder al contenedor
      # desde el exterior

```

Casos de uso

En la siguiente figura se observa un diagrama de los casos de uso del clúster:

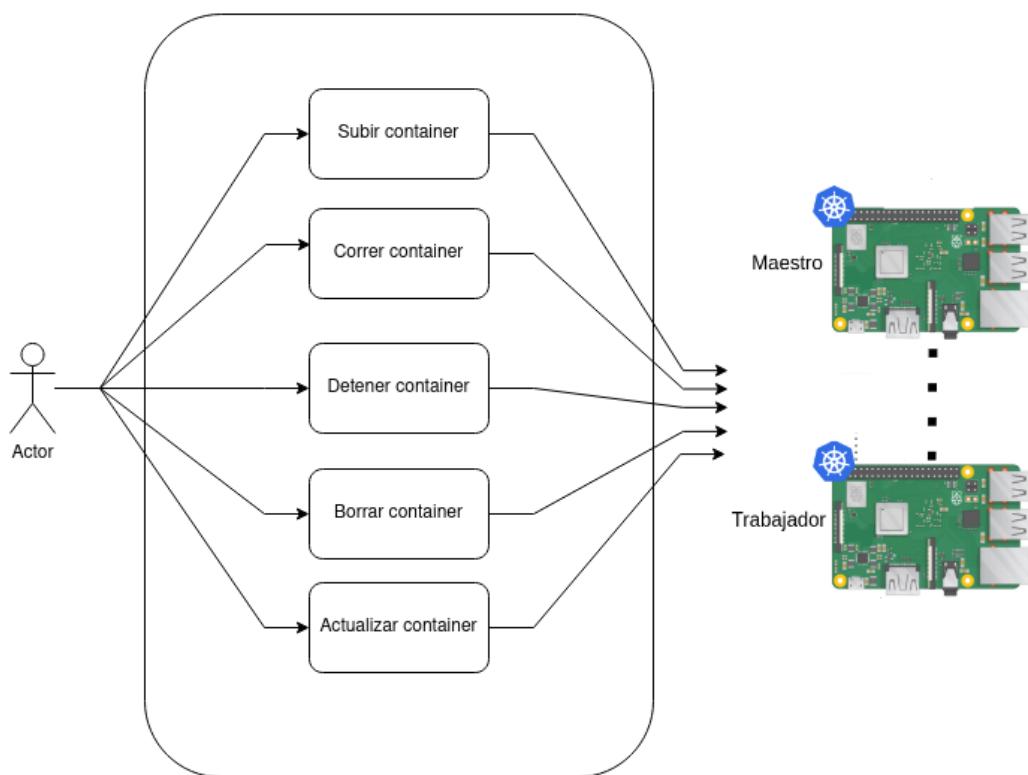


Figura 5: Casos de uso del sistema

Aplicación de prueba

La aplicación de prueba fue desarrollada mediante un modelo de prototipado, combinado con técnicas iterativas e incrementales.

El prototipo comenzó como una simple aplicación que permitía la lectura del sensor conectado a los puertos 4 (5V PWR) y 6 (GND) para alimentación, y 5 (GPIO 3) de una raspberry pi. Sobre esta, se realizaron ciertas mejoras que permitieron aumentar las probabilidades de éxito de una lectura, siendo dichas probabilidades prácticamente absolutas en la versión actual. En esta instancia, se utilizaron las librerías de Adafruit_DHT versión 1.4.0.

Una vez que dicho sistema se encontraba en funcionamiento, se pasaron a ejecutar pruebas en contenedores mediante Docker.

Para expandir la funcionalidad del mismo y ofrecer una vista desde una página web, se decidió utilizar Flask en su versión 2.0.2.

A continuación puede verse el código completo correspondiente a la aplicación. Además, las figuras 6 y 7 muestran la respuesta del servidor al ser invocado.

```
# import the Flask class from the flask module
from flask import Flask, render_template
import Adafruit_DHT

DATA_PIN = 3
SENSOR = Adafruit_DHT.DHT11

# create the application object
app = Flask(__name__)

# use decorators to link the function to a url
@app.route('/')
def home():
    humidity, temperature = Adafruit_DHT.read_retry(SENSOR, DATA_PIN)

    if ((temperature != None) and ((humidity != None))):
        return("Temperatura={0:0.1f}°C | "
Humedad={1:0.1f}%HR".format(temperature, humidity))
    else:
        return("Falla de lectura. Reintentando...")

# start the server with the 'run()' method
if __name__ == '__main__':
    port = int(os.environ.get("PORT", 5000))
    app.run(debug=True, host='0.0.0.0', port=port)
```

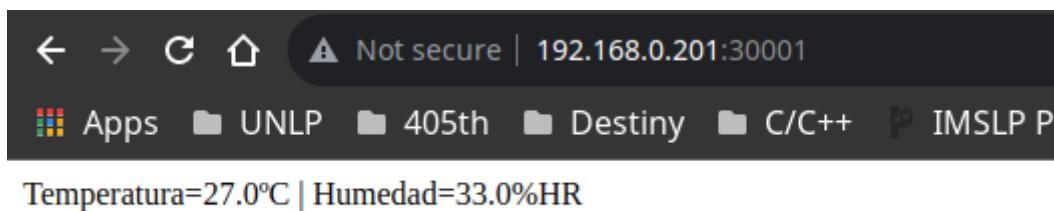


Fig. 6: Página web del servicio.

```
* Serving Flask app 'flask_dht11_sensor_test' (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: on
* Running on all addresses.
  WARNING: This is a development server. Do not use it in a production deployment.
* Running on http://10.42.1.17:5000/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 302-685-115
10.42.0.0 - - [19/Dec/2021 04:56:58] "GET / HTTP/1.1" 200 -
10.42.0.0 - - [19/Dec/2021 04:56:59] "GET /favicon.ico HTTP/1.1" 404 -
```

Fig. 7: Respuesta del servidor.

En este caso, el servicio se encuentra contenerizado dentro del cluster.

Diagrama de Posibles Modos de Deployment

La siguiente figura muestra cómo la aplicación desarrollada correría sobre el sistema al utilizar tanto Docker de forma directa como al utilizar kubernetes.

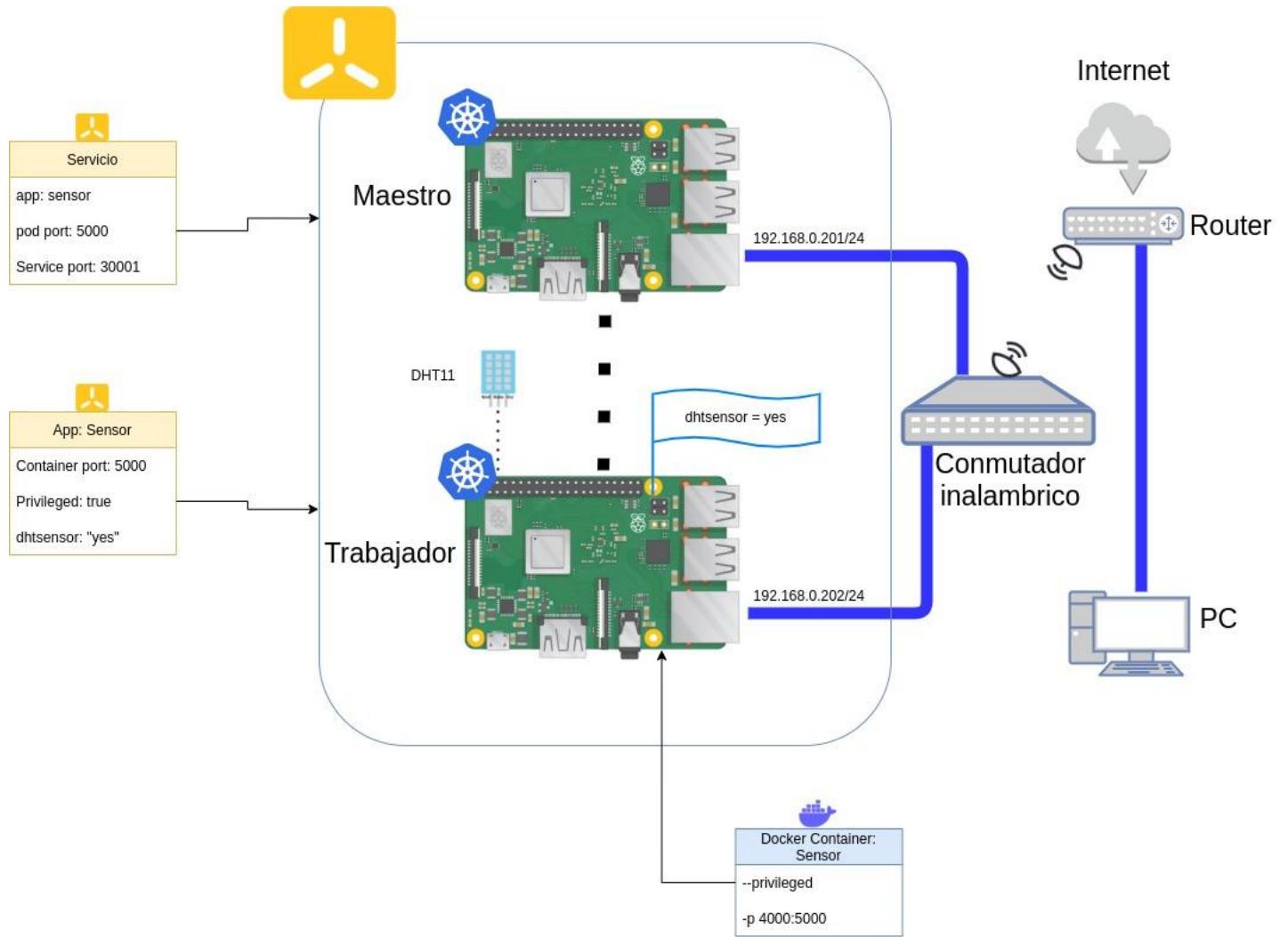


Fig. 8: Diagrama de deployments

Todo el código fuente se encuentra disponible en [el repositorio en github](#) del proyecto.

5. Documentación en Formato Gráfico, Video y Bitácora

La siguiente figura es una fotografía de las partes físicas del sistema, armado y en funcionamiento:

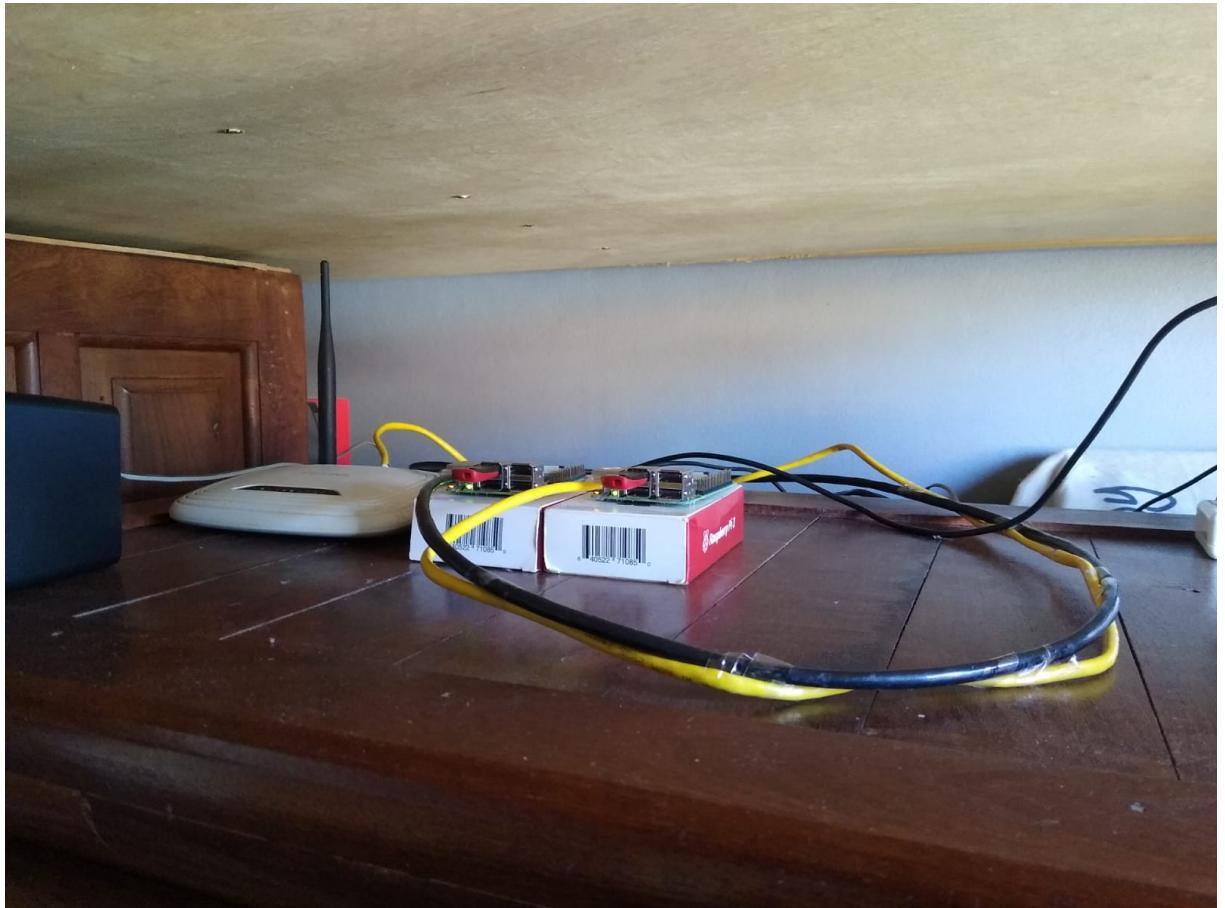


Figura 6: fotografía del sistema

Video

En el siguiente enlace, puede encontrarse un video explicativo resumido con las características del proyecto y la muestra del estado del mismo.

<https://www.youtube.com/watch?v=tQI6T4QSQMo>

Bitácora

Se recomienda la lectura de la misma directamente sobre la wiki del repositorio del proyecto en github.⁹

Investigación

16 de Agosto: me pasé una hora leyendo la documentación de docker en <https://docs.docker.com/get-started/>.

29 de Agosto: Recopilar información sobre docker y kubernetes.

¿Qué es Kubernetes?
<https://tanzu.vmware.com/developer/guides/kubernetes/what-is-kubernetes/>

Introducción a Docker Containers en Kubernetes:
<https://tanzu.vmware.com/developer/guides/kubernetes/what-is-kubernetes/>

Escribimos la plantilla de presentación del proyecto y el powerpoint correspondiente.

02 de Septiembre: Información sobre sistemas de Alta Disponibilidad (High Availability).

<https://us.sios.com/what-we-do/high-availability/>

<https://medium.com/velotio-perspectives/demystifying-high-availability-in-kubernetes-using-kubeadm-3d83ed8c458b>

05 de Septiembre: hecho el tutorial oficial de docker: <https://docs.docker.com/get-started/>

⁹ <https://github.com/tpII/2021-n10-kubernetes-rpi/wiki/Bit%C3%A1cora>

Documentación Docker: <https://docs.docker.com/get-started/>

Iniciando con Docker Containers en Kubernetes:
<https://tanzu.vmware.com/developer/guides/kubernetes/what-is-kubernetes/>

Alta disponibilidad: <https://us.sios.com/what-we-do/high-availability/>

Alta disponibilidad con Kubernetes:
<https://medium.com/velotio-perspectives/demystifying-high-availability-in-kubernetes-using-kubeadm-3d83ed8c458b>

Tutorial oficial de docker: <https://docs.docker.com/get-started/>

11 de Septiembre: experimentos Dockerizando un proyecto personal y desplegándolo a Heroku

15 de Septiembre: lectura de los conceptos teóricos de Kubernetes

16 de Septiembre: más conceptos teóricos de Kubernetes.

19 de Septiembre: hecho el tutorial interactivo oficial de Kubernetes

¿Qué es Kubernetes?
<https://tanzu.vmware.com/developer/guides/kubernetes/what-is-kubernetes/>

Documentación oficial de Kubernetes: <https://kubernetes.io/docs/home/>

29 de Septiembre: Puesta en marcha del sistema

Recibidas las SBC, se procedió a evaluar el estado y los materiales.

El sistema operativo que se utilizará para el proyecto en cada una de las Raspberry Pi será Raspbian 10 (Buster) con kernel Linux versión 5.10. Se comprobó utilizando los comandos

```
uname -r  
cat /etc/os-release
```

Para no tener que modificar la configuración de Access Point de las Raspberry Pi, se las conecta mediante cables ethernet a un access point.

El Access Point se encuentra alejado del router, por lo que se lo configuró para que funcione en modo puente WDS, a fin de brindar conexión a internet a las Raspberry Pi.

Modelo del Access Point: TpLink TL-WR741ND v4.22

Guía de configuración para modo WDS Bridge

Se estableció una IP fija para cada RPi, modificando la configuración del archivo /etc/dhcpcd.conf

```
sudo nano /etc/dhcpcd.conf
```

Dentro del mismo, se agregó al final del archivo:

```
interface eth0  
#reemplazar 'x' por el número identificador de la CBC  
static ip_address=192.168.0.20x/24  
static routers=192.168.0.1  
static domain_name_servers=192.168.0.1
```

x es el número de la RPi, 1 o 2.

Esta configuración permite que las Raspberry Pi operen vía cable ethernet y sean consistentes al conectarse a cualquier red cuyo router tenga la dirección 192.168.0.1, y las direcciones 192.168.0.201/24 y 192.168.0.202/24 se encuentren disponibles.

Con la finalidad de ahorrar energía y reducir interferencias, se desactivan las antenas de WiFi de las Raspberry Pi.

```
sudo rfkill block wifi
```

Si se quiere volver a activarlas:

```
sudo rfkill unblock wifi
```

Por último, se comprobó que ninguna de las RPis se encontrará funcionando en condiciones anormales o que pudiesen afectar su rendimiento

```
vcgencmd get_throttled
```

Referencia documentación Raspberry Pi

Si deseamos interpretar la respuesta, convertimos el valor hexadecimal que devuelve el comando y comparamos bit a bit. En nuestro caso, la respuesta fue 0x0, por lo que no se encuentran condiciones anormales.

Instalación de Docker

A fin de realizar pruebas con contenedores en el sistema antes de comenzar con el uso de un orquestador, instalamos docker y comprobamos su funcionamiento.

Para instalarlo, utilizamos un script recomendado por docker para instalar dicho sistema en casi cualquier distribución de linux

```
curl -sSL https://get.docker.com/ | sh
```

Agregamos al usuario actual al grupo "docker" para poder utilizar los comandos de docker sin necesidad de ser superusuario

```
sudo usermod -aG docker $USER  
comprobamos que docker esté corriendo utilizando
```

Iniciamos el daemon de docker

```
sudo systemctl start docker
```

```
docker version
```

OPCIONAL: Aparentemente, al instalar docker se agregó la entrada de que permite iniciar el daemon de docker junto con el resto del sistema. Si esta funcionalidad es necesaria, puede activarse utilizando

```
sudo systemctl enable docker
```

Si se desea desactivar la misma

```
sudo systemctl disable docker
```

Como prueba, iniciamos un contenedor con una imagen de Nginx (un popular webserver).

```
docker container run --rm -p 80:80 nginx
```

--rm indica a docker que debe eliminar el contenedor una vez terminada su ejecución

-p indica los puertos a exponer y ligar entre la SBC y en la red de contenedores, respectivamente.

Luego, utilizamos un navegador desde cualquier computadora conectada a la misma red que las SBC e insertamos la dirección IP de aquella que hemos configurado con el servidor Nginx.

Debería aparecer un mensaje como el siguiente.

Para finalizar el proceso, simplemente pulsamos ctrl + c

24 de Octubre: se buscó un proyecto semilla para facilitar la configuración inicial de la aplicación de prueba. Despues de probar con varios proyectos públicos encontrados en github,

la gran mayoría de los cuales o no incluían una base de datos o no estaban demasiado desactualizados como para funcionar, se terminó optando por el siguiente: <https://github.com/rgaino/docker-node-react-mysql-boilerplate>

Hecho el git pull, el proyecto se puede levantar usan el comando:

```
docker-compose up
```

Una vez que se terminó de levantar, se corrobora que el contenedor esté corriendo correctamente con el comando:

```
docker ps
```

Luego, a modo de prueba se procede a crear una nueva tabla en la BD. Para esto se utilizó el programa DBeaver. La conexión a la base de datos local, si no se cambia la configuración del proyecto semilla, se configura de la siguiente manera:

```
Server Host: localhost Port: 3306 Database: boilerplate_db Username: dbuser Password: dbuserpwd Driver: MySQL
```

Se procede a crear una tabla sencilla usando la interfaz gráfica de DBeaver. El código DDL generado es el siguiente:

```
CREATE TABLE boilerplate_db.Sensores ( temp DOUBLE NULL, id BIGINT UNSIGNED auto_increment NOT NULL, CONSTRAINT Sensores_PK PRIMARY KEY (id) ) ENGINE=InnoDB DEFAULT CHARSET=latin1 COLLATE=latin1_swedish_ci;
```

Se persisten los cambios y se corrobora que la base de datos está funcionando correctamente.

7 de noviembre:

Se investigaron los framework express y sequelize de javascript para escribir una API REST y manejar los ORM respectivamente.

Pruebas del sensor

13 de noviembre:

Para las pruebas con el sensor, se decidió ejecutar directamente sobre la placa un script en python para comprobar su funcionamiento.

El sensor utilizado es el DHT11, que es un sensor de temperatura y humedad. La placa elegida para correr el script es la misma que utilizaremos como worker en el cluster.

<https://www.mouser.com/datasheet/2/758/DHT11-Technical-Data-Sheet-Translated-Version-143054.pdf>

Se trata de un sensor que puede medir temperaturas en un rango de 0°C a 50°C con una precisión de $\pm 2^\circ\text{C}$ y humedades en un rango de 20%HR a 90%HR con una precisión de $\pm 5\%$ HR. La resolución mínima en ambos casos es de 1°C y 1%HR respectivamente.

HR = Humedad Relativa

En su paquete original, el sensor requiere de una resistencia de pull up de alrededor de 5k en el pin de datos para funcionar. Sin embargo, la versión que nosotros poseemos es la de un módulo que ya cuenta con dicha resistencia, más un capacitor de desacople ubicado en el pin de alimentación para prevenir fallas por cambios bruscos en la alimentación (filtro pasa-bajos).

<http://tutorialesdeelectronicabasica.blogspot.com/2021/03/que-es-el-condensador-de.html>

Para conectarlo, se utilizaron los pines 4 (5V PWR) y 6 (GND) para alimentación, y 5 (GPIO 3) para la transmisión de datos. A su vez, fue necesario instalar los paquetes python3-pip y python3-dev para utilizar el módulo.

El programa de prueba consiste en un simple script que lee los datos del sensor y los muestra por consola. Se realizan lecturas cada 5 segundos. Un detalle que fue descubierto es que dicho sensor es propenso a fallas al leer, sin importar el tiempo de espera entre lecturas. Por lo tanto, se utilizó una función especial de la librería de Adafruit, la cual realiza 15 lecturas sucesivas en intervalos de 2 segundos hasta obtener una lectura válida. Esto reduce la probabilidad de obtener una lectura errónea.

<https://learn.adafruit.com/dht-humidity-sensing-on-raspberry-pi-with-gdocs-logging/python-setup>

Las librerías necesarias para la ejecución del script son Adafruit_DHT, versión 1.4.0 y time. Raspbian, por defecto, no trae instalado el manejador de paquetes de python, pip. Por lo tanto, se recomienda proceder con los siguientes comandos antes de correr el script.

```
sudo apt-get install python3-dev python3-pip  
sudo python3 -m pip install --upgrade pip setuptools wheel  
sudo pip3 install Adafruit_DHT
```

El programa de prueba consiste en el siguiente script:

```
import Adafruit_DHT  
import time  
  
DATA_PIN = 3  
SENSOR = Adafruit_DHT.DHT11
```

```

while True:

    humidity, temperature = Adafruit_DHT.read_retry(SENSOR, DATA_PIN)

    if ((temperature != None) and ((humidity != None)):

        print("Temperatura={0:0.1f}°C | Humedad={1:0.1f}%HR".format(temperature,
humidity))

    else:

        print("Falla de lectura. Reintentando...")

    time.sleep(5)

```

El mismo se encuentra dentro del archivo dht11_sensor_test.py, dentro del directorio dht11_sensor. Si se desea probarlo, copiar el script dentro de la Raspberry Pi que posea el sensor conectado y simplemente correr el mismo dentro del directorio donde se encuentre guardado mediante el comando:

```
python3 dht11_sensor_test.py
```

Preparativos para el cluster con K3s

20 de noviembre:

Enabling cgroups for Raspbian Buster
<https://rancher.com/docs/k3s/latest/en/installation/installation-requirements/>

Enabling cgroups for Raspbian Buster & Enabling legacy iptables on Raspbian Buster:
<https://rancher.com/docs/k3s/latest/en/advanced/>

K3s necesita tener activado algo llamado cgroup para poder ejecutar los contenedores. Raspbian buster tiene dicha característica desactivada por defecto. Para habilitarla:

modificar el archivo /boot/cmdline.txt

sudo nano /boot/cmdline.txt

agregar al final

cgroup_memory=1 cgroup_enable=memory

Para operar, todo debe realizarse siendo root desde este punto

sudo su -

K3s necesita además utilizar algo llamado Legacy IP Tables. Raspbian buster utiliza por defecto nftables, por lo que debemos configurarlo para utilizar legacy iptables

sudo iptables -F

sudo update-alternatives --set iptables /usr/sbin/iptables-legacy

sudo update-alternatives --set ip6tables /usr/sbin/ip6tables-legacy

reiniciamos el sistema

sudo reboot

Instalación de K3s en el nodo maestro

21 de noviembre:

Una vez realizados los preparativos, instalamos K3s en modo superusuario

Tener en cuenta que K3s usa por default a containerd como el Container Runtime. A fin de simplificar la guía, pasaremos a utilizar containerd.

el siguiente comando se encarga de descargar un script provisto por Rancher (los creadores de K3s), el cual se encargará de realizar la instalación

A su vez, rancher recomienda una herramienta Rancher, que facilita el manejo de distintas plataformas de Kubernetes. Aunque no es necesaria para nuestro caso, dejaremos la instalación de K3s lista en caso de que decidamos probarla, ya que según rancher, no causará problemas en caso de dejarse activada.

<https://rancher.com/docs/k3s/latest/en/quick-start/>

<https://rancher.com/docs/rancher/v2.5/en/cluster-provisioning/registered-clusters/>

sudo su -

```
curl -sfL https://get.k3s.io | sh -s - --write-kubeconfig-mode 644
```

Una vez finalizada la instalación, debido a que la capacidad de las SBC es bastante limitada y que un orquestador de Kubernetes es algo pesado, es recomendable volver a reiniciar el sistema para finalizar cualquier proceso que no sea necesario.

A diferencia de antes, esperar algunos minutos hasta que la actividad del cpu baje antes de operar sobre el sistema. (se recomienda utilizar htop para monitorear la actividad)

Luego de unos minutos, comprobamos el estado del sistema utilizando haciendo que liste los nodos que tiene conectados. En este caso, solo se listará a sí mismo, ya que no hay otros nodos.

sudo su -

```
kubectl get nodes
```

Instalación de K3s en nodos worker

21 de noviembre:

Es necesario realizar los mismos preparativos que para el nodo maestro, pero la instalación cambia un poco.

Para instalar K3s en un nodo trabajador, debemos tener la dirección ip y el token de autenticación del nodo maestro.

La dirección ip ya la conocemos por haberla seteado antes. de todas formas, si no se la conoce, puede encontrarse utilizando el comando

```
ipconfig -a | grep -A 7 "eht0"
```

Utilizamos eth0 porque es la interfaz que configuramos inicialmente

El Token de autenticación puede encontrarse en el archivo /var/lib/rancher/k3s/server/node-token, por lo que simplemente hacemos un cat para verlo y copiarlo.

```
cat /var/lib/rancher/k3s/server/node-token
```

Teniendo la dirección ip y el token, iniciamos una sesión mediante ssh con uno de los nodos trabajadores. En la instalación, seteamos dichos parámetros mediante las variables de entorno K3S_URL y K3S_TOKEN Es importante mencionar que para que K3s funcione, cada nodo debe tener un hostname diferente. En nuestro caso, todos los nodos tienen por nombre 'raspberry', por lo que para cambiarlo, editamos también la variable de ambiente correspondiente al nombre del nodo, K3S_NODE_NAME.

```
sudo su -
```

```
curl -sfL https://get.k3s.io | K3S_URL=https://[ip_del_nodo_maestro]:6443  
K3S_TOKEN=[token_del_nodo_maestro]  
K3S_NODE_NAME="[nombre_del_nuevo_nodo]" sh -
```

En nuestro caso, utilizamos la configuración:

```
sudo su
```

```
curl -sfL https://get.k3s.io | K3S_URL=https://192.168.0.201:6443  
K3S_TOKEN=K1001dec79df9dfa648fe3633367dc8cdfd182bb7260444e40ebafa9fbef595032  
0::server:a2aa8e621c89ee0bbd309b3d11c191d0 K3S_NODE_NAME="worker1" sh -
```

Corriendo el script en un contenedor

28 de noviembre:

Para utilizar dicho script con docker, primero debemos crear una imagen. Y antes de ello, debemos crear un dockerfile para especificar cómo crearla.

También hay que tener en cuenta que, como el script debe correrse en una raspberry pi, y la imagen va a tener un SO para una arquitectura ARM como es la de nuestra SBC, lo más eficiente es crear directamente la imagen desde la raspberry pi.

Dockerfile Reference: <https://docs.docker.com/engine/reference/builder/>

Utilizamos nano para poder crear y editar el archivo.

```
nano dockerfile
```

Y en el mismo colocamos.

```
FROM arm32v7/python:3.7.12-buster
```

```
RUN apt-get update && apt-get install -y python3-pip=18.1-5 rpi.gpio=0.6.5-1
```

```
RUN python3 -m pip install --upgrade pip setuptools wheel
```

```
RUN pip3 install Adafruit_DHT==1.4.0
```

```
COPY dht11_sensor_test.py ./
```

```
CMD [ "python3", "DHT11_sensor_test.py" ]
```

Corriendo pruebas, nos dimos cuenta que, para funcionar en contenedores de forma adecuada, debemos hacerle una pequeña modificación a nuestro script. El mismo funciona, sin embargo, dentro del contenedor, los mensajes de print no se muestran hasta detener el script. Esto se debe a cómo maneja python los buffers de salida. Por lo tanto, debemos agregar el parámetro "flush=True" al print, para que funcione de forma correcta.

El script quedaría así:

```
import Adafruit_DHT
import time

DATA_PIN = 3
SENSOR = Adafruit_DHT.DHT11

for i in range(0, 10):

    humidity, temperature = Adafruit_DHT.read_retry(SENSOR, DATA_PIN)

    if ((temperature != None) and ((humidity != None)):

        print("Temperatura={0:0.1f}°C | Humedad={1:0.1f}%HR".format(temperature,
humidity, flush=True))

    else:

        print("Falla de lectura. Reintentando...", flush=True)

    time.sleep(5)
```

Una vez creado nuestro dockerfile y modificado el script, podemos crear la imagen.

```
docker build -t rpi_dht11_sensor:v1 .
```

Creada la imagen, procedemos a probarla dentro de un contenedor.

```
docker container run --rm --privileged rpi_dht11_sensor:v1
```

Nótese el parámetro `--privileged` para que el contenedor pueda acceder a los dispositivos de la raspberry pi. Esto es necesario para poder utilizar los pines GPIO.

Existen dos formas básicamente de lograr esto. Una es eligiendo el dispositivo, reemplazando la sentencia `--privileged` por `--devices /dev/gpiomem`. Esta última forma es la más recomendada para estos casos, ya que simplemente comparte con el contenedor el dispositivo físico del host. Sin embargo, las limitaciones de la imagen base impidieron que la imagen final pueda acceder al archivo correspondiente a dicho dispositivo. `/dev/gpiomem`, incluso habiendo modificado los grupos del sistema y de la imagen como es debido. Concluimos que el problema puede deberse a que no utilizamos una imagen base con la arquitectura provista por raspbian, sino una basada en Debian. El método de `--privileged` dió buenos resultados, funcionando de forma automática. De todas formas, no es recomendable utilizar este parámetro en un contenedor de forma general. Esto suele ser una técnica insegura y una práctica desaconsejada, ya que el contenedor correrá más cerca del sistema operativo del host, perdiendo ciertas ventajas de seguridad propias de los contenedores. En nuestro caso, dado que nos encontramos en un ambiente controlado y cerrado, podemos darnos el lujo de utilizar el parámetro `--privileged`.

Docker Privileged: <https://phoenixnap.com/kb/docker-privileged>

Página web del sensor con Flask version 2.0.2

4 de Diciembre:

Teniendo nuestro sensor funcionando, procedemos a crear una página web que muestre los datos del sensor. Para ello, utilizaremos la librería Flask.

Fask quick start: <https://flask.palletsprojects.com/en/2.0.x/quickstart/>

Flask Tutorial: <https://pythonbasics.org/flask-tutorial-hello-world/>

```
pip3 install Flask
```

Luego, aplicamos algunas modificaciones a nuestro script para que sea ejecutable como una aplicación web.

```
nano dht11_sensor_test.py

# import the Flask class from the flask module
from flask import Flask, render_template
import Adafruit_DHT

DATA_PIN = 3
SENSOR = Adafruit_DHT.DHT11

# create the application object
app = Flask(__name__)

# use decorators to link the function to a url
@app.route('/')
def home():
    humidity, temperature = Adafruit_DHT.read_retry(SENSOR, DATA_PIN)

    if ((temperature != None) and ((humidity != None))):
        return("Temperatura={0:0.1f}°C | Humedad={1:0.1f}%HR".format(temperature,
humidity))
    else:
        return("Falla de lectura. Reintentando...")

# start the server with the 'run()' method
if __name__ == '__main__':
```

```
port = int(os.environ.get("PORT", 5000))
app.run(debug=True, host='0.0.0.0', port=port)
```

Luego simplemente corremos el script.

```
python3 dht11_sensor_test.py
```

Nos dirigimos a un navegador web y colocamos la dirección de nuestra página web.

```
[ip_nodo_host]:5000
```

Servicio Flask en contenedores

5 de Diciembre:

Metemos nuestro nuevo script en una imagen de docker. Para ello, editamos un dockerfile.

```
nano dockerfile
FROM arm32v7/python:3.7.12-buster

RUN apt-get update && apt-get install -y python3-pip=18.1-5 rpi.gpio=0.6.5-1

RUN python3 -m pip install --upgrade pip setuptools wheel

RUN pip3 install Adafruit_DHT==1.4.0 Flask==2.0.2

COPY flask_dht11_sensor_test.py .

CMD [ "python3", "flask_dht11_sensor_test.py" ]
```

Luego, creamos la imagen.

```
docker build -t flask_dht11:v1 .
```

Una vez terminada, corremos un contenedor en docker para probarla. En este caso, cambiaremos

```
docker container run --rm --privileged -p 4000:5000 flask_dht11:v1
```

Subiendo la imagen a docker hub

5 de Diciembre:

Docker hub: <https://hub.docker.com/>

Docker hub quickstart: <https://docs.docker.com/docker-hub/>

Docker Repositories: <https://docs.docker.com/docker-hub/repos/>

El tener la imagen creada por nosotros subida a un repositorio en docker hub, permite facilitar el proceso de deployment en otras plataformas, incluyendo el cluster en sí.

Para ello, uno debe tener una cuenta en docker hub. Una vez creada, seleccionamos "Create Reopistory" y le damos un nombre. En nuestro caso, llamaremos a nuestro repositorio "flask-dht11-rpi".

A continuación, cambiamos el tag de la imagen creada por uno que posea todas las características estándar para el repositorio.

```
docker      image      tag      imagen_fuente:tag_fuente
nombre_de_usuario/nombre_repositorio:tag_destino
```

El tag puede ser el mismo que el tag de la imagen original, o puede ser uno nuevo. Por ejemplo, en nuestro caso, el comando quedaría

```
docker image tag flask_dht11:v1 alcaolpg/flask-dht11-rpi:latest
```

Luego, iniciamos sesión y subimos el archivo a docker hub.

```
docker login  
docker push nombre_de_usuario/nombre_repositorio:tag_destino
```

Al finalizar la carga, cerramos sesión.

```
docker logout  
Imagen subida a docker hub:  
https://hub.docker.com/repository/docker/alcaolpg/flask-dht11-rpi
```

Deployment en el cluster de kubernetes

6 de Diciembre:

Al momento de lanzar la aplicación en el cluster, es importante notar que existe un tipo de archivo de vital importancia para el proceso. Los archivos .yaml (Yet Another Markup Language o YAML Ain't Markup Language) son una forma de definir los servicios y los contenedores que se encuentran en el cluster de forma rápida y entendible.

What is YAML?: <https://www.redhat.com/en/topics/automation/what-is-yaml>

Understanding	Kubernetes	Objects:
		https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/

En nuestro caso, debemos crear un archivo de configuración de contenedores y otro de servicios. El de contenedores se encargará de crear las pods que contienen nuestra aplicación, con todas sus características necesarias. El de servicios se encargará de definir un servicio que permita acceder a la aplicación desde el exterior. Al igual que con docker, es necesario que la cápsula que contenga la aplicación funcione en modo privilegiado.

Pod	Security	Polices:
		https://kubernetes.io/docs/concepts/policy/pod-security-policy/#privileged

Kubernetes	Privileged	Pod	Practical	Example:
				https://www.golinuxcloud.com/kubernetes-privileged-pod-examples/

nano sensor.yaml

Dentro del mismo, escribimos:

```
apiVersion: apps/v1 # Versión de Kubernetes para este objeto
kind: Deployment # Tipo de objeto
metadata:
  name: sensor # Nombre del objeto
  namespace: default # Nombre del espacio de trabajo
spec:
  replicas: 1 # Cantidad de pods que se crearan
  selector:
    matchLabels:
      app: sensor # Nombre del contenedor para identificación de la aplicación
  template:
    metadata:
      labels:
        app: sensor # Nombre del contenedor
    spec:
      containers:
        - name: flask-dht11-rpi # Nombre de la aplicación
          image: alcaolpg/flask-dht11-rpi:latest # Nombre de la imagen que debe buscarse en
          docker hub
      ports:
        - containerPort: 5000 # Puerto utilizado para comunicarse con la aplicación
          protocol: TCP
      securityContext:
```

```
privileged: true # Permite que el contenedor acceda a los dispositivos GPIO  
nodeSelector:
```

```
dhtsensor: "yes" # Selecciona los nodos que poseen el dispositivo DHT11
```

Parémonos en la última sentencia de la sección "nodeSelector". En nuestro caso, el sensor se encuentra conectado a una única placa Raspberry Pi 3B+ mediante GPIO. Por lo tanto, si queremos que el sistema sea capaz de obtener información del sensor, debemos especificar que el nodo debe tener el dispositivo DHT11. Para ello, nos valemos de la posibilidad de usar etiquetas o labels para cada nodo.

Assigning pods to nodes:
<https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/>

How to use Node Selectors in Kubernetes:
<https://www.howtoforge.com/use-node-selectors-in-kubernetes/>

Desde el nodo maestro, agregaremos una etiqueta para que el nodo seleccionado por el sistema sea el correcto.

kubectl label node [nombre del nodo con el sensor] nombre_del_campo_label=label_a_usar
En nuestro caso, la sentencia anterior quedaría:

```
kubectl label node worker1 dhtsensor=yes
```

Para comprobar la etiqueta, podemos usar el comando:

```
kubectl get nodes --show-labels
```

Creada nuestra etiqueta, procedemos a lanzar la aplicación en el cluster.

```
kubectl apply -f sensor.yaml
```

Podemos ver los resultados en la consola de kubernetes.

```
kubectl get pods -o wide
```

Para poder acceder a la aplicación, todavía debemos crear el servicio que se encargue de redireccionar las solicitudes del exterior al contenedor de la aplicación. Este tipo de servicio se llama "nodeport".

```
nano sensor_nodeport.yaml
```

```
apiVersion: v1
kind: Service
metadata:
  name: sensor-nodeport # Nombre del servicio
  namespace: default
spec:
  type: NodePort
  selector:
    app: sensor # Nombre del contenedor que identifica a la aplicación
  ports:
    - port: 5000 # Puerto utilizado para comunicarse con la aplicación
      targetPort: 5000 # Puerto del contenedor de la aplicación
      nodePort: 30001 # Puerto utilizado para acceder al contenedor desde el exterior
```

Una vez que creamos el servicio, podemos lanzarlo en el cluster.

```
kubectl apply -f sensor_nodeport.yaml
```

Podemos comprobar que el servicio esté corriendo utilizando:

```
kubectl get services
```

Por último, accedemos desde el exterior al contenedor de la aplicación. Desde una computadora dentro de la misma red, accedemos utilizando la dirección ip de cualquier nodo del cluster desde un navegador web.

[http://\[ip del nodo\]:30001](http://[ip del nodo]:30001)

En caso de querer eliminar el servicio o el contenedor, podemos hacerlo con los comandos:

kubectl delete pod [nombre del contenedor]

kubectl delete deployment [nombre del contenedor]

kubectl delete service [nombre del servicio]

Removing data collector Docker container / Kubernetes pod:
<https://www.ibm.com/docs/en/cloud-paks/cp-management/2.3.x?topic=monitoring-removing-data-collector-docker-container-kubernetes-pod>

6. Propuestas de Proyectos a Futuro

Como proyectos a futuro, hay tres puntos sobre los que sería deseable continuar con la investigación y el desarrollo del sistema:

- Control del clúster y otras configuraciones de archivos YAML: La cantidad de funcionalidades provistas por la herramienta kubectl y la diversa cantidad de combinaciones de opciones que pueden llevarse a cabo con los archivos de configuración YAML, son tan extensos que bien podrían resultar en una investigación particular de distintos casos de aplicación de los mismos.
- Monitoreo y control del clúster mediante Rancher: Rancher es una herramienta que cuenta con una GUI mediante la cual puede monitorearse en tiempo real el estado del clúster y realizar despliegues tanto de contenedores como de servicios de manera remota y cómoda. La misma es desarrollada por los creadores de K3S.
- Alternativa al uso de GPIO en raspberry pi y contenedores: Ya se habló respecto a las desventajas de correr contenedores en modo privilegiado. Investigar una alternativa a dicha problemática sería algo interesante.

7. Bibliografía

Getting started with Docker:

<https://docs.docker.com/get-started/>

¿Qué es Kubernetes?:

<https://tanzu.vmware.com/developer/guides/kubernetes/what-is-kubernetes/>

Introducción a Docker Containers en Kubernetes:

<https://tanzu.vmware.com/developer/guides/kubernetes/what-is-kubernetes/>

Sistemas de alta disponibilidad:

<https://us.sios.com/what-we-do/high-availability/>

<https://medium.com/velotio-perspectives/demystifying-high-availability-in-kubernetes-using-kubeadm-3d83ed8c458b>

Docker tutorial:

<https://docs.docker.com/get-started>

Guía de documentación Raspberry Pi:

<https://www.raspberrypi.com/documentation/computers/os.html#vcgencmd>

Sensor DHT11:

<https://www.mouser.com/datasheet/2/758/DHT11-Technical-Data-Sheet-Translated-Version-1143054.pdf>

<http://tutorialesdeelectronicabasica.blogspot.com/2021/03/que-es-el-condensador-de.html>

<https://learn.adafruit.com/dht-humidity-sensing-on-raspberry-pi-with-gdocs-logging/python-setup>

Instalación de K3S:

<https://rancher.com/docs/k3s/latest/en/installation/installation-requirements/>
<https://rancher.com/docs/k3s/latest/en/advanced/>
<https://rancher.com/docs/k3s/latest/en/quick-start/>
<https://rancher.com/docs/rancher/v2.5/en/cluster-provisioning/registered-clusters/>

Creación de imágenes con docker:

<https://docs.docker.com/engine/reference/builder/>
<https://phoenixnap.com/kb/docker-privileged>
<https://hub.docker.com/>
<https://docs.docker.com/docker-hub/>
<https://docs.docker.com/docker-hub/repos/>

Flask:

<https://flask.palletsprojects.com/en/2.0.x/quickstart/>
<https://pythonbasics.org/flask-tutorial-hello-world/>

YAML, archivos de configuración y herramientas de Kubernetes:

<https://www.redhat.com/en/topics/automation/what-is-yaml>
<https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/>
<https://kubernetes.io/docs/concepts/policy/pod-security-policy/#privileged>
<https://www.golinuxcloud.com/kubernetes-privileged-pod-examples/>
<https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/>
<https://www.howtoforge.com/use-node-selectors-in-kubernetes/>
<https://www.ibm.com/docs/en/cloud-paks/cp-management/2.3.x?topic=monitoring-removing-data-collector-docker-container-kubernetes-pod>

8. Anexo

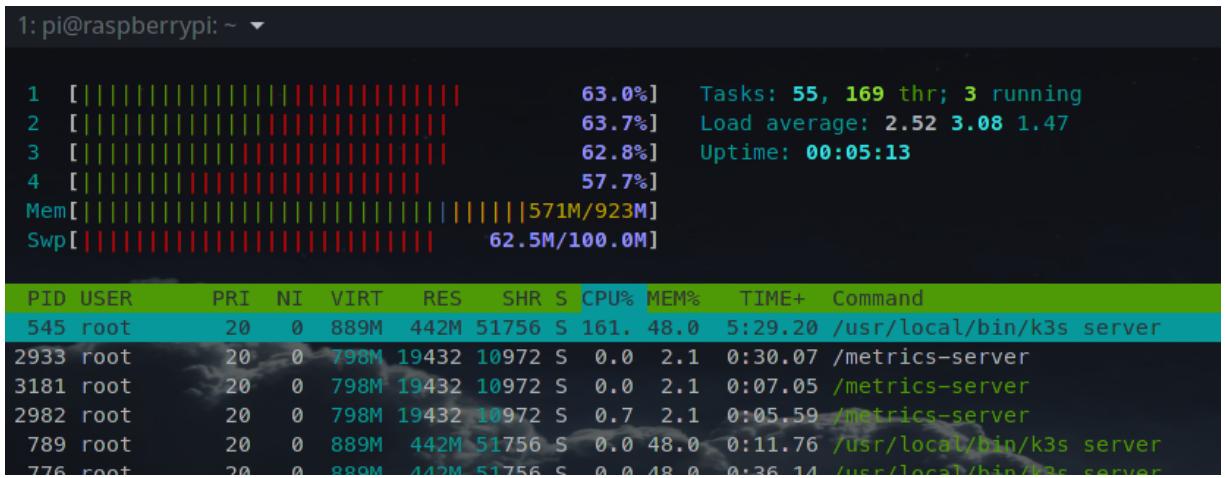


Fig. 4.1: Estado del nodo maestro al arrancar

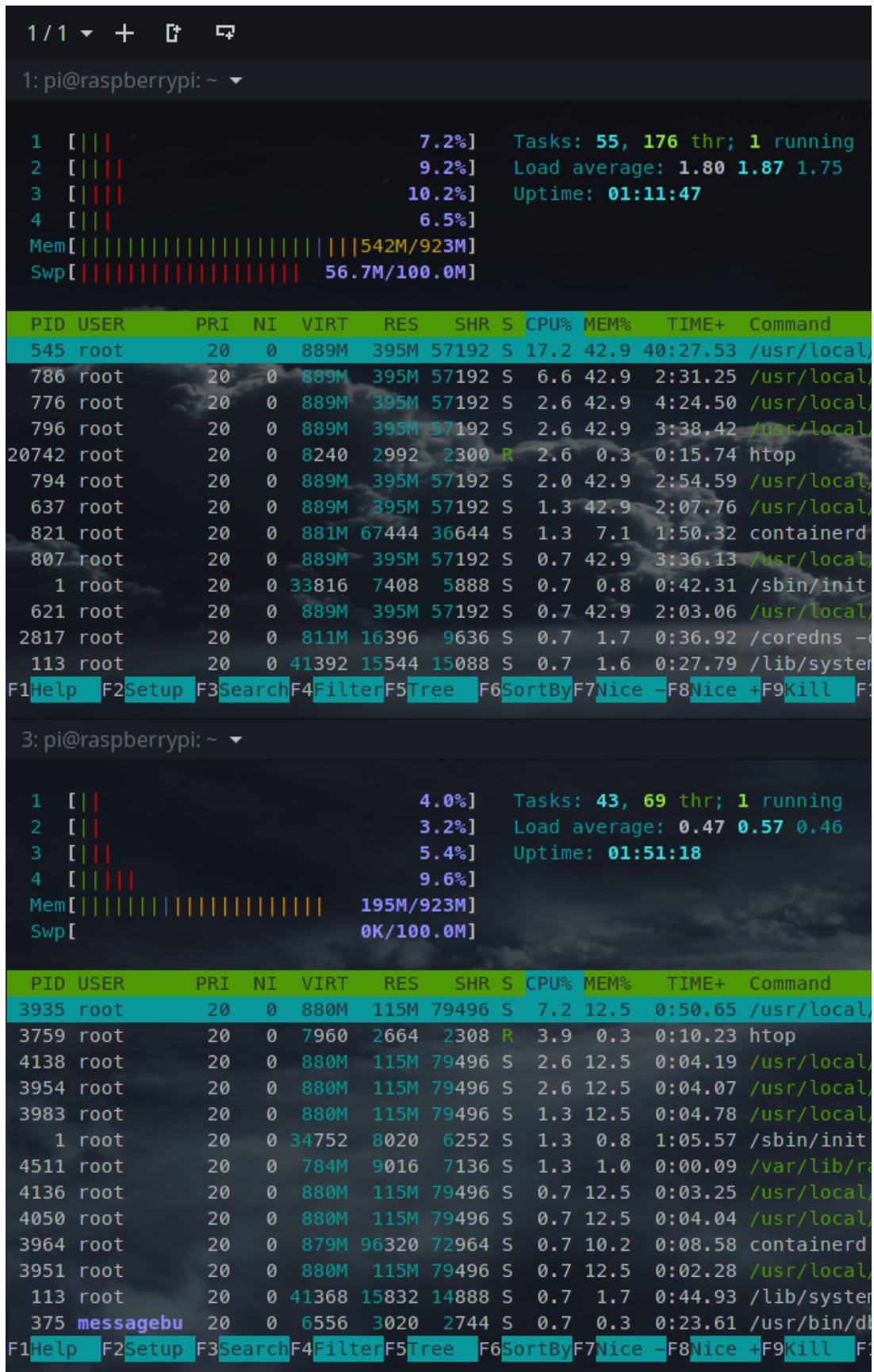


Fig. 4.2: Comparativa entre la carga del nodo maestro (arriba) y el nodo trabajador (abajo)