Diseño e implementación de una microCloud abierta para IoT

José Elías Rael Gutierrez
Grado de Ingeniería Informática

Félix Freitag

Junio 2018

Esta obra está sujeta a una licencia de Reconocimiento-NoComercial-SinObraDerivada 3.0 España de Creative Commons

A mi familia,

FICHA DEL TRABAJO FINAL

Título del trabajo:	Diseño e implementación de una microCloud abierta para IoT
Nombre del autor:	José Elías Rael Gutierrez
Nombre del consultor:	Félix Freitag
Fecha de entrega (mm/aaaa):	06/2018
Área del Trabajo Final:	Aplicaciones y Sistemas distribuidos
Titulación:	*Grado de Ingeniería Informática*

Resumen del Trabajo (máximo 250 palabras):

Últimamente hemos podido comprobar la aparición en el mercado de dispositivos de bajo costo que permiten ofrecer servicio de hogar inteligente en redes Wifi existentes. Estos dispositivos no necesitan servidor local y se controlan mediante una sencilla interfaz de usuario en una aplicación de móvil.

Sin embargo, estas soluciones baratas están diseñadas para ser gestionadas mediante servicios en la nube proporcionados por el propio fabricante. Este enfoque tiene varias deficiencias e inconvenientes, siendo el más importante e inaceptable que el sistema de control no funciona si el servicio del fabricante no está disponible a través de una conexión de Internet.

Comprobaremos que gracias al movimiento Open Source, podemos actualmente diseñar y crear una microCloud genérica en nuestra red local, con protocolos, software y hardware abierto que no solo suplirá las funcionalidades ofrecidas por el fabricante, sino que las mejorará ampliamente y nos dará la capacidad de adaptar todo el sistema a nuestras necesidades.

Reprogramaremos estos dispositivos de bajo coste para desconectarlos de la Cloud del fabricante y conectarlos de forma nativa a nuestra microCloud para crear un nuevo sistema abierto constituido por nodos inalámbricos programables en distintos entornos de desarrollo libres.

Una vez que consigamos un sistema funcional, exploraremos la implementación de características inéditas en instalaciones caseras de bajo coste, tales como alta disponibilidad y soluciones distribuidas.

Finalmente, ampliaremos nuestra perspectiva averiguando como compartir y extender nuestra microCloud en redes colaborativas abiertas o como añadir capacidades de recopilación y análisis de datos a nuestro sistema.

Abstract (in English, 250 words or less):

Recently we have witnessed the appearance in the market of low cost devices that are able to offer smart home services in existing Wi-Fi networks. These devices do not need a local server and are controlled by a simple mobile user interface application.

However, these cheap solutions are designed to be managed through proprietary cloud services provided by the manufacturer. This approach has several shortcomings and drawbacks, the most important and unacceptable being that the control system does not work if the manufacturer's service is not reachable through Internet.

Thanks to the Open Source movement, we will be able to design and create a generic microCloud in our local network, with open protocols, software and hardware that will not only supply the features offered by the manufacturer's system, but also will greatly improve it and give us the ability to adapt the entire system to our needs.

After reprogramming these low-cost devices to disconnect them from the manufacturer's Cloud and connect them natively to our microCloud, we will create a new open system made of a multitude of wireless nodes. These nodes will be programmable in different free development environments.

Once we get a functional system, we will explore the implementation of unprecedented features in low cost home installations, such as high availability and distributed solutions.

Finally, we will expand our perspective by finding out how to share and extend our installation to open collaborative networks or how to add data collection and analysis capabilities to our system.

Palabras clave (entre 4 y 8):

microCloud, IoT, MQTT, Cloudy, Node-Red, Docker, Swarm, Raspberry

Índice

1 Introducción ... 1
1.1 Contexto y justificación del Trabajo ... 1
1.2 Objetivos del Trabajo ... 2
1.3 Enfoque y método seguido ... 2
1.4 Planificación del Trabajo ... 2
1.4.1 Estructura de desglose de Tareas (EDT) ... 3
1.4.2 Cronograma del proyecto ... 4
1.5 Breve sumario de productos obtenidos ... 5
1.6 Breve descripción de los otros capítulos de la memoria ... 5
2 Primera fase: microCloud local ... 6
2.1 Estado actual ... 6
2.1.1 Modelo de referencia IoT ... 6
2.1.2 Solución de hogar inteligente de Vera Control, Ltd. ... 7
2.1.3 Solución de hogar inteligente de Itead, Ltd. ... 8
2.2 Diseño del Sistema ... 9
2.2.1 Protocolo de comunicación ... 9
2.2.2 Redes ... 10
2.2.3 Hardware ... 11
2.2.3.1 Hardware de la microCloud ... 11
2.2.3.2 Hardware de los nodos IoT ... 12
2.2.4 Software ... 13
2.2.4.1 Sistema Operativo ... 13
2.2.4.2 Gestor de Contenedores ... 14
2.2.4.3 Bróker MQTT ... 15
2.2.4.4 Entorno de programación ... 16
2.2.4.5 Interfaz de usuario ... 16
2.3 Implementación del sistema ... 16
2.3.1 Implementación de las redes ... 16
2.3.2 Hardware de la microCloud ... 17
2.3.3 Instalación del Sistema Operativo Cloudy ... 17
2.3.4 Instalación de Docker ... 19
2.3.5 Instalación del contenedor Mosquitto ... 19
2.3.6 Instalación del Contenedor Node-Red ... 21
2.3.7 Configuración de los nodos IoT ... 22
2.3.8 Configuración de la Interfaz de usuario ... 24
2.4 Integración y Programación ... 25
2.4.1 Publicar servicios en la CWAN ... 25
2.4.2 Publicar contenedores al reiniciar ... 27
2.4.3 Instalar servidor NTP en la LoT ... 27
2.4.4 Actualización automática del firmware ... 28
2.4.5 Activar o Desactivar todos los dispositivos IoT ... 30
2.4.6 Crear una escena de ejemplo ... 31
2.4.7 Medir el consumo de la microCloud ... 31
2.4.8 Añadir botón de ‘mute’ en la interfaz de usuario ... 32
2.4.9 Añadir un sensor de temperatura a nuestro sistema ... 33
2.5 Conclusión de la fase 1 ... 34

3 Segunda fase: microCloud Avanzada 35
3.1 Análisis de la microCloud de la Fase 1 35
3.1.1 Arquitectura Lógica 35
3.1.2 Arquitectura Física 36
3.2 Diseño de una microCloud Tolerante a fallos 37
3.2.1 Redundancia de Hardware y redes 37
3.2.2 Redundancia de Software 38
3.2.2.1 Failover Manual 39
3.2.2.2 IP Failover 39
3.3 Implementación de la microCloud Tolerante a fallos 40
3.3.1 Añadir Wifi de respaldo 40
3.3.2 Creación del segundo nodo Cloudy 41
3.3.2.1 Hardware 41
3.3.2.2 Sistema Operativo y redes 41
3.3.2.3 Software 42
3.3.2.4 Replicación manual de la configuración 42
3.3.3 Implementación del clúster 43
3.3.3.1 Configuración de keepalived 43
3.3.3.2 Configuración de la IP flotante en los clientes 45
3.3.3.3 Publicación de servicios en dirección IP alternativa. 45
3.4 Optimización de recursos del clúster 48
3.4.1 Docker Swarm 48
3.4.1.1 Tipos de Nodos Swarm y alta disponibilidad 49
3.4.2 Creación del tercer nodo Cloudy 50
3.4.2.1 Hardware 50
3.4.2.2 Sistema Operativo y redes 50
3.4.2.3 Software 51
3.4.3 Implementación del clúster Swarm 51
3.4.3.1 Ejecución de servicios 52
3.4.3.2 Publicación de servicios 54
3.4.3.3 Problema de configuración de los servicios Swarm 55
3.4.3.4 Replicación de datos entre Nodos 56
3.4.3.5 Configuración de los servicios de la microCloud 58
3.4.3.6 Integración de Swarm en la interfaz de Cloudy 59
3.5 Extensión de la microCloud 60
3.5.1 Extensión del Swarm 60
3.5.1.1 Docker Swarm con varios propietarios 60
3.5.1.2 Docker Swarm en múltiples ubicaciones 61
3.5.1.3 Conectar Swarm mediante una VPN 61
3.5.2 Programación distribuida con DNR-Editor 65
3.6 Conclusiones de la fase 2 68
3.6.1 Implementación de alta disponibilidad en redes domesticas 68
3.6.2 Replicación de configuración para servicios Swarm 68
3.6.3 Integración de Docker Swarm en la interfaz de Cloudy 68
3.6.4 Extensión de la microCloud 69
4 Tercera fase: Aplicaciones Avanzadas 70
4.1 Monitorización y Análisis 70
4.1.1 Componentes 70
4.1.2 Monitorización de Swarm con TIG 71
4.1.3 Monitorización de la microCloud 72

4.2 Serverless computing con Openfaas ... 74
4.2.1 Instalar OpenFaaS ... 75
4.2.2 Crear una función Serverless de ejemplo ... 76
4.2.3 Crear una aplicación Serverless ... 78
4.3 Conclusiones de la fase 3 ... 79
5 Valoración de la solución desarrollada ... 80
5.1 microCloud en redes domésticas ... 80
5.2 microCloud en redes comunitarias ... 80
6 Conclusiones Finales ... 82
7 Glosario ... 83
8 Bibliografía ... 90
9 Anexos de la Memoria ... 96
ANEXO 1: Configuración de la Red en la Fase 1 ... 97
ANEXO 2: Configuración de la Red en la Fase 2 ... 99
ANEXO 3: Configuración MQTT-Dash ... 101
ANEXO 4: Problema de pérdida de paquetes ... 103
ANEXO 5: Descripción de ficheros adjuntos a la Memoria ... 105
ANEXO 6: Enlaces de Video ... 107

Lista de figuras

Figura 1: Estructura de desglose de tareas 3
Figura 2: Diagrama de Gantt 4
Figura 3: IoT World Forum Reference Model 6
Figura 4: Solución de hogar inteligente de Vera 7
Figura 5: Solución de hogar inteligente de ITEAD 8
Figura 6: Arquitectura genérica MQTT 9
Figura 7: Diseño de la red 10
Figura 8: Configuración de Redes 11
Figura 9: Direccionamiento de los equipos 11
Figura 10: Raspberry pi 3 11
Figura 11: SoC ESP8266 12
Figura 12: Costes de dispositivos IoT en amazon.es 13
Figura 13: Arquitectura de Cloudy 14
Figura 14: Containers vs Virtual Machines 15
Figura 15: Raspberry pi 3 con caja apilable y cargador de alta potencia 17
Figura 16: Login de Cloudy en ‘SmartHome1’ 18
Figura 17: Comprobación de Servicio Docker en ‘SmartHome1’ 19
Figura 18: Comprobación de contenedores 20
Figura 19: Pantalla de MQTT.fx 21
Figura 20: Comprobación de Node-Red 22
Figura 21: Adaptador USB a TTL 23
Figura 22: Entorno Deviot 23
Figura 23: Pantalla de configuración de un dispositivo 24
Figura 24: Servicios encontrados desde ‘Cloudy-Ext1’ 26
Figura 25: Flujo ‘set sonoffs’ 28
Figura 26: Configuración WebDAV 30
Figura 27: Flujo de ‘Escena Estudio’ 31
Figura 28: Interfaz web de Power1 32
Figura 29: Flujo ‘Consumo microCloud’ 32
Figura 30: Flujo ‘TV Mute’ 33
Figura 31: Sensor de temperatura 33
Figura 32: Flujo ‘Temperatura’ 34
Figura 33: Arquitectura lógica de la microCloud 35
Figura 34: Arquitectura física de la microCloud de la fase 1 36
Figura 35: MicroCloud completamente redundada 37
Figura 36: Solución de compromiso de microCloud redundada 37
Figura 37: Direccionamiento de los equipos en la Fase 2 38
Figura 38: Nodos independientes Docker 38
Figura 39: IP failover en clúster de 2 nodos 39
Figura 40: Pila de 2 Raspberry pi 41
Figura 41: Flujo 'Test Health' 44
Figura 42: Publicación IP flotante en Safe@Office 47
Figura 43: Reglas para publicar servicios redundados 47
Figura 44: Configuración de equipos de red de la Fase 2 47
Figura 45: Balanceo de carga interno y Routing Mesh 48
Figura 46: Ejemplo de clúster Swarm con 3 managers 49

Figura 47: Pila de 3 Raspberry pi 50
Figura 48: IP failover en clúster de 3 nodos 51
Figura 49: Direcciones IP de los nodos del Clúster Swarm 51
Figura 50: Nodos en Clúster con Swarm 53
Figura 51: Malla completamente conectada 56
Figura 52: Configuración de carpeta docker-config 58
Figura 53: Puertos y Protocolos de Swarm 61
Figura 54: VPN Swarmnet 62
Figura 55: Instalación de DNR-Editor 65
Figura 56: Arquitectura DNR-Editor 65
Figura 57: Interfaz web de DNR-Editor 66
Figura 58: Devices en DNR-Editor 67
Figura 59: Flujo distribuido en DNR-Editor 67
Figura 60: Flujo en Cloudy-Ext2 67
Figura 61: Flujo en microCloud 67
Figura 62: Mensajes recibidos en microCloud 67
Figura 63: Arquitectura de monitorización 70
Figura 64: Swarm Dashboard 72
Figura 65: Crear BD influxDB 72
Figura 66: Estadística de mensajes por minuto 73
Figura 67: Estadística de Consumo de Sonoff 73
Figura 68: Estadística de Pings 73
Figura 69: microCloud Dashboard 74
Figura 70: Portal OpenFaaS 75
Figura 71: Flujo de prueba de OpenFaaS 75
Figura 72: OpenFaaS Serverless Dashboard 76
Figura 73: Despliego de función jp2a 77
Figura 74: Resultado de la función jp2a 78
Figura 75: Flujo de prueba OpenFaaS con jp2a 78

1 Introducción

En este trabajo diseñaremos e implementaremos con componentes abiertos un sistema de hogar inteligente que incluye una microCloud local genérica, ampliable, flexible y programable que es capaz de proporcionar servicios en alta disponibilidad no solo en redes locales sino también en redes comunitarias.

Construiremos nuestro sistema con dispositivos IoT reprogramados de bajo coste y para nuestra microCloud nos basaremos en ordenadores de tarjeta única tipo Raspberry pi, capaces de ejecutar un sistema operativo Linux y un gestor de contenedores que nos permita publicar microservicios.

Todos los componentes del sistema, hardware, software y protocolos serán abiertos o serán compatibles con tecnologías abiertas.

1.1 Contexto y justificación del Trabajo

En los años 80, Richard Stallman pidió a Xerox acceso al código fuente del driver de su impresora para añadirle funcionalidades que permitiesen utilizarlo más fácilmente en su entorno de trabajo. La negativa de Xerox enfureció a Stallman llevándole a lanzar el proyecto GNU para crear un sistema operativo libre **[1]**. En 1991 Linus Torvalds liberó el núcleo de Linux, completando de esta manera el sistema operativo GNU/Linux. Con el tiempo, Linux ha llegado a convertirse en el sistema operativo más usado en servidores de Internet y actualmente está extendiéndose cada vez más al mundo empresarial **[2]**.

Simultáneamente, hemos sido también testigos de la evolución del hardware abierto, desde la aparición de sistemas orientados a la docencia y la experimentación como Arduino hasta finalmente tener a disposición ordenadores de tarjeta única (SBC o Single Board Computers) **[3]**, con la potencia suficiente para ejecutar distribuciones Linux completas.

El bajo coste, bajo consumo y la gran potencia de estos últimos SBCs permiten ya ejecutar tecnologías de virtualización a nivel de sistema operativo tales como Docker **[4]**, que pueden hacer llegar a los hogares la revolución en flexibilidad y alta disponibilidad que la virtualización basada en hipervisor aportó al mundo empresarial durante estas tres últimas décadas.

Recientemente han aparecido dispositivos IoT de bajo coste que ofrecen control domótico en redes Wifi existentes. Estos dispositivos no necesitan servidor local y se controlan mediante una sencilla aplicación de móvil a través de los servidores Cloud del fabricante. Son dispositivos completamente dependientes del fabricante de los cuales no tenemos ningún tipo de control.

En este trabajo comprobaremos como el movimiento Open Source nos sigue proporcionando herramientas, protocolos, hardware y software para crear sistemas abiertos que no solo nos permiten superar las limitaciones que nos son impuestas por los fabricantes sino que nos permiten añadir nuevas funcionalidades. En resumen, nos ofrece la libertad de usar nuestros dispositivos como y cuando queramos.

1.2 Objetivos del Trabajo

Los objetivos generales de este trabajo son:

- Investigar los distintos componentes de software, Hardware y protocolos necesarios para la creación de una infraestructura IoT.
- Diseñar e Implementar soluciones distribuidas y/o de alta disponibilidad en entornos domésticos con hardware y software abierto de bajo coste.
- Conocer y elegir los componentes abiertos disponibles para alcanzar estos objetivos y demostrar la vigencia del movimiento Open Source para darnos control sobre nuestra información y nuestros dispositivos.

Los objetivos específicos serán los siguientes:

- Crear un sistema de control domótico de bajo coste que con componentes abiertos y dispositivos IoT Wifi reprogramados, supere en funcionalidad las soluciones existentes en el mercado.
- Crear una microCloud genérica y versátil que permita a los usuarios ejecutar servicios en alta disponibilidad en redes locales y comunitarias.
- Contribuir al desarrollo del proyecto open source Cloudy **[5]** estudiando ampliaciones, nuevas aplicaciones y funcionalidades avanzadas.

1.3 Enfoque y método seguido

Este será un trabajo que pretende abordar la creación desde cero de un sistema completo integrando componentes abiertos mediante los diversos conocimientos adquiridos en la carrera de ingeniería informática.

Por esa razón, en lugar de buscar obtener un único producto al final de nuestro proyecto, he preferido usar un enfoque más práctico en el que intentaré obtener un producto básico en la fase inicial del proyecto al que iremos añadiendo funcionalidades, aplicaciones y complejidad de manera incremental.

De esta manera, en una primera fase crearemos una microCloud sencilla pero funcional. En una segunda fase, sobre esta base, investigaremos como implementar alta disponibilidad y cómo extender la microCloud a otras ubicaciones. En la tercera y última fase implementaremos aplicaciones avanzadas sobre la infraestructura desarrollada en las fases anteriores.

1.4 Planificación del Trabajo

Este proyecto se implementará en las tres fases descritas en el apartado anterior. Para definir su alcance, crearemos una EDT (Estructura de Desglose de Tareas) donde detallaremos todas las tareas necesarias para completar el proyecto. Efectuaremos la estimación de duración de tareas en la misma EDT, y a partir de ahí desarrollaremos el cronograma del proyecto.

1.4.1 Estructura de desglose de Tareas (EDT)

En la metodología de gestión de proyectos PMP, el propósito de la EDT es definir exhaustivamente el alcance del proyecto. Esta EDT o WBS **[6]** refleja todas las tareas que van a ser incluidos en nuestro proyecto. Una vez definido el alcance, podremos proceder a estimar la cantidad de tiempo necesaria.

Figura 1: Estructura de desglose de tareas

		Nivel 3	Nivel 4	Horas	%Proy
1. TFC	1.1	1.1.1 Planificación de Alcance		20	6,67
		1.1.2 Planificación de Tiempo		20	6,67
	Fase 1	1.2.1 Investigación y diseño	1.2.1.1 Estado actual	10	3,33
			1.2.1.2 Diseño del Sistema	20	6,67
		1.2.2 Implementación de la Red	1.2.2.1 Configurar e Integrar la LoT	5	1,67
			1.2.2.2 Implementación de la LoT	2	0,67
		1.2.3 Implementación de HW	1.2.3.1 Implementación de los Nodos IoT	6	2,00
			1.2.3.2 Implementación HW Servidor	2	0,67
		1.2.4 Configuración de S.O.	1.2.4.1 Instalar y configurar S.O.	4	1,33
			1.2.4.2 Instalar y configurar Docker	4	1,33
		1.2.5 Configuración de contenedores	1.2.5.1 Instalar y configurar Bróker MQTT	4	1,33
			1.2.5.2 Instalar y configurar Node Red	4	1,33
		1.2.6 Implementación de la UI	1.2.6.1 Instalar y configurar App Android	6	2,00
			1.2.6.2 Instalar y configurar Dashboard	5	1,67
		1.2.7 Programación del sistema	1.2.7.1 Análisis de necesidades	5	1,67
			1.2.7.2 Programación en Node Red	15	5,00
		1.2.8 Integración y pruebas	1.2.8.1 Integración de la microCloud	2	0,67
			1.2.8.2 Pruebas y conclusiones	10	3,33
	Fase 2	1.3.1 Investigación y diseño	1.3.1.1 Estado del Arte de HA	10	3,33
			1.3.1.2 Estudio Docker Swarm	20	6,67
			1.3.1.3 Estudio de HA de Cloudy	20	6,67
			1.3.1.4 Diseño del Sistema en HA	10	3,33
		1.3.2 Implementación clúster	1.3.2.1 Instalar y configurar HW clúster	4	1,33
			1.3.2.2 Instalar y configurar SW clúster	4	1,33
		1.3.3 Implementación de HA	1.3.3.1 Implementación de Docker Swarm	10	3,33
			1.3.3.2 Implementación de HA en nodos IoT	10	3,33
			1.3.3.3 Implementación de HA con Cloudy	10	3,33
		1.3.4 Integración y prueba de HA	1.3.4.1 Integración de HA	20	6,67
			1.3.4.2 Prueba del Sistema en HA	10	3,33
	Fase 3	1.4.1 Publicar datos microCloud		4	1,33
		1.4.2 Extender la microCloud		4	1,33
		1.4.3 Añadir análisis de datos		10	3,33
	1.5	Presentaciones		10	3,33
			TOTALES:	300	100

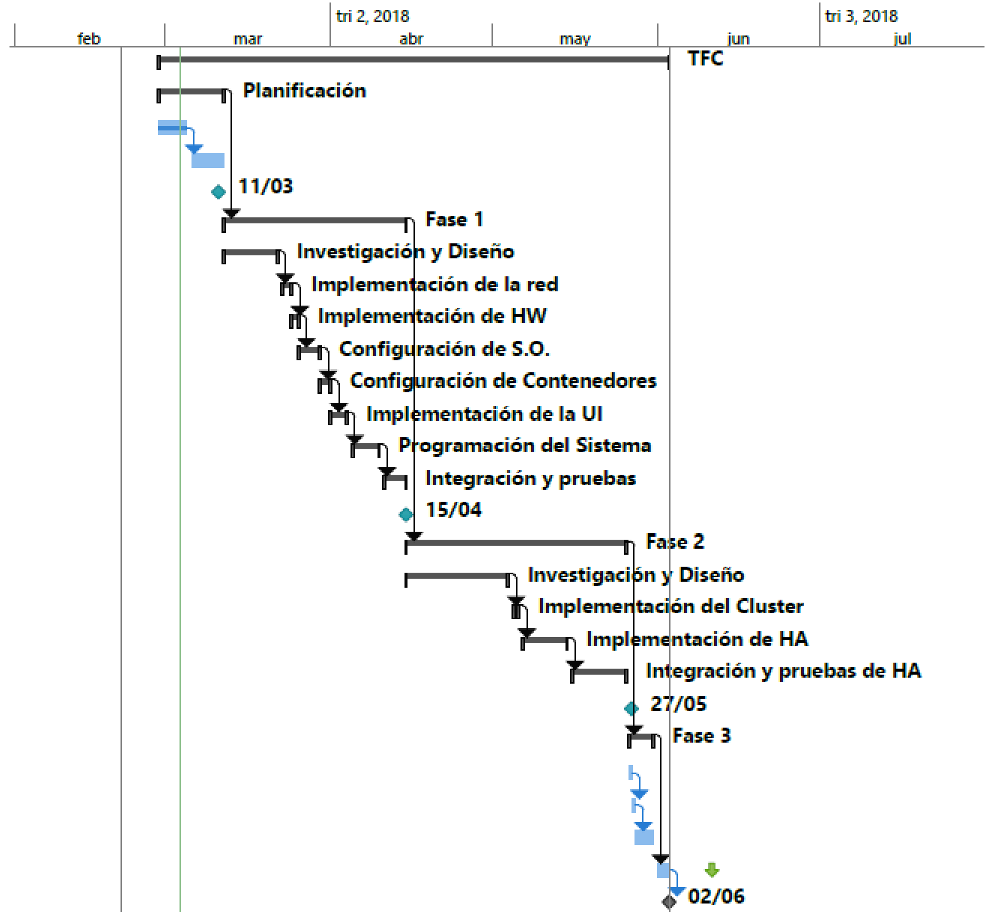
La estimación de duración del proyecto es de 300 horas (Figura 1), dedicando de estas 40 horas a la planificación del proyecto detallada en este documento y 10 horas finales para la creación de las presentaciones del trabajo de fin de grado. Recordemos que según PMI, en la planificación del proyecto, esta estimación puede tener una desviación entre -25% y +75% en tiempo y coste.

1.4.2 Cronograma del proyecto

He utilizado Microsoft Project 2016 para obtener un diagrama de Gantt (Figura 2) después de definir un calendario de recursos con 2 horas diarias entre semana y 5 horas diarias los fines de semana e introducir todas las tareas definidas en la EDT, incluida su duración en horas.

Al haber una única persona para ejecutar este proyecto, todas las tareas se efectúan de manera secuencial, pero en caso de atasco, problema, o necesidad de comprar y esperar la entrega de algún recurso, se pueden ejecutar la fase 1 y la parte teórica de la fase 2 de manera solapada.

Figura 2: Diagrama de Gantt



Como era de esperar, al ser todas las tareas secuenciales, nuestro diagrama de Gantt del proyecto coincide con las 300 horas de duración de nuestra EDT, con una fecha de finalización estimada de 02/06/2018, lo que cumple con la restricción de entregar este proyecto antes de la fecha límite de 10/06/2018.

Observamos en el diagrama que las entregas parciales (PEC) definidas en el calendario del curso coinciden con el final de cada fase del proyecto, por lo que intentaré finalizar la documentación de cada fase antes de cada entrega parcial para mejorar el control de proyecto con estos hitos intermedios.

En el Anexo 5, se indica la lista de ficheros adjuntos entre los cuales podremos encontrar las EDT y el Gantt usados en esta planificación.

1.5 Breve sumario de productos obtenidos

Tal como hemos visto en los apartados anteriores, nuestra intención será obtener un producto funcional al final de cada una de las tres fases:

- Primera Fase: obtendremos un sistema con una microCloud básica que ofrecerá servicios de hogar inteligente en nuestra red local.
- Segunda Fase: obtendremos una microCloud avanzada que podrá extenderse a varias ubicaciones y que nos permitirá desplegar servicios en alta disponibilidad en redes propias y externas.
- Tercera Fase: estudiaremos aplicaciones avanzadas que se pueden desplegar en la microCloud desarrollada en la segunda fase.

1.6 Breve descripción de los otros capítulos de la memoria

En los capítulos siguientes desarrollaremos las tres fases diferenciadas en la planificación y posteriormente repasaremos las conclusiones del trabajo:

- Capítulo 2. Corresponde a la primera fase: estudio, diseño e implementación de la infraestructura básica para nuestra aplicación IoT.
- Capítulo 3. Corresponde a la segunda fase: modificación de la microCloud de la primera fase para aumentar la disponibilidad y escalabilidad.
- Capítulo 4. Corresponde a la última fase. Investigaremos las aplicaciones avanzadas que se pueden ejecutar en nuestra infraestructura final.
- Capítulo 5. Valoración del impacto de la solución desarrollada en el proyecto en redes domésticas y redes comunitarias.
- Capítulo 6. Conclusiones finales que sintetizará todas las conclusiones de cada fase de nuestro proyecto.

2 Primera fase: microCloud local

En esta primera fase, veremos algunas de las soluciones para hogar inteligente que existen actualmente en el mercado, y en base a nuestras observaciones, diseñaremos e implementaremos una infraestructura genérica que ofrezca estos servicios localmente en nuestra red doméstica, pero evitando los inconvenientes de las soluciones propietarias.

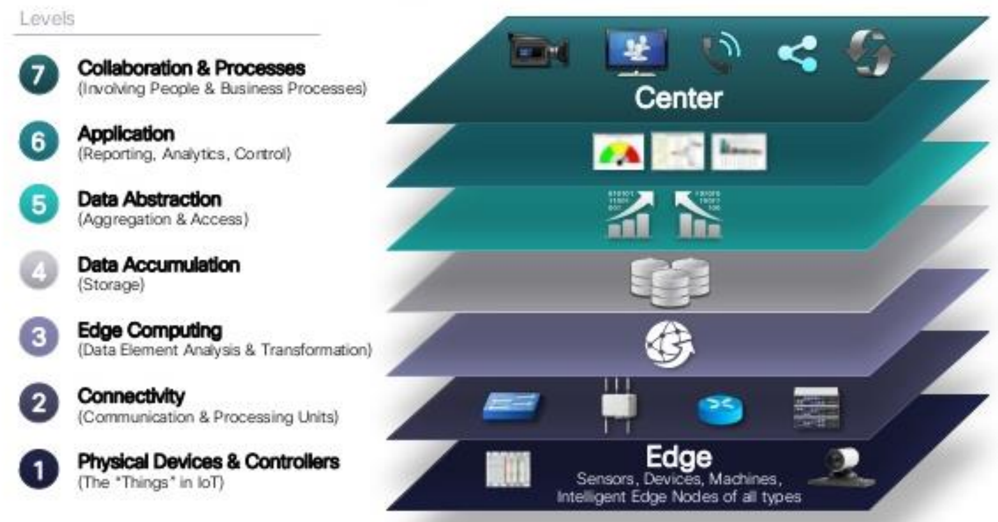
2.1 Estado actual

Estudiaremos el modelo de referencia propuesto por el IoT Word Reference Fórum y veremos cómo se implementa por parte de distintos fabricantes en soluciones con enfoques diametralmente opuestos.

2.1.1 Modelo de referencia IoT

En la Figura 3 podemos ver el modelo de referencia para IoT que propuso Cisco en el IoT World Forum de 2014 **[7].**

Figura 3: IoT World Forum Reference Model



En el nivel 1 y 2, encontramos los nodos IoT (sensores y actuadores) y los dispositivos de red y procesamiento situados en la ubicación del usuario.

En el nivel 3, están los equipos que se sitúan cerca o en el límite (Edge) de la red del usuario y aportan comunicación entre redes y cierto procesamiento de los datos. Estos equipos pueden pertenecer al usuario o al proveedor.

Los niveles 4 a 7 se utilizan para recopilación, análisis de datos y aplicaciones avanzadas. Estos niveles demandan más potencia de cómputo y suelen resolverse mediante soluciones de Cloud Computing por el proveedor.

Queda claro que una solución IoT es un compromiso entre Cloud Computing y Edge Computing. Cuantos más niveles de la arquitectura IoT traslademos a la capa de Edge Computing, más potencia de cálculo será necesaria en nuestra

pasarela IoT y más independientes seremos. A la inversa, cuantos más niveles se trasladen a la nube, menos potencia necesitará nuestra pasarela, hasta llegar al extremo de desaparecer, pero más dependeremos del proveedor.

Cuando la necesidad de potencia de cálculo para el Edge Computing se resuelve de forma colaborativa entre varios equipos Edge y/o los propios dispositivos de la IoT, este paradigma toma el nombre de Fog Computing **[8]**.

A continuación, veremos dos soluciones comercializadas en la actualidad para proporcionar servicios de hogar inteligente en una red doméstica.

2.1.2 Solución de hogar inteligente de Vera Control, Ltd.

En la Figura 4 apreciamos una instalación de hogar inteligente con un sistema Vera **[9]**. Esta solución tiene como característica principal el no necesitar ningún servicio Cloud, usando Internet únicamente para acceso remoto:

- En la figura tenemos 3 redes: una LAN para conectarnos a Internet, una Wifi para conectar los equipos de los usuarios y una red especifica de dispositivos IoT, en este caso con tecnología Zigbee.

- La interfaz de usuario se ejecuta en un dispositivo móvil del usuario, un smartphone o una Tablet, que se conecta a la pasarela IoT con Wifi.

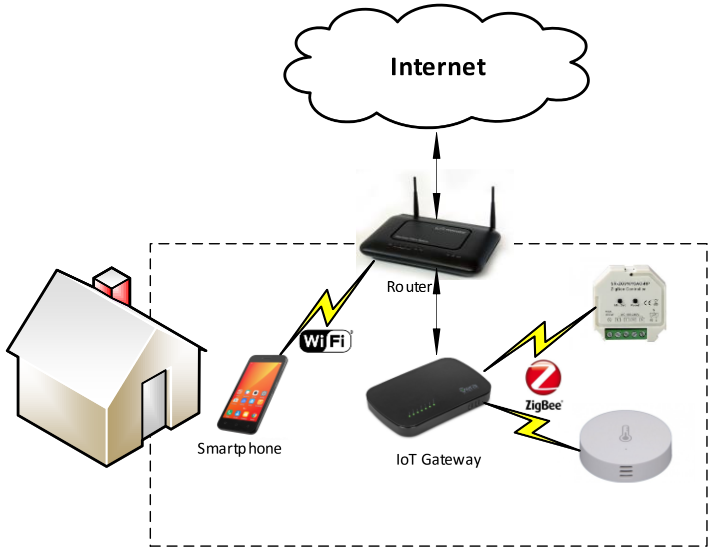
- La pasarela IoT acapara todos los niveles del modelo de referencia IoT, por eso no es necesario ningún servidor Cloud.

Y estos son los inconvenientes que podemos reseñar:

- No tenemos ningún control sobre la red Zigbee, no tenemos herramientas de red, ni acceso a al firmware de los dispositivos IoT.

- Las funcionalidades complejas (análisis de datos, estadísticas, etc…) están limitadas por la potencia de la pasarela IoT.

Figura 4: Solución de hogar inteligente de Vera



2.1.3 Solución de hogar inteligente de Itead, Ltd.

La solución del fabricante Itead **[10]** que podemos ver en la Figura 5 es una solución enfocada al mercado de bajo coste. Estas son las diferencias con relación a la solución de Vera que hemos estudiado en el apartado anterior:

- Los nodos de la IoT usan Wifi, pero solo la usan para comunicarse directamente con el servidor Cloud situado en Internet.
- No existe una pasarela IoT en la red local. Es el servidor Cloud el que acapara todos los niveles superiores (3 al 7) del modelo de referencia.
- Solo existen dos redes, una red local cableada para conectar la pasarela Iot y una red Wifi compartida para los nodos y los usuarios.
- La interfaz de usuario solo se comunica directamente con el servidor Cloud en Internet, no con los dispositivos locales.

Estos son los inconvenientes de situar todos los servicios en el servidor Cloud:

- En caso de pérdida de la conexión con el servidor Cloud solo se puede operar los dispositivos IoT de forma manual.
- La red Wifi de propósito general es menos fiable que una red dedicada a comunicación de los dispositivos IoT.
- Se comparte el tráfico entre los usuarios y los dispositivos IoT en la misma red.
- Las contraseñas de nuestra wifi y todos nuestros datos de uso están en un servidor ajeno a nuestra red del cual no tenemos control.
- El tiempo de respuesta de los dispositivos es excesivo ya que es el tiempo de respuesta del servidor Cloud.

Figura 5: Solución de hogar inteligente de ITEAD

2.2 Diseño del Sistema

En vez de usar un hardware y software específico y especializado para gestionar los nodos IoT, diseñaremos una microCloud abierta y versátil que asumirá entre otras cosas, las funciones de pasarela IoT dentro de nuestra LAN. En este caso no estaremos limitados por el hardware o el software ya que aunque la microCloud sea independiente de la conexión a Internet, esta será escalable y extensible, lo que nos permitirá ejecutar funciones avanzadas tales como alta disponibilidad o visualización y análisis de datos.

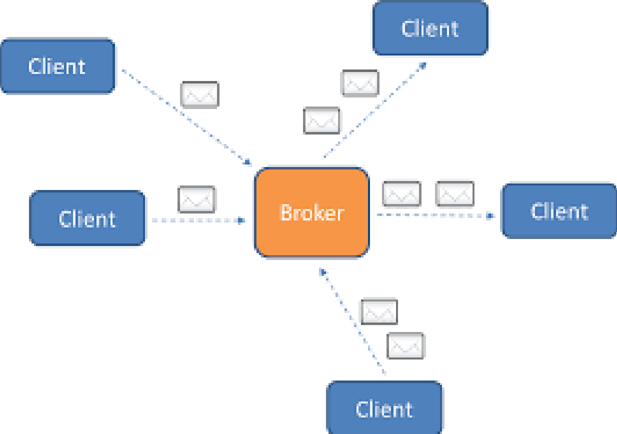
2.2.1 Protocolo de comunicación

Como protocolo de comunicación de nuestra microCloud, proponemos el protocolo MQTT (Message Queuing Telemetry Transport). MQTT **[11]** es un estándar ISO (ISO/IEC PRF 20922) basado en el paradigma de publicación/subscripción para paso de mensajes.

MQTT es un protocolo de comunicación M2M (Machine to Machine) optimizado para implementarse en dispositivos con pocos recursos y con un ancho de banda reducido y que se ejecuta sobre redes TCP/IP, en nuestro caso lo ejecutaremos en una red Wifi.

Para poder usar el protocolo en nuestra red local, necesitaremos un nodo central que ejecute un Bróker de mensajes, ya que MQTT implementa una arquitectura en estrella (Figura 6). Este software se ejecutará en la microCloud. Esta configuración es ideal para mensajes de uno a muchos.

Figura 6: Arquitectura genérica MQTT



La comunicación se basa en mensajes con un tópico (tema) y un contenido. Los publicadores envían mensajes al bróker MQTT y los subscriptores se subscriben en el bróker a los tópicos de su interés para recibir los mensajes relacionados. Un nodo puede ser a la vez publicador y subscriptor

Los protocolos basados en el paradigma de Publicación/Suscripción como MQTT se caracterizan por estar espacialmente desacoplados, es decir que publicadores y subscriptores no necesitan conocerse entre sí **[12]**. Esto nos permitirá añadir o reemplazar fácilmente cualquier componente de nuestro sistema sin necesidad de modificar otras partes.

2.2.2 Redes

En una instalación clásica como en la solución de Vera (2.1.2), existen tres redes distintas: una LAN para conectar la pasarela IoT con Internet, una red Wifi para permitir a los usuarios acceder a la interfaz de usuario y una red LoT (LAN of Things) donde ubicaremos nuestros dispositivos IoT.

En nuestro diseño, en vez usar redes diferentes para usuarios y dispositivos, usaremos únicamente una única red inalámbrica como en la solución de Itead (2.1.3). Esta única red será una Wifi ya que así reduciremos los costes tanto de los nodos IoT como de la pasarela, debido a que es una tecnología madura, muy difundida, muy amortizada y por lo tanto muy barata.

Otra ventaja añadida al usar Wifi para nuestros dispositivos IoT es que el protocolo MQTT elegido anteriormente para la comunicación entre nuestros dispositivos se ejecuta directamente sobre redes TCP/IP, como es el caso, por lo que no necesitaremos adaptar o encapsular esas comunicaciones.

Usaremos como enrutador de la LAN un Safe@Office 500 **[13]**, que es un recurso recuperado que ya teníamos disponible, por lo que no supondrá un coste adicional. Así tendremos un control total sobre nuestra configuración de red sin afectar a nuestra LAN y nos evitaremos la compra del Switch de red ya que este dispositivo incorpora cuatro puertos LAN ethernet.

En vez de conectar el Safe@Office directamente a Internet, lo conectaremos a nuestra red doméstica. De esta manera podremos simular la conexión de nuestra LAN a una red comunitaria (CWAN).

En nuestra CWAN simulada, podremos crear otros nodos Cloudy, en esta primera fase crearemos 'Cloudy-Ext1' y 'Cloudy-Ext2'. Al estar estos nodos completamente bajo nuestro control, podremos comprobar la publicación y el acceso de los servicios de nuestra microCloud desde esos nodos externos.

Figura 7: Diseño de la red

Para el direccionamiento de nuestras redes propondremos una configuración de redes privada de clase C (Figura 8), lo que nos proporcionará la capacidad de conectar hasta 254 dispositivos en cada red. Utilizaremos un único direccionamiento para la red LoT, tanto para la parte cableada (microCloud) como la parte inalámbrica (nodos IoT y otros clientes).

Figura 8: Configuración de Redes

Descripción	Direccionamiento	Mascara	Gateway
Red LoT	192.168.10.0	255.255.255.0	192.168.10.1
Red CWAN	172.16.0.0	255.255.255.0	172.16.0.1

Para conseguir un direccionamiento único para nuestra LoT implementaremos Bridging entre la red cableada y la red inalámbrica del Safe@Office. De esta manera, además, podremos compartir protocolos que no se pueden enrutar como UDP entre la pasarela IoT y los nodos.

Usaremos DHCP para nuestra LoT para facilitar la conexión de los nodos ya que con el protocolo MQTT no es necesario saber la dirección IP un nodo para enviar y recibir mensajes. Asignaremos direccionamiento de red estático a todos los demás equipos que detallaremos en la tabla de la Figura 9.

Figura 9: Direccionamiento de los equipos

Descripción	Dirección IP	Mascara	Gateway	IP Publica
Nodos LoT	192.168.10.1xx	255.255.255.0	192.168.10.1	
microCloud	192.168.10.10	255.255.255.0	192.168.10.1	172.16.0.253
Safe@Office WAN	172.16.0.254	255.255.255.0	172.16.0.1	172.16.0.254
Safe@Office LoT	192.168.10.1	255.255.255.0		
Router Internet	172.16.0.1	255.255.255.0		
Cloudy-Ext1	172.16.0.203	255.255.255.0	172.16.0.1	
Cloudy-Ext2	172.16.0.204	255.255.255.0	172.16.0.1	

2.2.3 Hardware

En este apartado detallaremos todo el hardware que vamos a usar para la microCloud y los nodos IoT de nuestro sistema domótico.

2.2.3.1 Hardware de la microCloud

Para el primer nodo de nuestra microCloud usaremos una Raspberry pi 3, recurso que ya tenemos disponible y que tiene la suficiente potencia para ejecutar el sistema operativo Linux y los servicios necesarios para nuestra LoT.

Figura 10: Raspberry pi 3

La Raspberry pi 3 (Figura 10) es un SBC (Single Board Computer) con arquitectura ARM de 64 bits que tiene las características siguientes:

- Procesador: Chipset Broadcom BCM2387. 1,2 GHz de cuatro núcleos ARM Cortex-A53.
- GPU. Dual Core VideoCore IV ® Multimedia Co-procesor.
- RAM: 1GB LPDDR2.
- Ethernet socket Ethernet 10/100 BaseT. 802.11 b / g / n LAN inalámbrica y Bluetooth 4.1 (Classic Bluetooth y LE)

Aunque la Raspberry Pi 3 tiene capacidades inalámbricas, no las usaremos de momento ya que la conectaremos a nuestra LoT mediante una red cableada. Para conectar los usuarios y los dispositivos inalámbricos de nuestra LoT usaremos la red WLAN proporcionada por nuestro enrutador Safe@Office.

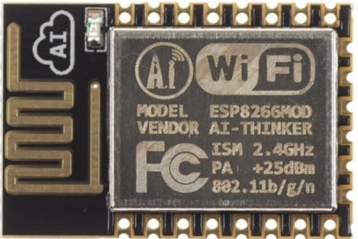
Montaremos la Raspberry en una sencilla caja acrílica apilable pensando en la posibilidad de ampliar nuestra instalación en la segunda fase. Para instalar el sistema operativo y como almacenamiento principal usaremos una tarjeta microSD clase 10 de 16GB.

2.2.3.2 Hardware de los nodos IoT

El ESP8266 **[14]** es un SoC (System on Chip) producido por el fabricante Espressif para proporcionar capacidades Wifi por un coste muy bajo. Este chip incluye una CPU RISC de 32 bits, una interfaz serie, una antena integrada (Figura 11) y lo más importante, una pila completa TCP/IP que soporta IPv4 y los protocolos TCP / UDP / HTTP y FTP.

Al principio el chip se publicitó como un módulo de Wifi a Serie para ser usado como interfaz inalámbrica para otros procesadores, pero tras comprobar que solo necesita un 20% de potencia de su CPU para gestionar sus capacidades Wifi, la comunidad Open Source empezó a desarrollar firmwares para su uso como un procesador autónomo, culminando con la integración del chip en el entorno de programación de Arduino.

Figura 11: SoC ESP8266



Los nodos IoT que usaremos estarán pues basados en el SoC ESP8266 y podrán bien ser creados ad hoc con placas de desarrollo, por ejemplo las del fabricante Wemos **[15]**, o bien comprando y reprogramando los dispositivos comercializados que podemos encontrar disponibles en el mercado.

Un ejemplo de dispositivos reprogramables con SoC EsP8266 son la gama de dispositivos para Smart Home llamada Sonoff del fabricante Itead que vimos en el apartado 2.1.3.

Las características de los dispositivos Sonoff son las siguientes:

- Están basados en el chip ESP8266
- Tienen la certificación CE
- Funcionan conectados a la red eléctrica de 220V
- Son relativamente fáciles de reprogramar
- Son de muy bajo costo

En la tabla de la Figura 12, mostramos un pequeño estudio de coste de distintos dispositivos IoT de equivalente funcionalidad, pero con tecnologías inalámbricas distintas que hemos encontrado en el sitio web amazon.es:

Figura 12: Costes de dispositivos IoT en amazon.es

Tecn. Inalamb.	Fabricante	Dispositivo	Coste
Wifi	ITEAD	Sonoff S20	17,99 Euros
Zigbee	BITRON	Enchufe Inteligente	56,69 Euros
Zwave	FIBARO	Wall Plug	58,99 Euros

Al poder reprogramar estos dispositivos con SoC ESP8266 con el entorno Arduino, la comunidad Open Source ha desarrollado varios firmwares alternativos para la Gama Sonoff que permite conservar toda su funcionalidad al mismo tiempo que proporciona la capacidad de integrarse en una Red MQTT y permite además añadir de forma fácil sensores adicionales a los dispositivos.

Reprogramaremos los dispositivos con el firmware Sonoff-Tasmota de Theo Arendst **[16]**. Este firmware proporciona entre otras características actualizaciones OTA (Over the Air) desde el entorno Arduino y soporta además dispositivos de otros fabricantes basados en ESP8266.

2.2.4 Software

En este apartado detallaremos el Sistema Operativo, el gestor de contenedores y los distintos contenedores que ejecutaremos para crear los servicios que incluiremos en nuestra microCloud.

2.2.4.1 Sistema Operativo

Una vez elegido el Hardware del servidor, y tras haber repasado sus características para comprobar qué sistemas operativos son compatibles con el dispositivo, nos decantamos por el sistema operativo Raspbian, una distribución Linux basada en Debian que está compilada y optimizada para ejecutarse en la Raspberry pi.

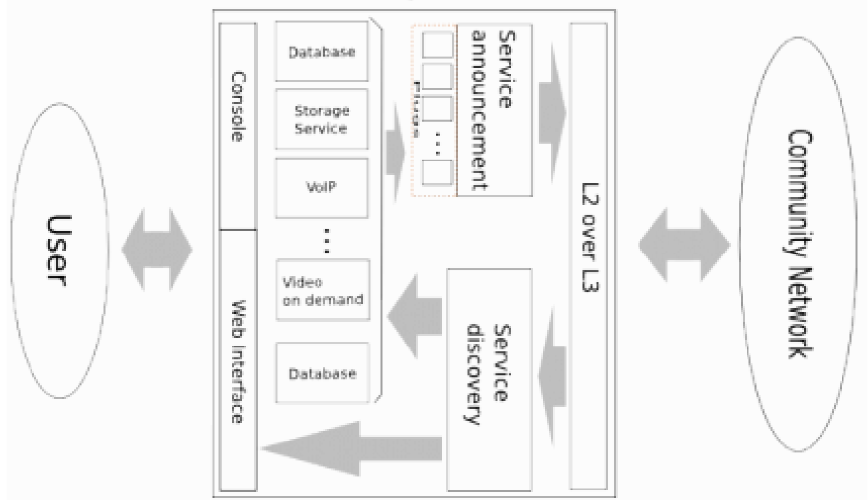
Sin embargo, Raspbian no será exactamente el sistema operativo definitivo que ejecutaremos en nuestra Raspberry ya que uno de nuestros requisitos es poder compartir fácilmente en redes comunitarias los servicios que sean desarrollados en este proyecto.

Usaremos el sistema operativo Cloudy porque es una distribución de Linux orientada a redes comunitarias que proporciona varios mecanismos para publicar y compartir servicios (Figura 13). Cloudy se usa actualmente para proporcionar servicios para la infraestructura de la red comunitaria Guifi.net.

Cloudy tiene como base un sistema Debian con una interfaz de usuario creada en PHP llamada cDistro que se ejecuta en un servidor propio en los puertos 7000 o 7443. La interfaz cDistro nos da acceso a servicios y aplicaciones preconfigurados y además tiene la capacidad de ejecutar y publicar contenedores Docker. Para publicar los servicios localmente se usa Avahi, y para publicar los servicios en otros nodos de la red comunitaria se usa Serf.

Para instalar Cloudy, ejecutaremos un procedimiento llamado Cloudynitzar **[17]** que permite transformar una instalación de Raspbian en una instalación de Cloudy ejecutable en una Raspberry pi.

Figura 13: Arquitectura de Cloudy



Una de las dificultades que nos encontraremos durante este proyecto es que al tener la Raspberry pi 3 una arquitectura ARM, deberemos adaptar y usar aplicaciones o imágenes compiladas para esa arquitectura en concreto. .

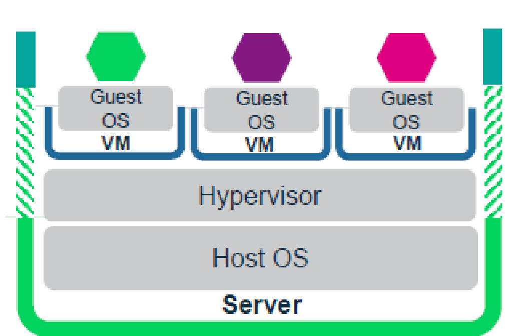
2.2.4.2 Gestor de Contenedores

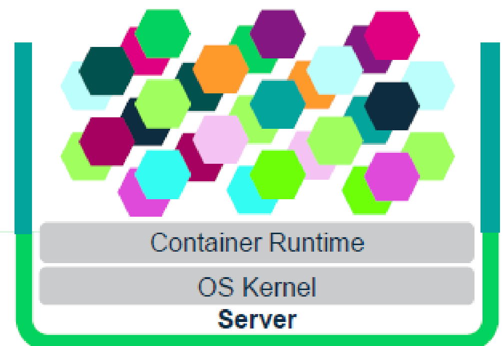
Para implementar los servicios necesarios para nuestra microCloud usaremos preferentemente la tecnología de contenedores Docker que ya está integrada en Cloudy con el objetivo de facilitar la reusabilidad de las configuraciones y tener la posibilidad en la segunda fase de usar el modo Swarm de Docker **[18]**.

Usaremos preferentemente los contenedores con servicios preconfigurados ya disponibles para Cloudy, pero buscaremos o generaremos nuevos

contenedores con arquitectura ARM según sean necesarios para el funcionamiento de nuestro nuevo sistema.

Figura 14: Containers vs Virtual Machines





En la Figura 14 podemos comprobar las ventajas e inconveniente de usar una plataforma de contenedores como Docker con respecto a ejecutar máquinas virtuales en un hipervisor **[19]**. Las ventajas son las siguientes:

- Para ejecutar contenedores se necesitan muchos menos recursos que para ejecutar máquinas virtuales. Por esta razón se puede ejecutar un gestor de contenedores como Docker un SBC con recursos limitados como nuestra Raspberry pi.

- Los contenedores son entornos preconfigurados que se pueden ejecutar en distintas instalaciones, por lo que ya están testeados y permiten poner en producción servicios de forma mucho más rápida.

Y estas son las desventajas inherentes en el uso de contenedores:

- Son menos seguros que las máquinas virtuales, ya que comparten un único kernel en el servidor y están mucho menos aislados entre sí.

- No se pueden usar sistemas operativos o arquitecturas distintas del sistema operativo del Host. Por lo que, en este caso, los contenedores tendrán que ser contenedores Linux con arquitectura ARM.

2.2.4.3 Bróker MQTT

Como ya hemos visto en el apartado 2.2.1, necesitamos un Bróker de mensajes para implementar el protocolo MQTT en nuestra microCloud.

Mosquitto es una implementación estándar de un bróker para el protocolo de mensajes MQTT que está diseñado para ser usado en equipos con bajos recursos **[20]**. De hecho, este software existe en Raspbian ya compilado para la arquitectura ARM de la Raspberry Pi.

En Cloudy, además ya existe un contenedor estándar de Mosquitto para Docker que nos puede ofrecer un servicio preconfigurado que se puede instalar fácilmente desde la interfaz web.

2.2.4.4 Entorno de programación

La lógica de funcionamiento y la integración de todos los componentes del sistema se implementará con la herramienta de programación de flujos Node-Red en el lenguaje JavaScript **[21]**.

Node-Red es una plataforma enfocada al IoT desarrollada inicialmente por IBM, permite enlazar dispositivos hardware, APIs y servicios Online. Node-Red permitirá acceder a los usuarios a las funcionalidades implementadas directamente desde una interfaz web y desarrollar y probar nuevas funcionalidades de forma sencilla e interactiva.

Es una herramienta ideal para entornos experimentales como es nuestro caso por su flexibilidad, extensibilidad y el variado ecosistema de nodos que existe en torno a esta aplicación (más de 1400 nodos adicionales).

En este caso, tras comprobar que no hay predefinido en la instalación de Cloudy ningún contenedor de Node-Red, buscaremos en Docker Hub un contenedor Node-Red preparado para ejecutarse en arquitectura ARM con el objeto de integrarlo en nuestra microCloud.

2.2.4.5 Interfaz de usuario

Instalaremos y configuraremos como interfaz de usuario la aplicación de Android para crear Dashboards IoT llamada MQTT-Dash **[22]**. Esta aplicación, desarrollada por Vadim V. Mostovoy es gratis y se distribuye con licencia Creative Common cc by 4.0.

MQTT-Dash es una aplicación que se conectará como cliente al bróker MQTT de nuestra microCloud. Se puede usar en un móvil o una Tablet para manejar los dispositivos IoT de forma remota como un mando a distancia inalámbrico.

Podremos crear controles en nuestro teléfono que enviarán y/o recibirán mensajes MQTT con distintos tópicos y contenidos. Con esta interfaz, daremos acceso a nuestro sistema de hogar inteligente a los usuarios menos experimentados de nuestro sistema domótico.

2.3 Implementación del sistema

Una vez elegidos todos los componentes de nuestro sistema, pasamos de la fase de diseño a la fase de implementación.

2.3.1 Implementación de las redes

Procederemos a configurar nuestras redes en la interfaz web del enrutador Safe@Office, entre otros configuraremos los siguientes apartados:

- Configuración de la conexión WAN.
- Configuración de las redes Internas LAN y WLAN

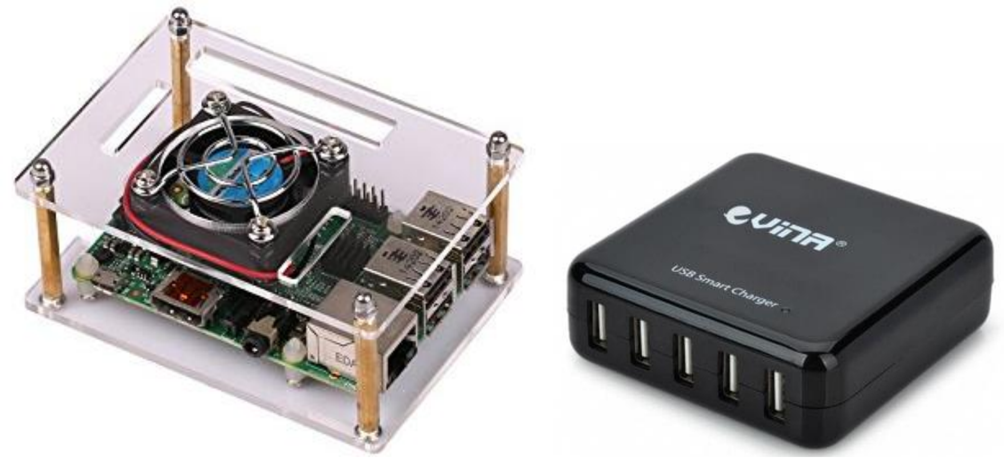
- Creación de un Bridge entre las redes internas
- Activación de NAT en las redes internas
- Reserva de DHCP para equipos fijos
- Creación de Reglas de Salida y Entrada

En el Anexo 1 (Configuración de red de la Fase1) está registrada la configuración del Safe@Office 500 en esta primera fase.

2.3.2 Hardware de la microCloud

Montamos la Raspberry pi en su caja acrílica, conectando el ventilador incluido en los pines GPIO 4 y 6 (+5v y GND). De momento no incluimos la tarjeta microSD ya que antes de instalarla tenemos que efectuar la instalación del sistema operativo desde nuestro portátil Windows.

Figura 15: Raspberry pi 3 con caja apilable y cargador de alta potencia



Alimentaremos nuestra Raspberry con un cargador USB de 5 bocas de 40w y 8A (Figura 15) lo que nos permitirá ampliar fácilmente nuestro montaje en la fase 2. Finalmente conectaremos nuestra Raspberry al Safe@Office con un cable de red a uno de los 4 puertos LAN disponibles.

2.3.3 Instalación del Sistema Operativo Cloudy

Primero haremos la instalación y configuración básica del sistema operativo Raspbian en la microSD desde nuestro portátil. Configuraremos la Raspberry sin necesidad de conectar monitor ni teclado (headless) siguiendo las instrucciones detalladas en el sitio web hackernoon **[23]**. El nombre de este primer nodo de la microCloud será ‘SmartHome1’.

Siguiendo las instrucciones anteriores, una vez arrancado el S.O., nos conectaremos desde nuestro portátil con el protocolo SSH a la IP de ‘SmartHome1’ usando el usuario por defecto de Raspbian, ‘pi’ y lo primero que

haremos será desde la utilidad 'raspi-config' cambiar la contraseña de este usuario por defecto y el nombre del equipo.

Configuraremos primero la red en 'SmartHome1' para asignar a nuestra Raspberry pi el direccionamiento de la microCloud (Figura 9), lo haremos configurando la dirección IP de forma estática en el fichero '/etc/dhcpcd.conf'.

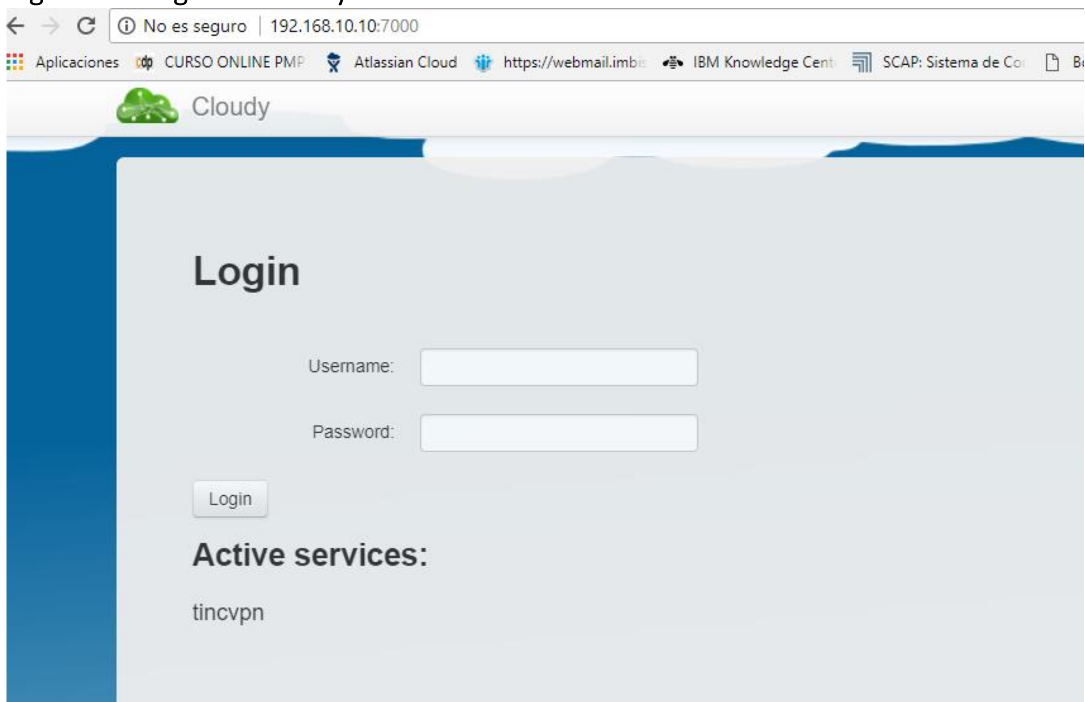
```
# Example static IP configuration:
interface eth0
static ip_address=192.168.10.10/24
static routers=192.168.10.1
static domain_name_servers=192.168.10.1 8.8.8.8 fd51:42f8:caae:d92e::1
```

Una vez configurado Raspbian con 'raspi-config', lo convertiremos en Cloudy ejecutando el Script 'Cloudytnizar.sh' que está disponible en GitHub **[24]**.

```
pi@SmartHome1:~ $ sudo bash
root@SmartHome1:/home/pi# apt-get update; apt-get install -y curl lsb-release
root@SmartHome1:/home/pi# curl -k
https://raw.githubusercontent.com/Clommunity/cloudynitzar/master/cloudynitzar
.sh | bash -
```

Ya solo nos queda comprobar que Cloudy está instalado correctamente accediendo con un navegador a su interfaz web a través la IP de '192.168.10.10' en el puerto 7000 (Figura 16).

Figura 16: Login de Cloudy en 'SmartHome1'



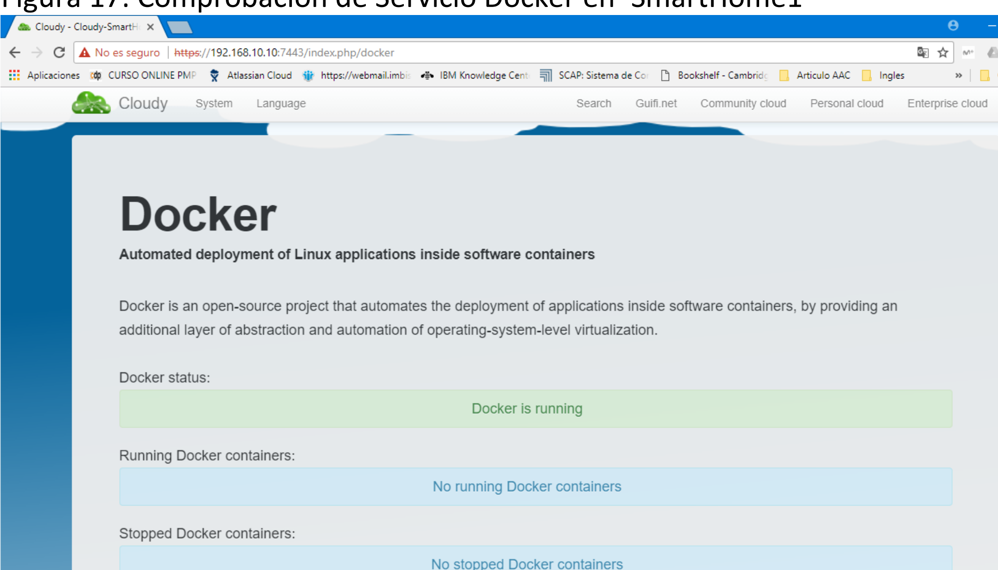
El nombre y contraseña para acceder a Cloudy, pueden ser el nombre y contraseña del usuario 'pi' que hemos visto anteriormente, o de cualquier usuario que creemos en nuestro sistema operativo.

2.3.4 Instalación de Docker

Como ya hemos comentado anteriormente, Docker ya está integrado en la interfaz web de Cloudy. Por lo que podremos instalarlo y ponerlo en marcha efectuando los siguientes pasos través de nuestro navegador:

- Ir a la opción de menú 'Enterprise Cloud'.
- Nos aparecerá una pantalla indicando que faltan las fuentes de Docker en el gestor de paquetes. Pulsaremos el botón para instalar las fuentes.
- Tras instalar las fuentes, Cloudy nos indica que ya podemos instalar el software, por lo que pulsaremos el botón para instalar Docker ce (Community Edition).
- Nos aparecerá una pantalla del gestor de paquetes, donde confirmaremos la instalación del paquete Docker ce.
- Al finalizar la instalación, nos aparecerá una pantalla indicando que Docker está funcionando correctamente (Figura 17).

Figura 17: Comprobación de Servicio Docker en 'SmartHome1'



2.3.5 Instalación del contenedor Mosquitto

Nuestro sistema necesita un contenedor Docker de Mosquitto para ofrecer servicios de Bróker MQTT. Sin embargo, al intentar instalar Mosquitto, comprobamos que desgraciadamente no existe en Cloudy ningún contenedor predefinido para la arquitectura ARM.

Afortunadamente Cloudy nos da opción de buscar e instalar imágenes de contenedores desde Docker Hub. Posteriormente podremos configurarlos y publicarlos, desde la opción 'Docker FORM' de la interfaz web.

Seleccionamos la imagen de Docker Hub 'robotany/mosquitto-rpi' que tiene las siguientes características:

- Bróker MQTT Mosquitto v1.4.12
- Arquitectura ARM
- 45 MB de tamaño, 15MB comprimido
- Posibilidad de montar volúmenes para configuración, datos y logs
- Puertos MQTT y WebSocket

Para ejecutar el contenedor de Mosquitto efectuaremos los siguientes pasos:

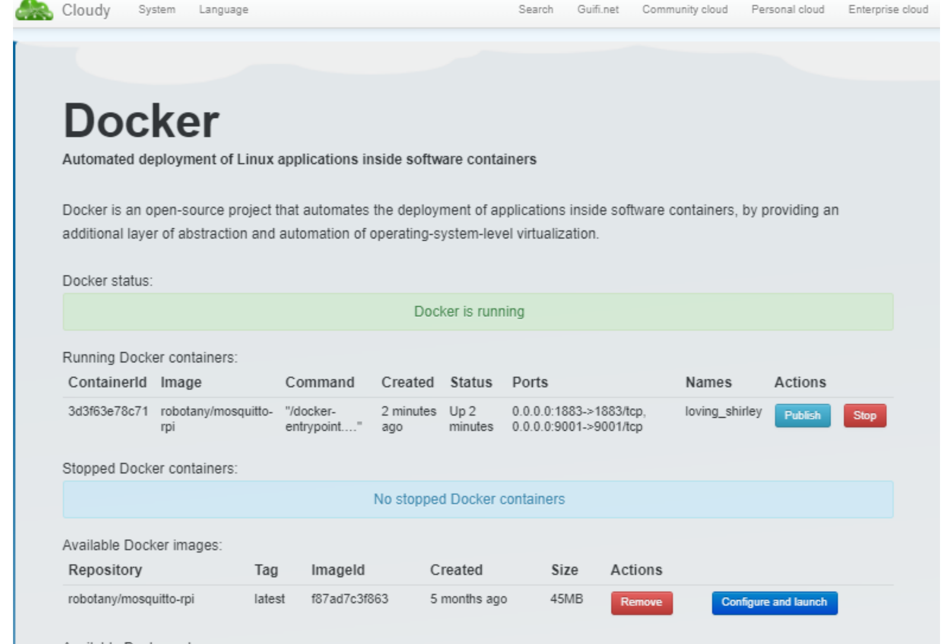
- Configuraremos el contenedor en la opción 'Docker FORM'
- Lanzaremos el contenedor desde 'Predefined Docker Images' – botón 'Configure' – botón 'Launch'.

Comprobamos que el contenedor no arranca automáticamente en caso de reinicio. Para resolver este problema, nos conectaremos mediante el protocolo SSH con nuestro usuario administrador y ejecutaremos estos comandos:

```
pi@SmartHome1:~ $ sudo docker update --restart always loving_shirley
pi@SmartHome1:~ $ sudo reboot
```

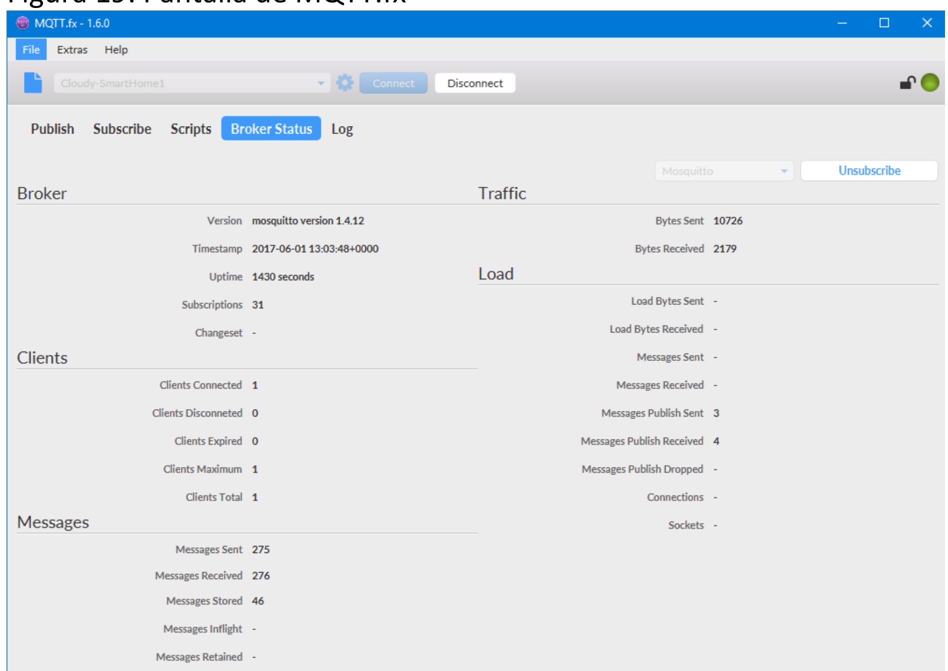
Donde 'loving_shirley' es el nombre de nuestro contenedor Mosquitto. De esta manera actualizamos la política de rearranque del contenedor y al reiniciar SmartHome1, comprobamos en la opción Docker del menú de Cloudy que los contenedores ya arrancan correctamente (Figura 18).

Figura 18: Comprobación de contenedores



Y finalmente para comprobar que el bróker MQTT está funcionando correctamente, podemos usar un cliente MQTT desde nuestro portátil o teléfono, por ejemplo, el cliente MQTT.fx **[25]** (Figura 19) con el que nos conectarnos desde nuestro portátil en la LoT.

Figura 19: Pantalla de MQTT.fx



2.3.6 Instalación del Contenedor Node-Red

Seleccionaremos en Docker Hub una imagen de contenedor Docker de Node-Red con arquitectura ARM en el sitio Web Docker Hub, en concreto elegimos 'nodered/node-red-docker' con las siguientes características:

- Node-Red v0.18.4
- Arquitectura ARM usando el tag rpi-v8
- Posibilidad de montar un volumen para los datos
- Tamaño 560MB. Comprimido 213MB

Seguimos los mismos pasos que efectuamos en el apartado anterior:

- Configuración del contenedor en la opción 'Docker FORM'
- Lanzar el contenedor desde 'Predefined Docker Images' – botón 'Configure' – botón 'Launch'.
- Comando 'Docker update' para actualizar la política de rearranque.

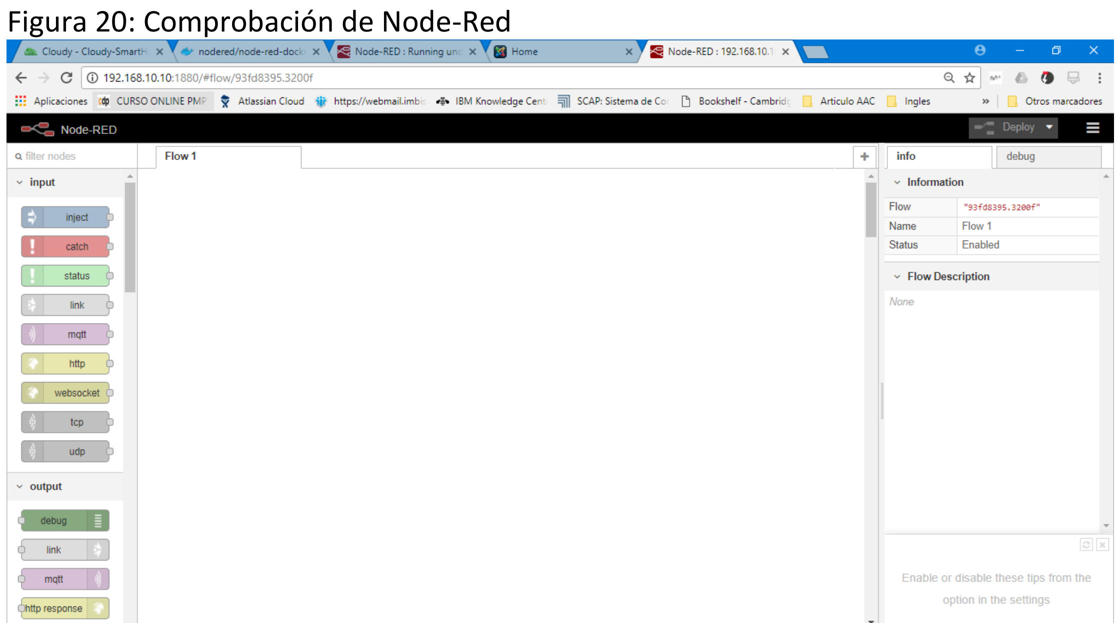
En este caso observamos que el contenedor al ser lanzado crea un nuevo volumen anónimo de Docker. En este volumen, es donde se guardará toda la configuración y código de Node-Red que desarrollemos durante el proyecto.

```
joe@SmartHome2:~ $ sudo docker volume list
DRIVER              VOLUME NAME
local               8f4f0a26c74ca1bd8a5665c12e322c14a3d046a142f9e1756efc4c86
```

Este volumen es usado por el contenedor cada vez que arranca y se mantiene, aunque se reinicie el dispositivo. Este comportamiento proporciona la persistencia de la configuración y de los flujos que vayamos creando.

Comprobamos que podemos acceder a Node-Red en el puerto 1880 de la IP de la pasarela IoT (Figura 20) desde el navegador de nuestro portátil en la LoT:

Figura 20: Comprobación de Node-Red



Con este procedimiento damos por finalizada la instalación y puesta en marcha de nuestra microCloud y nos centraremos en los dispositivos clientes.

2.3.7 Configuración de los nodos IoT

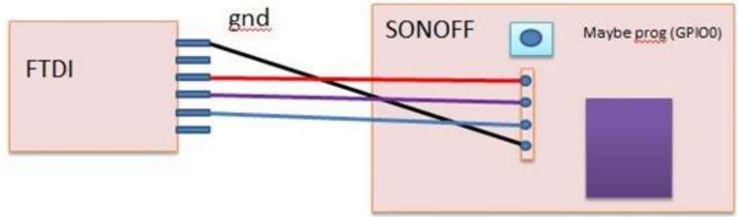
Tal como hemos comentado en la fase de diseño, usaremos dispositivos de la gama Sonoff **[26]** del fabricante Itead que reprogramaremos con el Firmware Sonoff-Tasmota de Theo Arendst.

Tenemos disponibles los siguientes dispositivos:

- Un enchufe inteligente Itead Sonoff S20
- Un Itead Sonoff Basic, un relé básico
- Un Itead Sonoff POW, un relé con medición de consumo
- Un Itead Sonoff B1, una bombilla RGB regulable

Efectuaremos la reprogramación de los nodos con un adaptador USB-TTL (generalmente conocido como FTDI) de 3v3 (Figura 21) ya que según las características técnicas del SoC ESP8266, esta es su tensión de alimentación.

Figura 21: Adaptador USB a TTL

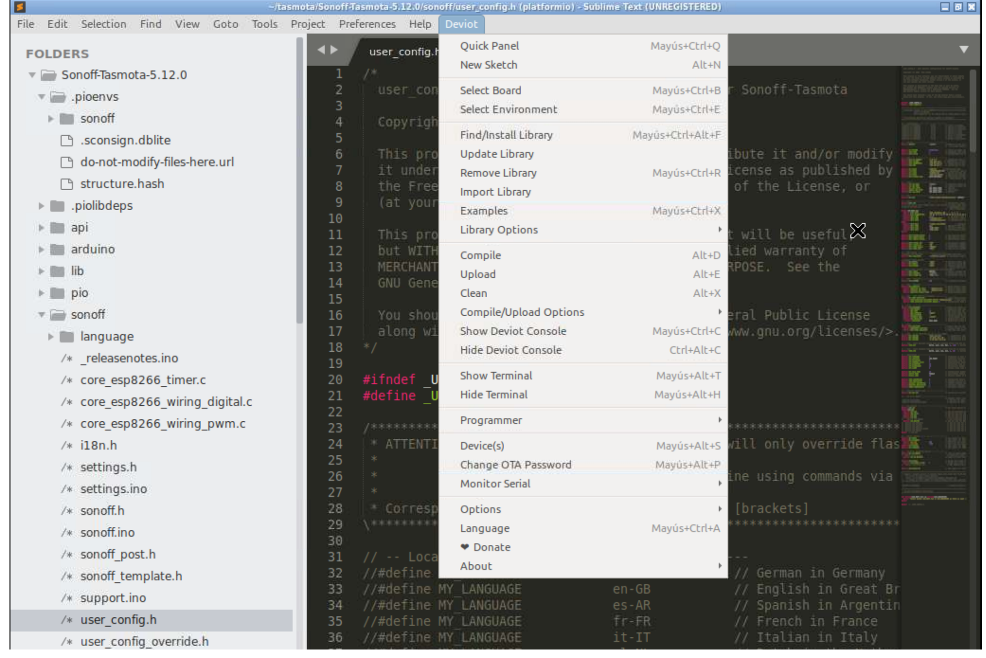


Seguiremos los pasos genéricos de la wiki de Tasmota **[27]** para todos nuestros sonoffs. En cada dispositivo existe una hilera de agujeros vacíos donde se pueden soldar los pines para efectuar la reprogramación, sin embargo, evitaré soldar apoyando la cabecera de 4 pines con la mano durante la carga del firmware.

Como entorno de programación, dispongo de un viejo ordenador de sobremesa Dell Optiplex G60 que al tener una CPU Celeron, solo puede ejecutar distribuciones de 32 bits de Linux. En este entorno he instalado la distribución Linux Lubuntu y el editor Sublime Text junto con el plugin de desarrollo IoT Deviot. Deviot es la única forma de integrar la plataforma de desarrollo platform.io a un editor en un entorno de 32 bits (Figura 22).

Tras descargar el firmware Tasmota, deberemos modificar los ficheros ‘platformio.ini’ y ‘user_config.h’ para acomodarlas a nuestra instalación. La configuración final de ‘platformio.ini’ y ‘user_config.h’ para la microCloud se puede consultar en los ficheros adjuntos a la memoria (Anexo 5).

Figura 22: Entorno Deviot



Tras grabar las modificaciones, creamos el firmware con la opción ‘compile’.

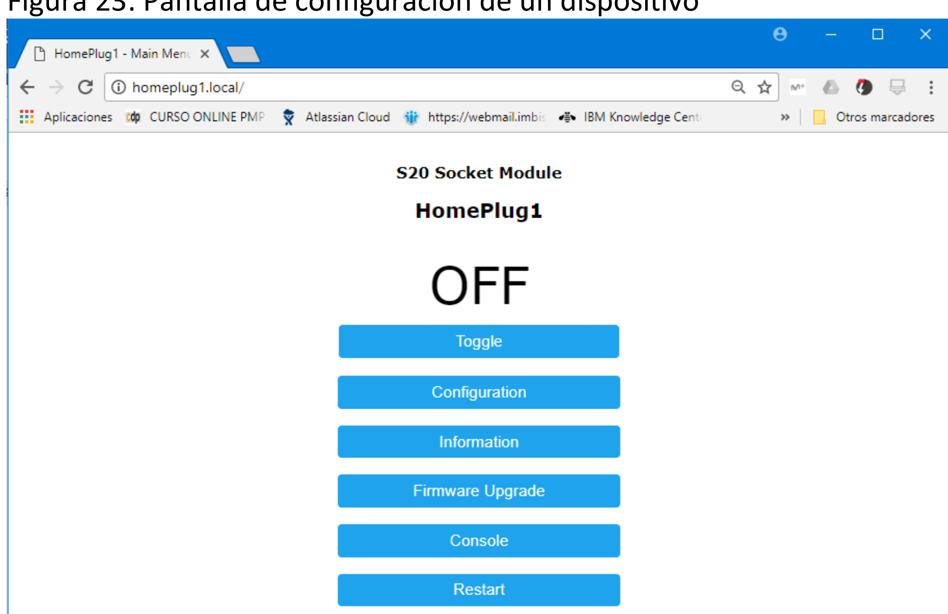
Una vez compilado, como el adaptador USB-TTL es reconocido automáticamente por Lubuntu como '/dev/ttyUSB0' y platform.io lo detecta por defecto, solo hay que cargar el firmware en el dispositivo con la opción 'upload' del menú de Deviot.

Hay que tener cuidado de añadir nuestro usuario al grupo de Linux 'dialout' para evitar recibir un error al escribir en el puerto USB ya que solo este grupo tiene permiso para usar los dispositivos que conectamos a través de USB.

Una vez reprogramados, reiniciamos los dispositivos y estos se conectan automáticamente a la LoT, podemos ver la lista de los nuevos dispositivos en la opción 'Report – My Computers' de nuestro Safe@Office.

Cada dispositivo crea una página web donde podemos afinar su configuración. Accedemos a la página web de cada dispositivo usando el formato '<nombrehost>.local' (Figura 23) y allí acabamos la configuración.

Figura 23: Pantalla de configuración de un dispositivo



2.3.8 Configuración de la Interfaz de usuario

Descargamos la App MQTT-Dash desde el Play Store de Android.

Tras instalar la App, y una vez configurado el bróker MQTT, ya podemos crear controles para nuestros nodos IoT. De momento solo crearemos botones para manejar los relés de cada dispositivo. Buscaremos los tópicos en la wiki del firmware de Tasmota, necesarios para controlar los relés de nuestros nodos con mensajes MQTT.

Tópico	Contenido	Descripción
smarthome/stat/<disp>/POWER	ON / OFF	Obtener estado de dispositivo <disp>

Tópico	Contenido	Descripción
smarthome/cmnd/<disp.>/POWER	ON / OFF	Activar o desactivar dispositivo <disp.>

Si indicamos ‘sonoffs’ en el lugar de <disp.> haremos referencia a todos los dispositivos de nuestra LoT ya que es el grupo que está configurado por defecto en el firmware de Tasmota.

Comprobamos que al pulsar cualquier icono desde la aplicación MQTT-Dash en el teléfono, accionamos los relés de nuestros nodos IoT. En el Anexo 3 de este documento podemos ver la configuración de MQTT-Dash.

2.4 Integración y Programación

Una vez tenemos todos los componentes configurados y conseguido una microCloud funcional, ahora nos podemos dedicar a acabar la integración de los elementos del sistema, eliminar problemas y añadir nuevas funcionalidades. Finalizaremos la configuración de nuestro sistema y aprenderemos a usarlo al programar nuevas funciones con Node-Red.

La programación de las nuevas funcionalidades con los Flujos de Node-Red se pueden consultar en los ficheros adjuntos a este documento que se detallan en el Anexo 5.

2.4.1 Publicar servicios en la CWAN

Primero publicaremos en nuestro Safe@Office todos los servicios públicos de nuestra microCloud. Podremos acceder a esos servicios desde la dirección IP publica ‘172.16.0.253’.

Para las pruebas de publicación de servicios, montamos un nuevo nodo Cloudy llamado ‘Cloudy-Ext1’ en la CWAN, este nodo es externo a nuestra microCloud y allí comprobaremos como se publican los servicios a través de Serf al comunicarse los nodos entre sí.

Creamos el nodo ‘Cloudy-Ext1’ en una máquina virtual con IP ‘172.16.0.203’ que ejecutamos en un portátil conectado en la CWAN. Esta máquina virtual la hemos instalado a partir de una distribución Linux Debian 9 ‘Stretch’ a la cual aplicamos el script ‘cloudynitzar.sh’.

Primero configuramos Serf en ‘SmartHome1’ para publicar los servicios públicos a través de la IP Pública del Safe@Office modificando el fichero ‘/etc/avahi-ps-serf.conf’, tal como se indica el procedimiento que encontramos en la wiki de Guifi.net **[28]**.

```
SERF_RPC_ADDR=127.0.0.1:7373
SERF_BIND=5000
SERF_JOIN=
ADVERTISE_IP=172.16.0.253
```

Modificamos también el fichero '/etc/getinconf-client.conf' para indicar a los nodos externos donde se ubican los servicios públicos mediante la variable 'PUBLIC_IP'.

```
#!/bin/sh
GETINCONF_IGNORE=1
PUBLIC_IP=172.16.0.253
```

Solo queda publicar los servicios de los contenedores Docker desde la interfaz web de Cloudy en la opción 'Enterprise Cloud - Docker' con el botón 'publish'.

En nuestro nodo 'Cloudy-Ext1', configuraremos Serf, indicando como referencia a la IP externa de nuestra microCloud.

```
SERF_RPC_ADDR=127.0.0.1:7373
SERF_BIND=5000
SERF_JOIN=172.16.0.253
ADVERTISE_IP=172.16.0.203
```

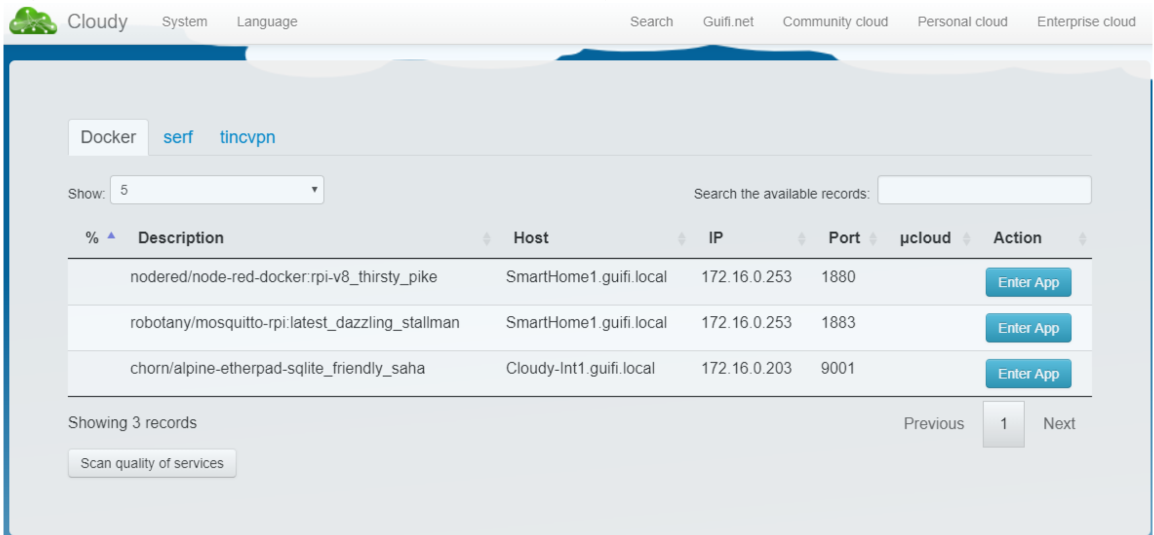
Sin olvidar de modificar el fichero '/etc/getinconf-client.conf' para indicar a la IP publica de 'Cloudy-Ext1'.

```
#!/bin/sh
GETINCONF_IGNORE=1
PUBLIC_IP=172.16.0.203
```

Una vez configurados nuestros dos nodos, podremos hacer búsquedas de los servicios publicados desde la opción 'search' de la interfaz web de Cloudy.

Por ejemplo, al ejecutar 'search' desde 'Cloudy-Ext1', esta muestra correctamente los servicios publicados para cada contenedor (Figura 24) en 'SmartHome1' y 'Cloud-Ext1', donde hemos añadido un contenedor con la aplicación 'Etherpad' como ejemplo de servicio Docker.

Figura 24: Servicios encontrados desde 'Cloudy-Ext1'



2.4.2 Publicar contenedores al reiniciar

Al reiniciar nuestro servidor, comprobamos que los servicios publicados mediante Docker no se publican de nuevo y hay que dar a los botones 'Unpublish' y 'Publish' en el menú de Docker para volver a publicarlos. Es un bug de Cloudy, que reportamos a los desarrolladores, pero haremos el siguiente procedimiento para solucionar el problema temporalmente a la vez que aprendemos a usar Serf.

Instalamos la librería 'docker' de Python para usar la API de Docker.

```
joe@Cloudy-Ex1:~ $ sudo apt-get install python-pip
joe@Cloudy-Ex1:~ $ sudo pip install docker
```

Una vez instalada, crearemos este pequeño script de Python en '/usr/local/Python/regpublic.py'.

```
#!/usr/bin/python
import docker
import json
import subprocess
client = docker.from_env()
for container in client.containers.list():
        if ('public' in container.name):
                cmd='publish'
                tipo= 'Docker'
                strport=container.attrs['Config']['ExposedPorts']
                img=container.attrs['Config']['Image']
                name=container.name[0:container.name.find('_public')]
                ecode=json.dumps(strport)
                dcode=json.loads(ecode)
                prot=dcode.keys()[0]
                dprot=prot.split('/')
                port = dprot[0]
                subprocess.call(['/usr/sbin/avahi-ps', cmd, img + '_' + name,
tipo, port, ""]
```

Finalmente añadiremos esta línea al fichero 'crontab' del usuario 'root' para ejecutar el script en cada arranque:

```
@reboot /usr/bin/python /usr/local/python/regpublic.py
```

2.4.3 Instalar servidor NTP en la LoT

Los dispositivos IoT necesitan obtener la fecha y hora por NTP para sus logs y mediciones periódicas. Como queremos que el único punto de contacto con el exterior sea 'SmartHome1', necesitamos instalar un servidor NTP dentro de nuestra LoT. Instalamos el servidor NTP con el siguiente comando:

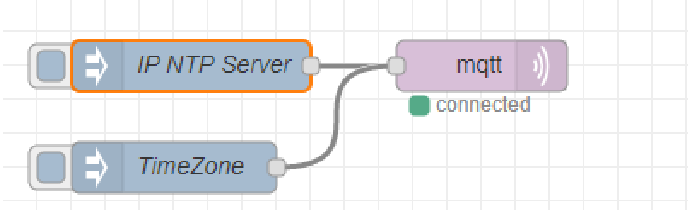
```
joe@SmartHome1:~ $ sudo apt-get update && sudo apt-get -y install ntp
```

Verificamos que el servidor NTP esté funcionando con el comando ‘ntpq’:

```
joe@SmartHome1:~ $ ntpq
ntpq> peers
     remote           refid      st t when poll reach   delay   offset  jitter
==============================================================================
 0.debian.pool.n .POOL.          16 p    -   64    0    0.000    0.000   0.001
 1.debian.pool.n .POOL.          16 p    -   64    0    0.000    0.000   0.001
 2.debian.pool.n .POOL.          16 p    -   64    0    0.000    0.000   0.001
 3.debian.pool.n .POOL.          16 p    -   64    0    0.000    0.000   0.001
-shackleton.red. 140.7.62.122     2 u   43  128  377   19.220    5.101  30.077
-scott.red.uv.es 140.7.62.122     2 u   38  128  377   19.379    4.594   0.966
+bjaaland.red.uv 140.7.62.122     2 u   14  128  377   18.705    5.153   0.792
+elv06.icfo.es   130.206.0.1      2 u   88  128  377   24.303    2.832   0.904
-ns3.intendia.co 150.214.94.10    2 u   36  128  377   11.486    5.418   1.967
*i2t15.i2t.ehu.e .GPS.            1 u   82  128  377   22.976    3.986   0.901
```

Finalmente configuramos el servidor NTP y nuestro huso horario en nuestros nodos creando un Flujo en Node-Red llamado ‘set sonoffs’ (Figura 25) para configurarlos mediante MQTT enviando a los nodos los mensajes con tópicos ‘smarthome/cmnd/sonoffs/NtpServer’ y ‘smarthome/cmnd/sonoffs/TimeZone’.

Figura 25: Flujo ‘set sonoffs’



2.4.4 Actualización automática del firmware

Crearemos un procedimiento automático de actualización de nuestra LoT, gracias a la funcionalidad WebDAV de Cloudy, al protocolo MQTT, la interfaz de usuario en MQTT-Dash y al firmware Tasmota.

Cloudy nos permitirá publicar en nuestra LoT el firmware para actualizar nuestros dispositivos. Tasmota junto a MQTT nos permitirá actualizar los nodos en masa, y solo necesitaremos configurar un botón en nuestra interfaz de usuario para lanzar la actualización a voluntad.

Como ya hemos visto, la carga inicial del firmware en los dispositivos IoT se efectúa mediante una conexión serie, sin embargo, este procedimiento no es práctico una vez que nuestros dispositivos están instalados ya que pueden encontrarse en lugares de difícil acceso.

Tasmota implementa una funcionalidad de actualización inalámbrica. La forma más sencilla para actualizar el firmware es conectarnos a la interfaz web de cada nodo de uno en uno y desde allí, cargar el nuevo firmware manualmente.

Este procedimiento es factible cuando hay pocos nodos, pero tenemos que prever que en un futuro podemos tener decenas de nodos en nuestro sistema, por lo que se hace imprescindible implementar un procedimiento automatizado.

Como primer paso, necesitamos crear un repositorio de firmware usando la funcionalidad WebDAV de Cloudy con el siguiente procedimiento:

- Creamos la carpeta '/etc/apache2/conf.d' desde ssh.
- Vamos a la opción 'Community Cloud' de Cloudy y elegimos la opción 'WebDAV Server'
- En esta pantalla primero creamos un usuario para nuestro WebDAV, introduciendo nombre y contraseña.
- creamos un punto de montaje con el botón 'Mount Point' indicando los siguientes datos:

  Alias Mount: upload
  Mount Point: var/www/upload

Finalmente nos conectamos por SSH y modificaremos el fichero '/etc/apache2/sites-available/000-default.conf' añadiendo la siguiente línea:

```
Include /etc/apache2/conf.d/*.conf
```

Al reiniciar, comprobamos que ahora podemos acceder correctamente a 'http://192.168.10.10/upload' introduciendo el usuario y la contraseña del usuario que definimos anteriormente.

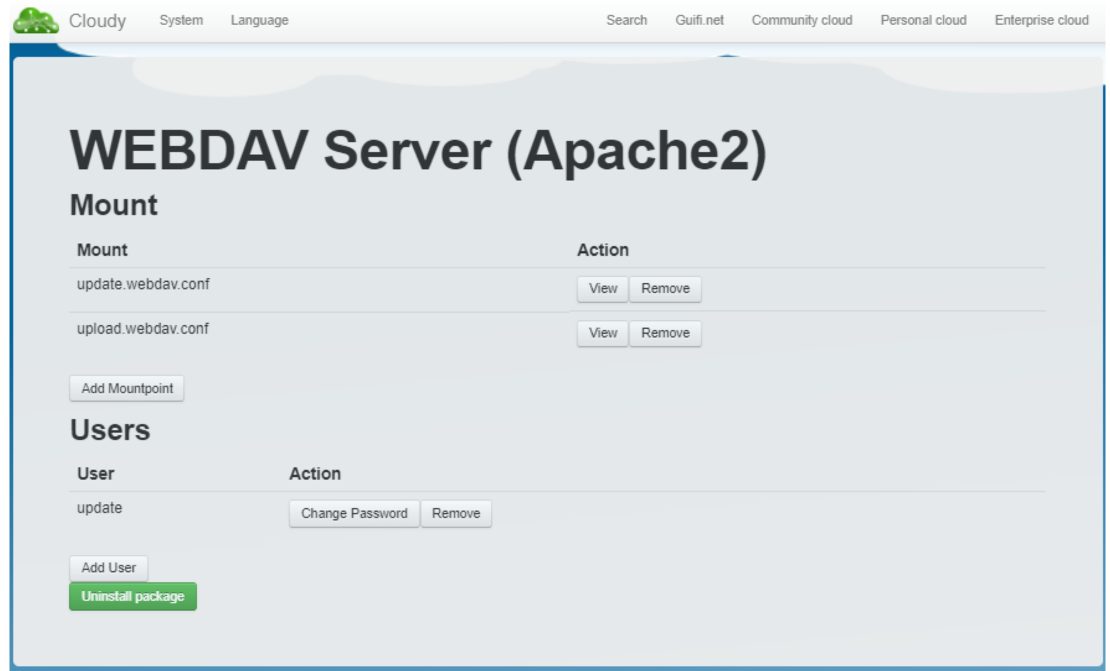
En esta página web cargaremos nuestro firmware con un software cliente WebDAV como 'cadaver' **[29]** o un acceso nativo desde Windows 10 **[30]**.

Sin embargo, nuestros nodos IoT no tienen ninguna opción para definir un usuario para la descarga del firmware por lo que necesitamos crear en la carpeta '/etc/apache2/conf.d' un nuevo fichero llamado 'update.webdav.conf' con la siguiente configuración:

```
Alias /update /var/www/upload
<Location /upload>
    DAV On
    Allow from all
    AuthType Basic
    Options Indexes MultiViews
</Location>
```

Comprobamos los puntos de montaje WebDAV en la interfaz web de Cloudy (Figura 26) y al reiniciar, comprobamos que ahora podemos acceder a nuestro repositorio de firmware ahora desde la página web 'http://192.168.10.10/update' sin necesidad de introducir usuario y contraseña.

Figura 26: Configuración WebDAV



Para configurar esta URL en nuestros dispositivos enviaremos el siguiente mensaje MQTT a todos nuestros sonoffs:

Tópico	Contenido	Descripción
smarthome/cmnd/sonoffs/OtaUrl	http://192.168.10.10	URL de actualización de firmware

A partir de ahora, podemos actualizar el firmware forma automática con el siguiente mensaje MQTT:

Tópico	Contenido	Descripción
smarthome/cmnd/sonoffs/upgrade	1	Lanza actualización de firmware

Para lanzar la actualización cuando queramos, crearemos un botón llamado ‘Update All’ en nuestra interfaz de usuario que enviará el mensaje MQTT ‘upgrade’ a todos los sonoffs del sistema.

2.4.5 Activar o Desactivar todos los dispositivos IoT

En este apartado, gracias al protocolo MQTT, al firmware Tasmota y a la interfaz de usuario que nos proporciona MQTT-Dash, crearemos una funcionalidad en nuestra instalación que nos permitirá activar y desactivar todos nuestros dispositivos sonoffs desde nuestra interfaz de usuario.

Esta funcionalidad es útil por ejemplo para asegurarse que se han apagado todos los dispositivos cuando salimos de casa o, al contrario, encender todos nuestros equipos en caso de alarma.

Añadiremos un 'botón' que llamaremos 'All' en la interfaz de usuario en MQTT-Dash y configuraremos el grupo por defecto en el firmware de todos los dispositivos. El mensaje para apagar o encender todos nuestros dispositivos será el siguiente:

Tópico	Contenido	Descripción
smarthome/cmnd/sonoffs/POWER	ON/OFF	Enciende o apaga todos los dispositivos

2.4.6 Crear una escena de ejemplo

Gracias al protocolo MQTT, a Node-Red y a MQTT-Dash, crearemos una escena sencilla en el que, al pulsar un botón en nuestra interfaz de usuario, encenderemos varios dispositivos simultáneamente, por ejemplo, la radio y una Lampara.

En vez de enviar un comando en un solo sentido, añadiremos retroalimentación de tal manera que la escena tenga confirmación de que todos los dispositivos se han encendido correctamente. Cuando tengamos encendida la radio y la lampara de forma independiente, la escena se activará igualmente.

Desde nuestra interfaz de usuario enviaremos el tópico MQTT 'smarthome/cmnd/Estudio'. Al recibir un mensaje con este tópico, Node-Red enviará comandos a los 2 dispositivos reales y se realimentará de los estados de ambos dispositivos recibidos a través de MQTT.

Una vez implementada la funcionalidad en Node-Red (Figura 27) con un flujo que llamaremos 'Escena Estudio', solo queda crear un nuevo botón llamado 'Estudio' en la interfaz de usuario que activará y desactivará la escena.

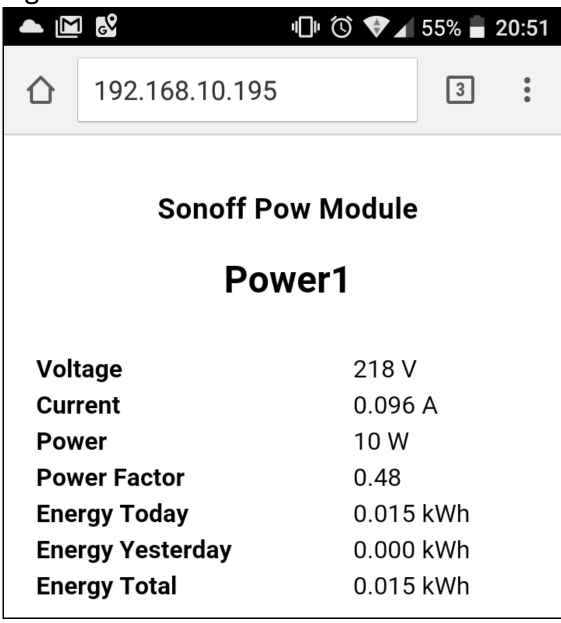
Figura 27: Flujo de 'Escena Estudio'

2.4.7 Medir el consumo de la microCloud

Usaremos el dispositivo 'Power1', Node-Red y MQTT-Dash para mostrar el consumo total de los servidores en nuestra interfaz de usuario. El dispositivo 'Power1' es un Sonoff POW de Itead, que permite medir el consumo eléctrico de todos los equipos que le conectemos.

Tras enchufar nuestros equipos a una regleta controlada por ‘Power1’, podemos conectarnos a la interfaz web de ‘Power1’ (Figura 28) para consultar los consumos eléctricos.

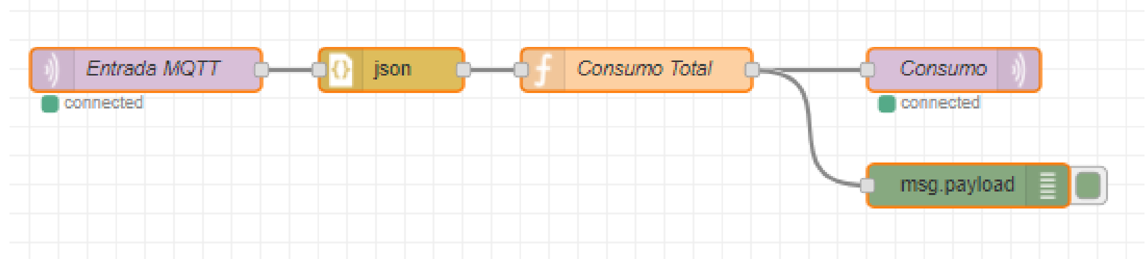
Figura 28: Interfaz web de Power1



Como el Firmware de Tasmota envía un mensaje MQTT con el tópico ‘smarthome/tele/power1/SENSOR’ cada 5 minutos con los datos de consumo energético, configuraremos en Node-Red un flujo que escuchará las actualizaciones de telemetría de ‘Power1’ que llamaremos ‘Consumo microCloud’ (Figura 29).

Una vez recibido el mensaje, extraerá el campo “Energy Total” y nos actualizará nuestra interfaz de usuario con un mensaje MQTT especifico un control que llamaremos ‘Consumo microCloud kWh’

Figura 29: Flujo ‘Consumo microCloud’



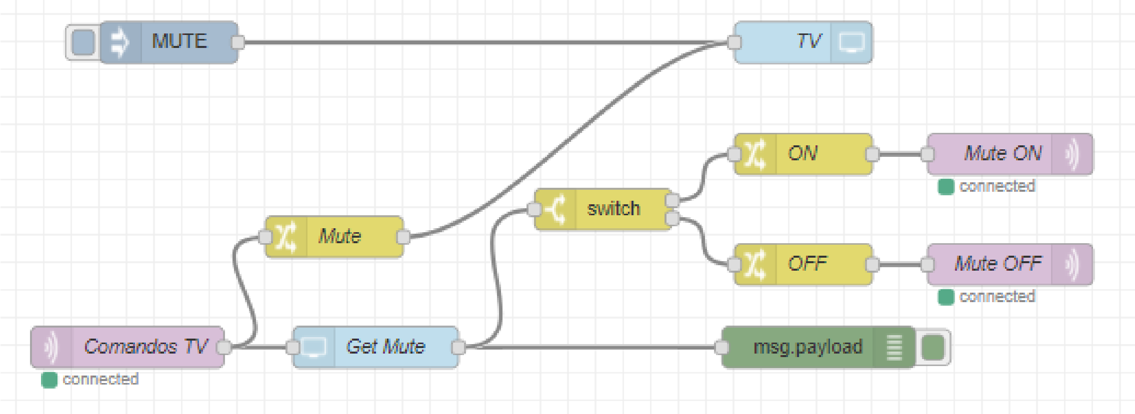
2.4.8 Añadir botón de ‘mute’ en la interfaz de usuario

Tras comprobar que existe un nodo Node-Red llamado ‘node-red-contrib-viera’ **[31]** para controlar la televisión del salón (Panasonic Viera) por Wifi, añadiremos un control llamado ‘Mute’ en MQTT-Dash que efectúe la función ‘mute’ desde nuestra interfaz de usuario.

Esto nos permitirá por ejemplo apagar el sonido de la TV desde Node-Red cuando lanzamos un mensaje de voz desde un altavoz.

La comunicación entre Node-Red y MQTT-Dash se efectuará con el protocolo MQTT y toda la lógica se implementará en Node-Red en un flujo llamado 'TV Mute' (Figura 30). Se obtendrá retroalimentación de la TV para confirmar que la función se ha ejecutado correctamente.

Figura 30: Flujo 'TV Mute'



Configuraremos un botón 'Mute' en MQTT-Dash para poder ejecutar esta función desde nuestra interfaz de usuario.

2.4.9 Añadir un sensor de temperatura a nuestro sistema

En este apartado se pretende instalar un sensor de temperatura (Figura 31) en nuestro sistema y publicarlo en nuestra interfaz de usuario con MQTT.

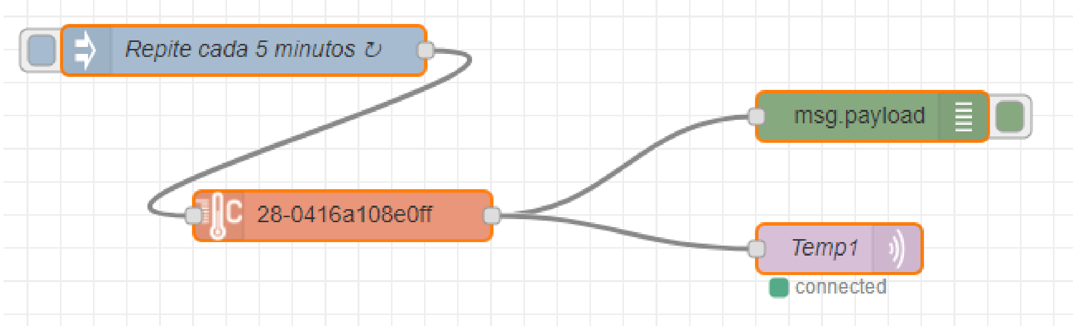
Figura 31: Sensor de temperatura

El sensor disponible es un DS18b20 que Keyes comercializa en un pequeño circuito impreso que incluye la resistencia pull-up para conectarlo fácilmente a nuestros dispositivos.

Para evitarnos soldar, conectaré la placa a los pines GPIO de la Raspberry Pi con unos cables Dupont. Instalaremos un nodo predefinido llamado 'node-red-contrib-sensor-ds18b20' **[32]** para leer este tipo de sensores.

Finalmente crearemos un nuevo flujo en Node-Red que llamaremos 'Temperatura' (Figura 32) con el que enviaremos periódicamente las lecturas de temperatura del sensor a la interfaz de usuario en MQTT-Dash.

Figura 32: Flujo 'Temperatura'



2.5 Conclusión de la fase 1

Hemos creado un sistema completamente funcional y con prestaciones profesionales tales como actualizaciones inalámbricas y scripting avanzado. Esta solución no tiene los inconvenientes de las soluciones alojadas en nube que nos proporcionan los fabricantes de soluciones comerciales.

Hemos comprobado que los dispositivos Wifi basados en SoC ESP8266 reprogramados con firmware abierto son perfectamente usables en un entorno doméstico y no son difíciles de reprogramar, incluso sin soldadura.

La integración de una suma de proyectos y tecnologías abiertas (Hardware, Firmware, Software, protocolos y entorno de programación) nos ha permitido diseñar e implementar una microCloud genérica que se ejecuta en nuestra LAN y que nos permite gestionar nuestros nodos IoT localmente.

La simplicidad y sencillez de uso del protocolo MQTT nos ha permitido entender y desarrollar nuestra solución rápidamente y nos ha permitido unir todos los componentes de nuestro sistema de una forma transparente junto con la herramienta de programación Node-Red.

Con MQTT y Node-Red hemos observado una capacidad de integración sobresaliente y una gran facilidad para resolver nuestros problemas y crear rápidamente aplicaciones que se ajustan a nuestras necesidades.

No nos hemos encontrado con ningún inconveniente a destacar durante el proyecto y todos los pasos se han efectuado siguiendo la planificación, cumpliendo los plazos prefijados.

3 Segunda fase: microCloud Avanzada

En esta segunda fase, analizaremos la infraestructura obtenida en la fase anterior y modificaremos nuestro diseño para obtener una microCloud tolerante a fallos que proporcione los mismos servicios, pero en alta disponibilidad.

Una vez implementado y probado nuestro sistema, optimizaremos el uso de los recursos empleados mediante la funcionalidad Swarm de Docker y estudiaremos como extender nuestra infraestructura a distintas ubicaciones.

3.1 Análisis de la microCloud de la Fase 1

Analizaremos la arquitectura lógica de la microCloud desarrollada en la primera fase para preservar su funcionalidad y estudiaremos los puntos únicos de fallo de la arquitectura física para eliminarlos y aumentar la redundancia del sistema.

3.1.1 Arquitectura Lógica

Como podemos ver en la arquitectura lógica de nuestra microCloud (Figura 33) el bróker de mensajes MQTT, Mosquitto, es el componente que vertebra todas las comunicaciones entre todos los elementos de nuestra red doméstica

Figura 33: Arquitectura lógica de la microCloud

El enrutador Safe@Office se encarga de redireccionar las peticiones externas al S.O. de la microCloud para el anuncio de servicios y los servicios predefinidos, pero el protocolo MQTT se redirecciona a Mosquitto y el resto de los protocolos TCP/IP a Node-Red o a los servicios predefinidos de Cloudy.

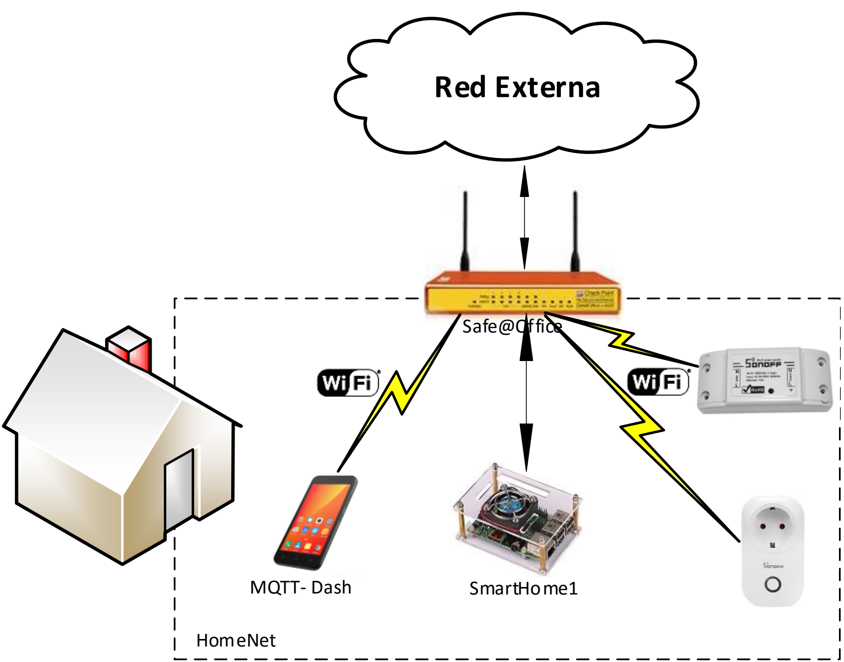
Hay que volver a destacar que el único componente que necesitan conocer los dispositivos participantes en la red MQTT es al propio Bróker MQTT porque en el protocolo MQTT todas las comunicaciones se efectúan a través de esta indirección.

De hecho, para incluir un nuevo componente en nuestra red MQTT, solo necesitamos configurarle la dirección del bróker MQTT y asegurarnos que pueda obtener automáticamente una dirección IP en la wifi con DHCP.

3.1.2 Arquitectura Física

Esta es la arquitectura física final de la microCloud que desarrollamos en la primera fase del proyecto (Figura 34). Nos centraremos en buscar los componentes que pueden provocar una pérdida total del servicio (puntos únicos de fallo).

Figura 34: Arquitectura física de la microCloud de la fase 1



El servicio de Bróker MQTT y la lógica de la microCloud en nuestra red doméstica son proporcionados por el software que se ejecuta en la Raspberry pi, por lo que el SBC es inequívocamente un punto único de fallo obvio.

Otro punto crítico cuyo fallo provocaría un malfuncionamiento total del sistema es la red Wifi que comunica todos los clientes de nuestra microCloud. Una red inalámbrica es mucho más vulnerable que una red cableada ya que es un medio compartido que puede estar sujeto a interferencias y saturación.

Puntos de fallo menos graves son el enrutador en sí y la conexión a la red externa. En este caso, un fallo en esos componentes solo afectaría a los servicios publicados fuera de nuestra red local. Este caso sería un fallo menor, porque nuestra prioridad es proporcionar servicio a los usuarios internos.

3.2 Diseño de una microCloud Tolerante a fallos

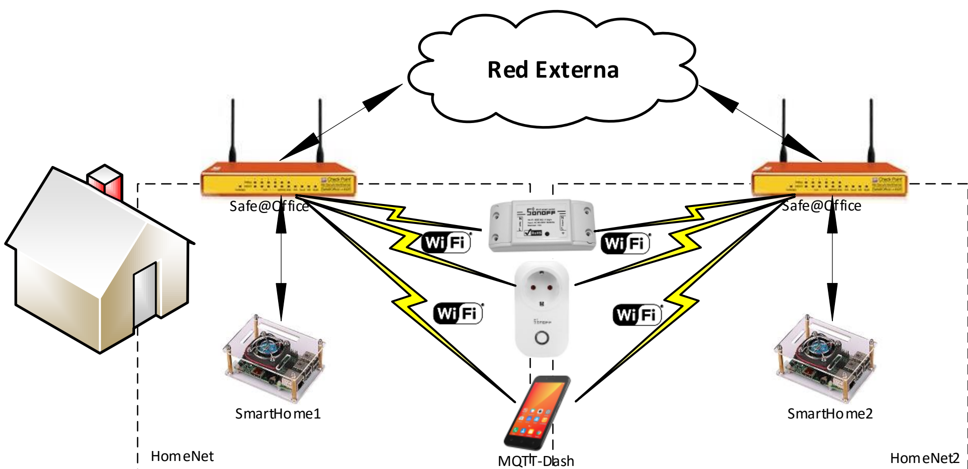
Tal como hemos comentado anteriormente, nuestro diseño buscará preservar la arquitectura lógica al mismo tiempo que eliminamos los puntos únicos de fallo. Para eliminar esos puntos de fallo, añadiremos redundancia en el hardware y en el software de nuestro diseño original.

3.2.1 Redundancia de Hardware y redes

El firmware Tasmota de los dispositivos Sonoffs permite especificar una red wifi de respaldo y el software MQTT-Dash al ejecutarse en el móvil, es capaz de conectarse a esa red de respaldo en caso de pérdida de la red principal.

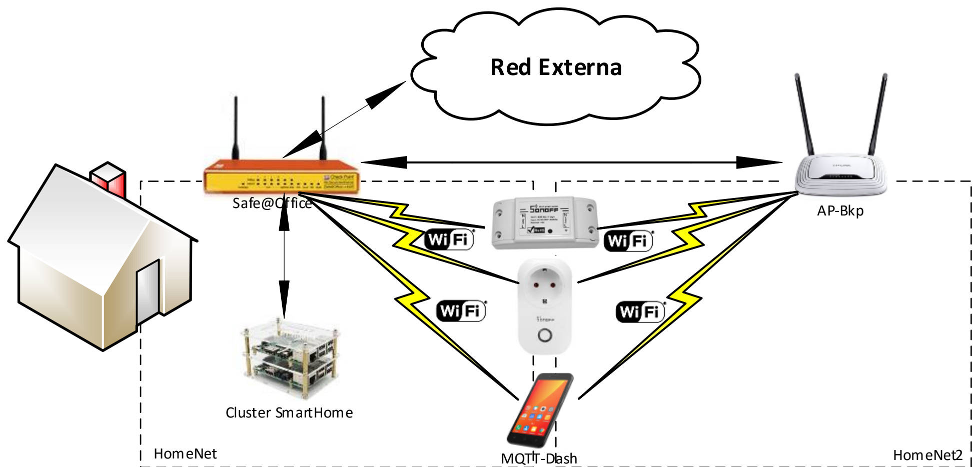
El diseño más obvio para aumentar la disponibilidad de nuestro sistema es redundar completamente todos los componentes físicos de la microCloud (Figura 35), de esta manera tendremos un reemplazo para cualquier elemento.

Figura 35: MicroCloud completamente redundada



Pero desgraciadamente, como en un entorno doméstico tenemos recursos limitados, y no solemos disponer de conexiones a redes externas redundadas con enrutadores activos, buscaremos una solución de compromiso (Figura 36).

Figura 36: Solución de compromiso de microCloud redundada



En esta última propuesta tendremos una única conexión con la red externa desde nuestro enrutador Safe@Office donde también conectaremos las dos Raspberry pi 3 que formaran parte de nuestro sistema .

Para proporcionar la red wifi de respaldo, en vez de usar un segundo enrutador usaremos un punto de acceso Wifi (AP-Bkp), mucho más sencillo y barato, que estará conectado mediante un cable de red al Safe@Office.

El enrutador seguirá siendo un punto único de fallo, pero en este caso solo tendremos un fallo total del sistema en caso de avería de la LAN cableada que proporciona el Safe@Office a todos los componentes conectados por cable.

Modificaremos la configuración de red de la fase 1 añadiendo la nueva red Wifi de respaldo que llamaremos 'HomeNet2' y un nuevo nodo Cloudy que llamaremos 'SmartHome2'. También añadiremos y configuraremos el nuevo punto de acceso wifi que llamaremos 'AP-Bkp'.

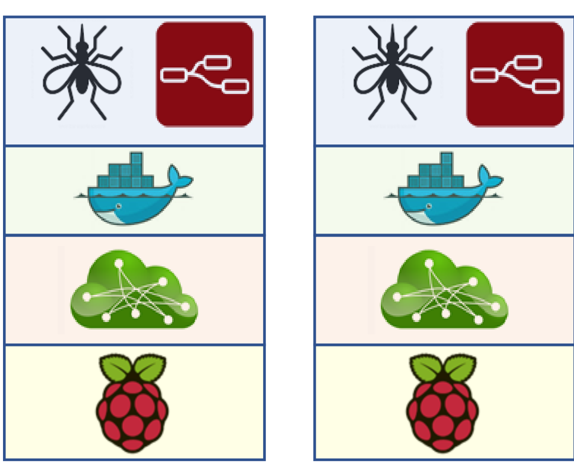
Figura 37: Direccionamiento de los equipos en la Fase 2

Descripción	Dirección IP	Mascara	Gateway	IP Publica
Homenet	192.168.10.1xx	255.255.255.0	192.168.10.1	
Homenet2	**192.168.20.1xx**	**255.255.255.0**	**192.168.20.1**	
SmartHome1	192.168.10.10	255.255.255.0	192.168.10.1	172.16.0.253
SmartHome2	**192.168.10.11**	**255.255.255.0**	**192.168.10.1**	**172.16.0.252**
AP-Bkp LAN	**192.168.10.2**	**255.255.255.0**	**192.168.10.1**	
AP-Bkp WLAN	**192.168.20.1**	**255.255.255.0**		
Safe@Office WAN	172.16.0.254	255.255.255.0	172.16.0.1	172.16.0.254
Safe@Office LoT	192.168.10.1	255.255.255.0		
Router Internet	172.16.0.1	255.255.255.0		

3.2.2 Redundancia de Software

La solución más obvia para ofrecer redundancia de software es crear un nuevo nodo Cloudy independiente que llamaremos 'SmartHome2' con exactamente el mismo software y la misma configuración que 'SmartHome1' (Figura 38).

Figura 38: Nodos independientes Docker



3.2.2.1 Failover Manual

Si configuramos los dos nodos Cloudy de forma completamente idéntica, solo podemos tener en funcionamiento uno de los nodos mientras el segundo está apagado, ya que no pueden existir 2 nodos con la misma IP en una LAN.

En caso de avería, apagaremos el nodo que falla y arrancaremos el nodo de respaldo. Los clientes seguirán conectándose en la misma dirección IP del bróker MQTT que tienen en su configuración, siendo el cambio de equipo totalmente transparente para todos los componentes de la red.

El principal inconveniente de este enfoque es que el proceso de parada y puesta en marcha se tiene que efectuar de forma manual ya que no existe en nuestro sistema ninguna forma sencilla de encender o apagar un nodo automáticamente. Además, de esta manera estamos desaprovechando todos los recursos del nodo de respaldo.

Para resolver este problema, podemos cambiar únicamente la configuración de red del nodo de respaldo con el objetivo de poder tener los dos nodos encendidos simultáneamente. El problema que surge entonces es que nuestros clientes MQTT solo pueden tener un único bróker de mensajes configurado.

Para utilizar nuestro nodo de respaldo, deberemos o bien cambiar la configuración de la dirección IP del bróker MQTT en todos los clientes, o bien intercambiar las direcciones IP entre los dos nodos, operaciones que son conveniente efectuar automáticamente cada vez que tengamos una avería.

Para intercambiar la dirección IP a la que apuntan los clientes entre los nodos Cloudy automáticamente, podemos usar una técnica llamada IP Failover.

3.2.2.2 IP Failover

Con IP Failover, cada nodo mantiene su dirección IP privada, pero además comparte una dirección IP flotante (Figura 39). Esta IP flotante apunta únicamente a uno de los nodos, pero en caso de fallo se cambia automáticamente al nodo superviviente. De esta manera nuestras máquinas trabajan de manera conjunta. Esta configuración de equipos se llama clúster.

Figura 39: IP failover en clúster de 2 nodos

Si cambiamos la configuración de los clientes MQTT y de nuestro enrutador para que apunten a la IP flotante de nuestro clúster, los clientes podrán acceder a nuestra microCloud incluso en caso de pérdida de uno de los nodos.

Para implementar IP failover podemos usar una familia de protocolos llamados FHRP (First Hop Redundancy Protocols). Estos protocolos se suelen usar en enrutadores para proteger el Gateway por defecto de una red, al permitir compartir una misma IP entre varios equipos.

Algunos ejemplos de FHRP son HSRP (Hot Standby Redundancy Protocol) **[33]**, VRRP (Virtual Router Redundancy Protocol) **[34]** y CARP (Common Address Redundancy Protocol) **[35]**. HSRP es un protocolo propietario de Cisco. VRRP es un estándar IETF y CARP es un protocolo libre.

En Debian podemos usar VRRP con el software keepalived o CARP con el software uCARP. Para nuestra implementación nos decantaremos por keepalived **[36]**, que ya está incluido en los paquetes de Raspbian.

3.3 Implementación de la microCloud Tolerante a fallos

Una vez finalizado nuestro diseño tolerante a fallos, pasaremos a efectuar la implementación: añadiremos la red Wifi de respaldo 'HomeNet2', crearemos el segundo nodo Cloudy y finalmente instalaremos IP Failover con keepalived.

3.3.1 Añadir Wifi de respaldo

Configuraremos la red inalámbrica de nuestro punto de acceso 'AP-Bkp' con el direccionamiento '192.168.20.0/24', el SSID 'HomeNet2' y configuraremos su puerto LAN con la dirección IP '192.168.10.2/24'. Crearemos en nuestro AP una ruta por defecto que apunte a la IP '192.168.10.1' del Safe@Office.

En nuestro Safe@Office, conectaremos nuestro punto de acceso a uno de los puertos LAN libres y añadiremos la ruta a la red '192.168.20.0/24' con Gateway por defecto '192.168.10.2' para acceder a la nueva red inalámbrica.

Para configurar esta Wifi a todos nuestros nodos IoT ya instalados, enviaremos a los nodos IoT los siguientes mensajes MQTT:

Tópico	Contenido	Descripción
smarthome/cmnd/sonoffs/SSid2	HomeNet2	SSID de Wifi de Respaldo
smarthome/cmnd/sonoffs/Password2	<passwd>	Password de Wifi de Respaldo

Y modificaremos la configuración de la Wifi de respaldo en el código fuente del firmware Tasmota en el fichero 'user_config.h'.

```
57 #define STA_SSID2      "HomeNet2"  // [Ssid2] Optional alternate AP Wifi SSID
58 #define STA_PASS2      "xxxxxxxx"  // [Password2] Optional alternate AP Wifi password
```

3.3.2 Creación del segundo nodo Cloudy

Añadiremos un nuevo nodo Cloudy a la microCloud que llamaremos 'SmartHome2'. Seguiremos los mismos procedimientos que utilizamos en el capítulo 2 para instalar el hardware y el software de nuestro primer nodo.

3.3.2.1 Hardware

Para optimizar el espacio ocupado, apilaremos una nueva Raspberry pi con una segunda caja acrílica (Figura 40) y la conectaremos a uno de los puertos LAN libres de nuestro enrutador Safe@Office.

Usaremos un nuevo cable microUSB para conectar la alimentación de la Raspberry a nuestro cargador de alta potencia. Crearemos e insertaremos una nueva tarjeta microSD con una instalación nueva de Raspbian.

Figura 40: Pila de 2 Raspberry pi

3.3.2.2 Sistema Operativo y redes

Cambiaremos el nombre del nodo a 'SmartHome2' con 'raspi-config' y asignaremos a nuestra Raspberry el direccionamiento de red que tenemos definido en la tabla de redes de la fase 2 (Figura 37), configuraremos la dirección IP de forma estática en el fichero '/etc/dhcpcd.conf'.

```
# Example static IP configuration:
interface eth0
static ip_address=192.168.10.11/24
static routers=192.168.10.1
static domain_name_servers=192.168.10.1 8.8.8.8 fd51:42f8:caae:d92e::1
```

Transformaremos Raspbian en Cloudy con el script 'Cloudytnizar.sh' y comprobaremos que Cloudy está instalado correctamente accediendo a su interfaz web a través la IP '192.168.10.11' en el puerto 7000.

Configuraremos 'SmartHome2' en nuestro Safe@Office basándonos en la configuración que efectuamos para 'SmartHome1' en la primera fase. Crearemos las reglas que estimemos necesarias para acceder a los servicios externos del nodo desde su IP pública. La configuración del Safe@Office para esta segunda fase se puede consultar en el Anexo 2.

3.3.2.3 Software

Pasaremos a instalar y activar los servicios predefinidos en nuestro nuevo nodo Cloudy siguiendo los pasos del capítulo 2, instalaremos Docker, Serf y los contenedores ‘Node-Red’ y ‘Mosquitto’ necesarios para nuestra microCloud.

Modificamos el fichero ‘/etc/avahi-ps-serf.conf’ en ‘SmartHome2’ para que Serf anuncie los servicios del nodo a través de su IP pública en el Safe@Office.

```
SERF_RPC_ADDR=127.0.0.1:7373
SERF_BIND=5000
SERF_JOIN=
ADVERTISE_IP=172.16.0.252
```

Modificamos también el fichero ‘/etc/getinconf-client.conf’ para indicar a los nodos externos la dirección IP donde publicaremos los servicios.

```
#!/bin/sh
GETINCONF_IGNORE=1
PUBLIC_IP=172.16.0.252
```

Una vez configurado Serf y lanzados los contenedores, podemos anunciar los servicios que estemos ejecutando desde la interfaz web de Cloudy en la opción ‘Enterprise Cloud - Docker’ pulsando el botón ‘publish’.

3.3.2.4 Replicación manual de la configuración

Nuestro recién creado contenedor ‘Node-Red’ en el nodo ‘SmartHome2’ está en blanco y no tiene ningún flujo creado. Para que asuma las mismas funciones que ‘SmartHome1’, procederemos a replicar todos los flujos existentes en ‘SmartHome2’ con el siguiente procedimiento:

- Nos conectamos a Node-Red en el puerto 1880 de ‘SmartHome1’ y desde el menú elegimos la opción ‘Exportar - Clipboard’ y exportaremos todos los flujos.

- Nos conectamos a Node-Red en el puerto 1880 de ‘SmartHome2’ y desde el menú elegimos la opción ‘Importar - Clipboard’ y pegaremos todos los flujos.

- Deberemos añadir los nodos que no vienen instalados por defecto desde la opción ‘Manage Palette’ del menú.

- Cambiaremos la configuración de la IP de nuestro bróker MQTT a ‘192.168.10.11’ en los flujos.

- Pulsaremos ‘Deploy’ para ejecutar nuestros flujos en el nuevo nodo.

Una vez duplicada la configuración de Node-Red, nuestro nodo de respaldo tendrá exactamente la misma funcionalidad que el nodo ‘SmartHome1’ por lo que podremos intercambiarlos en cuanto lo estimemos necesario.

3.3.3 Implementación del clúster

Instalaremos los paquetes de software ‘keeapalived’ y ‘mosquitto-clients’. El paquete ‘mosquitto-clients’ es un cliente MQTT en modo texto que nos permitirá crear un script para comprobar desde keepalived que los servicios de nuestra microCloud estén funcionando correctamente.

```
$ sudo apt-get install keepalived mosquitto-clients
```

3.3.3.1 Configuración de keepalived

Elegiremos la dirección IP ‘192.168.10.15’ como dirección IP flotante de nuestro clúster para IP Failover y crearemos el fichero de configuración ‘/etc/keepalived/keepalived.conf’ en el nodo ‘SmartHome1’.

```
! Configuration File for keepalived in SmartHome1

vrrp_script chk_mqtt {
  script       "/usr/local/bin/chkmqtt.sh"
  interval 10  # check every 10 seconds
  fall 2       # require 2 failures for KO
  rise 2       # require 2 successes for OK
}

vrrp_instance VI_1 {
    state MASTER
    interface eth0
    virtual_router_id 51
    priority 150
    advert_int 1
    authentication {
        auth_type PASS
        auth_pass a7bH@4tz
    }
    virtual_ipaddress {
        192.168.10.15
    }
  track_script {
        chk_mqtt
    }
}
```

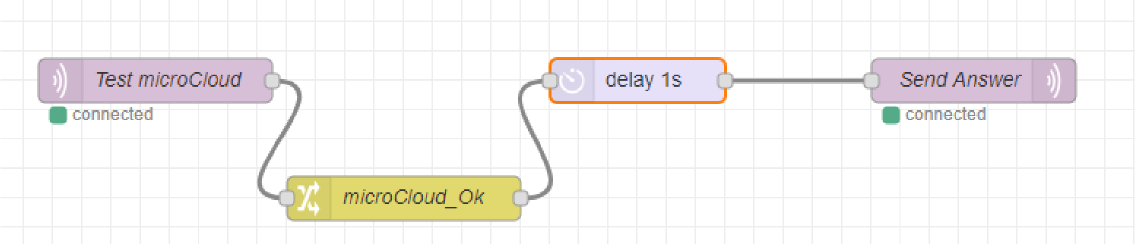
Crearemos otro script llamado ‘/usr/local/bin/chkmqtt.sh’ que servirá para testear la microCloud a nivel de servicios **[37]**.

```
#/bin/sh
mosquitto_pub -h 192.168.10.10 -t smarthome/test/microCloud -m "Test"
timeout --foreground 5 mosquitto_sub -h 192.168.10.10 -t smarthome/test/answe
r -C 1 | grep -v microCloud_Ok || exit 0
echo "Error"
exit 1
```

En este script, enviaremos un mensaje con tópico 'Test' al bróker MQTT y esperaremos la respuesta correcta desde Node-Red, dando la prueba por fallida si no recibimos respuesta en 5 segundos.

Crearemos el flujo 'Test Health' (Figura 41) en Node-Red para responder a los mensajes MQTT enviados por 'chkmqtt.sh'. De esta manera testearemos de una vez los contenedores 'Mosquitto' y 'Node-Red' de nuestra microCloud.

Figura 41: Flujo 'Test Health'



Solo falta crear otro fichero de configuración '/etc/keepalived/keepalived.conf' en 'SmartHome2', teniendo cuidado de indicar una prioridad más baja que 'SmartHome1'. En este caso el nodo secundario no necesita chequear nada.

```
! Configuration File for keepalived in SmartHome2

vrrp_instance VI_1 {
    state MASTER
    interface eth0
    virtual_router_id 51
    priority 100
    advert_int 1
    authentication {
        auth_type PASS
        auth_pass a7bH@4tz
    }
    virtual_ipaddress {
        192.168.10.15
    }
}
```

Y lanzamos el servicio en ambos nodos Cloudy con el siguiente comando.

```
$ sudo service keepalived start
```

La IP flotante '192.168.10.15' se asigna en un principio al nodo de mayor prioridad, 'SmartHome1' y en caso de fallo total o cuando la microCloud no responda correctamente al script 'chkmqtt.sh', la IP flotante cambiará automáticamente al nodo de menor prioridad, en este caso 'SmartHome2'.

Para probar keepalived, ejecutaremos el comando 'ping 192.168.10.15' desde 'SmartHome2' y comprobaremos cómo se mantiene el 'ping' aunque reiniciemos 'SmartHome1' y como cambia la latencia al moverse la IP de un nodo al otro cuando paramos los servios MQTT o Node-Red en SmartHome1.

3.3.3.2 Configuración de la IP flotante en los clientes

Una vez configurados nuestros servidores, solo nos queda configurar la IP Flotante como la dirección del bróker de mensajes en todos los clientes MQTT.

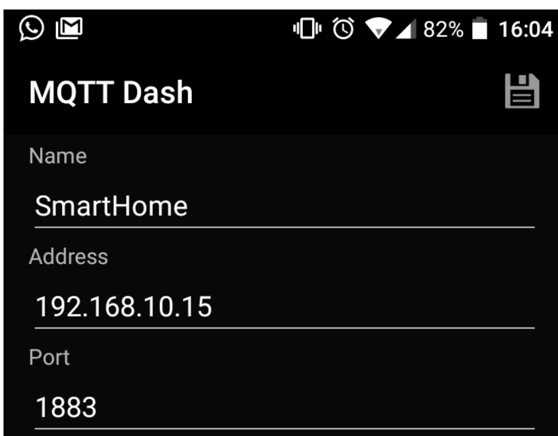
Actualizaremos simultáneamente todos los dispositivos Sonoffs de nuestra instalación enviándoles el siguiente mensaje MQTT:

Tópico	Contenido	Descripción
smarthome/cmnd/sonoffs/MqttHost	192.168.10.15	Dirección del Bróker MQTT

Y cambiaremos la configuración en 'user_config.h' del firmware Tasmota para que los futuros dispositivos a los que carguemos este Firmware usen esta IP.

```
84   #define MQTT_HOST            "192.168.10.15"          // [MqttHost]
```

Finalmente, solo nos queda modificar la configuración del bróker MQTT en la interfaz de usuario de la microCloud en la APP MQTT-Dash.



3.3.3.3 Publicación de servicios en dirección IP alternativa.

Seguiremos publicando nuestros dos nodos Cloudy para quien quiera usar un balanceador de carga externo o acceder a los servicios de forma independiente, Pero, a su vez publicaremos los servicios redundados a los usuarios externos en la dirección IP '172.16.0.250' con la descripción 'Cluster'.

Para publicar los servicios en Serf en esta IP alternativa, que no pertenece directamente a ninguno de nuestros dos nodos, tendremos que modificar algunos scripts y ficheros de configuración de Cloudy.

Empezaremos añadiendo al fichero '/etc/getinconf-client.conf' la siguiente línea para definir una dirección IP alternativa.

```
PUBLIC_ALT_IP = 172.16.0.250
```

Modificaremos la variable ‘actions’ en el script ‘/usr/sbin/avahi-ps’ para añadir la acción ‘publish_alt’, esta nueva acción nos permitirá publicar con Serf los servicios en la IP alternativa que definimos en el fichero anterior.

```
# Available actions
actions="publish publish_alt search unpublish"
```

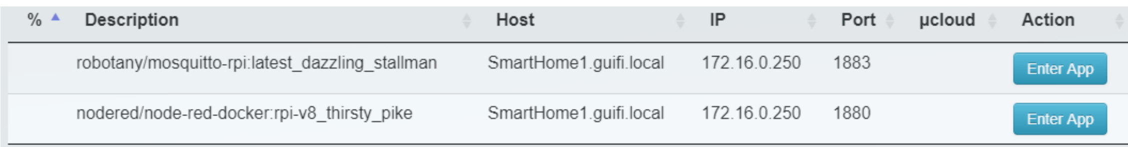
Con las funciones ‘serf_publish_alt_service()’ y ‘serf_add_service_alt()’ en el fichero ‘/usr/share/avahi-ps/plugs/avahi-ps-serf’ implementaremos el código que nos permitirá usar la IP alternativa para los servicios publicados en Serf.

```
serf_publish_alt_service() {

        local _DESCRIBE
        local _TYPE
        local _PORT
        local _TXT

        if [ $# -lt 3 ]
        then
                avahi-ps-help
        fi

        _DESCRIBE="$1"
        _TYPE="$2"
        _PORT="$3"
        _TXT=${4:-""}
        serf_add_service_alt "$_DESCRIBE" "$_TYPE" "$_PORT" $_TXT
}

serf_add_service_alt(){

        local _TXT
        local Service
        local Desc

        _TXT=$(echo "$4"|tr "&" " ")
        Service=$2
        Desc=$(echo $1|sed 's/ /_/g')

        setToMyServices  "$Service"  "$Desc"  "${NODENAME}.${CLOUD_NAME}.local"
"${PUBLIC_ALT_IP}" "$3" "$DEVICE_NAME" "$_TXT"
}
```

Y finalmente, modificaremos el fichero ‘/usr/local/python/regpublic.py’ para que, al reiniciar, si encuentra un contenedor Docker con la cadena ‘cluster’ en su nombre, este se publique con el nuevo mecanismo que hemos implementado.

```
                                If(‘cluster’ in container.name):
                                        cmd='publish_alt'
                                        tipo='Cluster'
                                else:
                                        cmd='publish'
                                        tipo='Docker'
```

Una vez reiniciado el nodo, verificamos como a partir de ahora, Serf publica los servicios redundados de 'SmartHome1' en la dirección IP '172.16.0.250'.

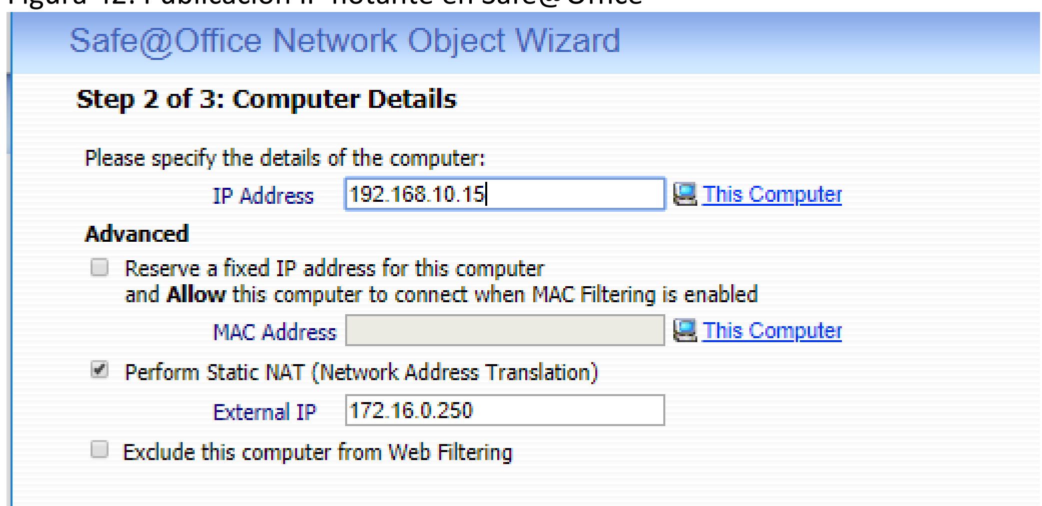


%	Description	Host	IP	Port	µcloud	Action
	robotany/mosquitto-rpi:latest_dazzling_stallman	SmartHome1.guifi.local	172.16.0.250	1883		Enter App
	nodered/node-red-docker:rpi-v8_thirsty_pike	SmartHome1.guifi.local	172.16.0.250	1880		Enter App

Efectuaremos el mismo procedimiento en el nodo 'SmartHome2' para publicar los mismos servicios con la misma IP alternativa, de esta manera los servicios de la microCloud serán publicados por ambos nodos, de manera redundante.

Solo nos queda configurar en el Safe@Office la IP flotante como 'SmartHome' con dirección publica '172.16.0.250' (Figura 42) con el objeto de facilitar el acceso a los servicios redundados del clúster a los clientes externos.

Figura 42: Publicación IP flotante en Safe@Office



Crearemos las reglas para que accedan a los servicios (Figura 43).

Figura 43: Reglas para publicar servicios redundados

13				Allow	WAN (Internet)	SmartHome (Network Object):1880 (TCP)
14				Allow	WAN (Internet)	SmartHome (Network Object):1883 (TCP)

La configuración final de nuestros equipos de red será la siguiente:

Figura 44: Configuración de equipos de red de la Fase 2

Descripción	Dirección IP	Mascara	Gateway	IP Publica
Homenet1	192.168.10.1xx	255.255.255.0	192.168.10.1	
Homenet2	192.168.20.1xx	255.255.255.0	192.168.20.1	
SmartHome	**192.168.10.15**	**255.255.255.0**	**192.168.10.1**	**172.16.0.250**
SmartHome1	192.168.10.10	255.255.255.0	192.168.10.1	172.16.0.253
SmartHome2	192.168.10.11	255.255.255.0	192.168.10.1	172.16.0.252
AP LAN	192.168.10.2	255.255.255.0	192.168.10.1	
AP WLAN	192.168.20.1	255.255.255.0		
Safe@Office WAN	172.16.0.254	255.255.255.0	172.16.0.1	172.16.0.254
Safe@Office LoT	192.168.10.1	255.255.255.0		
Router Internet	172.16.0.1	255.255.255.0		

3.4 Optimización de recursos del clúster

El clúster implementado en los apartados anteriores dista mucho de ser eficiente ya que duplica los recursos necesarios para ejecutar los servicios de la microCloud de forma redundante y nos obliga a hacer el doble de trabajo para su mantenimiento.

Esto es inaceptable en un entorno doméstico con recursos limitados como el nuestro. Para optimizar el uso de recursos de nuestra instalación utilizaremos las funcionalidades incluidas en el modo Swarm de Docker..

3.4.1 Docker Swarm

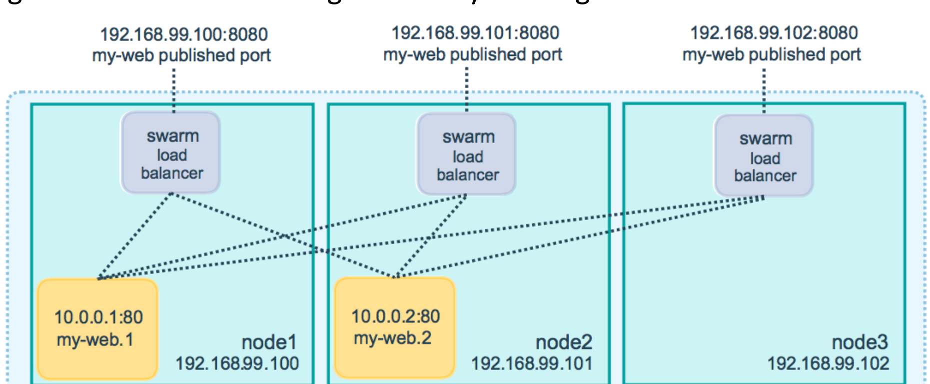
Desde la versión 1.12, el gestor de contenedores Docker incluye el modo Swarm, que permite gestionar de forma nativa un clúster de nodos Docker.

Docker Swarm nos proporciona una capa adicional de abstracción que nos permite definir servicios escalables que son ejecutados y publicados por el conjunto de los nodos Docker del clúster. Swarm gestiona los contenedores en cada nodo para cumplir con la configuración de servicios definida previamente.

Estas son las características de Swarm que nos interesan especialmente:

- Integración: está integrado en Docker, y se gestiona con la línea de comando de Docker sin necesidad de instalar ningún software adicional.
- Escalado: se puede indicar el número de tareas para cada servicio y al escalar hacia arriba o hacia abajo, Swarm añade o quita tareas para llegar al estado deseado.
- Reconciliación de estado: monitorización de todos los nodos del clúster y en caso de fallo, Swarm recupera automáticamente el número de tareas definidas en la configuración del servicio.
- Balanceo de carga y Routing Mesh: Swarm efectúa un balanceo de carga interno (Figura 45). Los puertos del servicio se exponen en todos los nodos y con Routing Mesh, desde cualquier puerto externo se puede acceder a cualquiera de los contenedores del servicio **[38]**.

Figura 45: Balanceo de carga interno y Routing Mesh



Ahora mismo, Swarm no está integrado en la interfaz web de Cloudy, por lo que nos proponemos implementar un clúster Swarm manualmente y estudiar cómo incluir sus funcionalidades en Cloudy a la vez que seguimos manteniendo la filosofía de facilidad de uso de este sistema operativo.

En esta ocasión, los nodos del clúster estarán todos en una misma ubicación perteneciente al mismo usuario. Posteriormente estudiaremos como extender el Swarm con nodos que estén repartidos entre múltiples ubicaciones pertenecientes al mismo o distintos usuarios.

3.4.1.1 Tipos de Nodos Swarm y alta disponibilidad

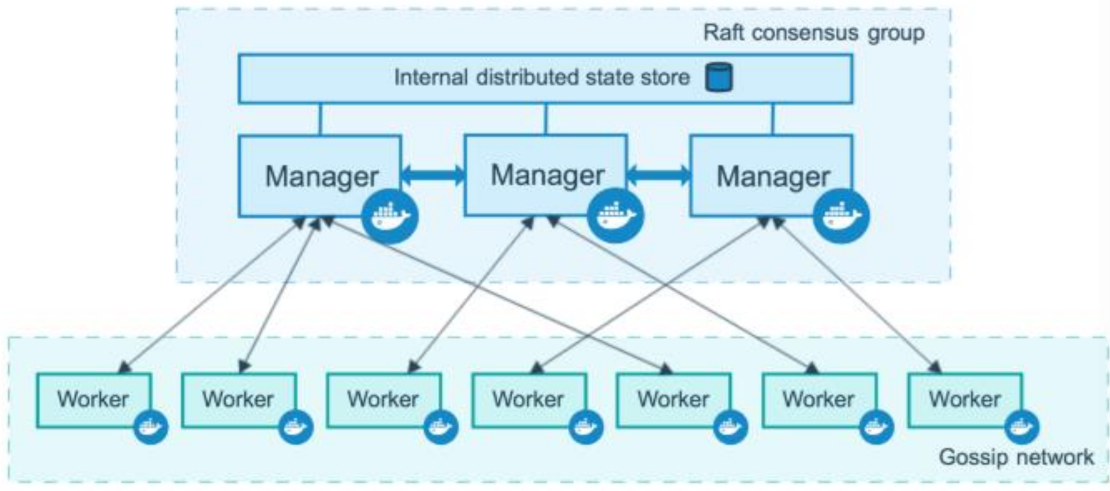
Podemos añadir a un clúster Swarm 2 tipos de nodos distintos:

- Modo 'manager': ejecuta servicios, coordina el clúster y permite recibir comandos específicos para gestionar el clúster.

- Modo 'worker': únicamente pone sus recursos a disposición del clúster para ejecutar servicios.

Revisando la documentación de Docker **[39]**, comprobamos que para obtener alta disponibilidad necesitamos al menos 3 nodos 'Manager' en un clúster (Figura 46), ya que Swarm utiliza el protocolo de consenso Raft que elige por una votación en mayoría el nodo líder entre los manager disponibles **[40]**.

Figura 46: Ejemplo de clúster Swarm con 3 managers



Se podría crear un clúster de 2 'managers' o de un 'manager' y un 'worker', pero en ese caso no conseguiríamos alta disponibilidad ya que, si fallase un nodo los contenedores que estén ejecutándose desde el nodo superviviente seguirían funcionando, pero no habría un mecanismo para arrancar automáticamente los servicios que se han perdido con el fallo.

Esa no es una solución válida en nuestro caso, ya que queremos optimizar el uso de recursos ejecutando un único contenedor por servicio. Por esa razón, para continuar con nuestro proyecto, no nos queda más remedio que añadir un nodo Cloudy adicional a nuestro clúster.

3.4.2 Creación del tercer nodo Cloudy

Añadiremos un nuevo nodo Cloudy que llamaremos ‘SmartHome3’ siguiendo los procedimientos que utilizamos previamente para crear los nodos ‘SmartHome1’ y ‘SmartHome2’.

3.4.2.1 Hardware

Apilaremos una nueva Raspberry para crear un clúster de 3 nodos (Figura 47) y como las demás, la conectaremos a la alimentación con un cable microUSB y el puerto de red a uno de los puertos LAN libres de nuestro Safe@Office.

Figura 47: Pila de 3 Raspberry pi

3.4.2.2 Sistema Operativo y redes

Seguiremos el mismo procedimiento que en los otros nodos: cambiaremos el nombre del equipo a ‘SmartHome3’ con el comando ‘raspi-config’ y asignaremos a nuestra Raspberry la dirección IP ‘192.168.10.12’ que configuraremos de forma estática en el fichero ‘/etc/dhcpcd.conf’.

```
# Example static IP configuration:
interface eth0
static ip_address=192.168.10.12/24
static routers=192.168.10.1
static domain_name_servers=192.168.10.1 8.8.8.8 fd51:42f8:caae:d92e::1
```

Actualizaremos Raspbian a Cloudy con el Script ‘Cloudytnizar.sh’ y comprobaremos que Cloudy está instalado correctamente accediendo a su interfaz web a través la IP de ‘192.168.10.12’ en el puerto 7000.

Configuraremos nuestro Safe@Office añadiendo la nueva configuración de red para ‘SmartHome3’ e incluyendo la IP publica ‘172.16.0.251’. Solo publicaremos los servicios del Swarm a través de la IP publica ‘172.16.0.250’. La configuración final de red de la segunda fase puede encontrar en el Anexo 2 de este documento.

3.4.2.3 Software

Instalaremos los servicios predefinidos necesarios en nuestro nuevo nodo Cloudy de la misma manera que en los nodos anteriores. Instalaremos Docker y Serf, pero esta vez no crearemos los contenedores Docker, ya que estos serán arrancados automáticamente al desplegar los servicios en el Swarm.

Modificamos el fichero '/etc/avahi-ps-serf.conf' para que Serf anuncie los servicios del nodo a través de su IP pública en el Safe@Office.

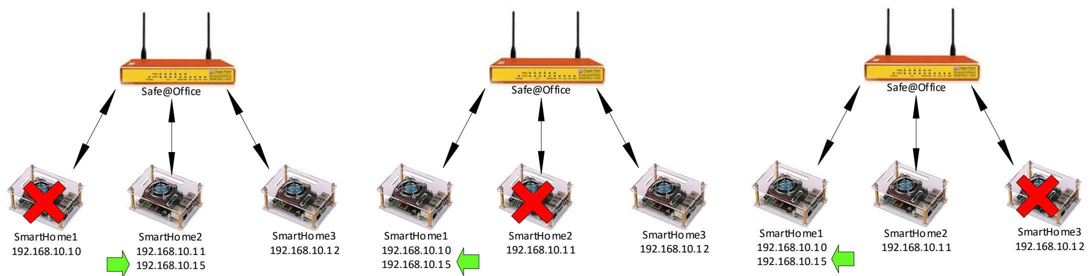
```
SERF_RPC_ADDR=127.0.0.1:7373
SERF_BIND=5000
SERF_JOIN=
ADVERTISE_IP=172.16.0.251
```

Modificamos también el fichero '/etc/getinconf-client.conf' para indicar a los nodos externos la dirección IP donde están los servicios publicados.

```
#!/bin/sh
GETINCONF_IGNORE=1
PUBLIC_IP=172.16.0.251
```

No instalaremos keepalived en 'SmartHome3' porque no es necesario, ya que con nuestro clúster Swarm de 3 nodos podemos perder cualquiera de los nodos sin necesidad de cambiar la IP flotante al tercer nodo (Figura 48).

Figura 48: IP failover en clúster de 3 nodos



Implementaremos los cambios en la publicación de servicios del apartado 3.3.3.3 para anunciar los servicios del Swarm en la IP flotante.

3.4.3 Implementación del clúster Swarm

Nuestro clúster Swarm estará pues finalmente formado por los nodos 'SmartHome1', 'SmartHome2' y 'SmartHome3' todos en modo 'Manager' con la siguiente configuración de red (Figura 49).

Figura 49: Direcciones IP de los nodos del Clúster Swarm

Descripción	Dirección IP	Mascara	Gateway	IP Publica
SmartHome1	192.168.10.10	255.255.255.0	192.168.10.1	172.16.0.253
SmartHome2	192.168.10.11	255.255.255.0	192.168.10.1	172.16.0.252
SmartHome3	192.168.10.12	255.255.255.0	192.168.10.1	172.16.0.251

Crearemos el clúster incluyendo al mismo tiempo el nodo como ‘manager’ ejecutando el siguiente comando en ‘SmartHome1’.

```
joe@SmartHome1:~$ sudo docker swarm init --advertise-addr 192.168.10.10
```

Una vez creado el primer nodo ‘manager’ podremos ir agregando más nodos Docker al clúster indicando la IP y el token anunciado por ‘SmartHome1’. Conectaremos ‘SmartHome2’ y ‘SmartHome3’ a nuestro Swarm como ‘workers’ ejecutando el comando ‘docker swarm join’ en cada uno de los nodos.

```
$ sudo docker swarm join --token SWMTKN-1-2efg7dcdk05lergpzz9ajtvj4cn5oehrqkv
9dh3c5b8xncun41-3vqpbeomlkbtnvdwrmjlpsdou 192.168.10.10:2377
```

Una vez finalizada la creación de nuestro clúster Swarm. Podemos listar los nodos que lo componen con el comando ‘docker node ls’ desde ‘SmartHome1’, aquí comprobaremos que la versión de todos los Dockers coincide y que todos los nodos se comunican correctamente.

```
joe@SmartHome1:~ $ sudo docker node ls
ID                            HOSTNAME        STATUS          AVAILABILITY       MANAGE
oijscqhgpbbk2t68yrseqk3ku *   SmartHome1      Ready           Active             Leader
skdx03l26yach8g3u03tj8ylq     SmartHome2      Ready           Active
g3lr17vxg7si3f35y645r2ujd     SmartHome3      Ready           Active
```

Una vez comprobada la comunicación y actualizada, en su caso, la versión de Docker de cada nodo, podremos convertir los nodos en ‘Managers’ ejecutando el comando ‘docker node promote <id>’ en ‘SmartHome1’.

```
root@SmartHome1:/home/pi$ sudo docker node promote hhk4t7vaxi77jwou79ybst1hp
hhk4t7vaxi77jwou79ybst1hp
root@SmartHome1:/home/pi$ sudo docker node promote jwt8x4zkaoeqn8ly42s08uvr0
jwt8x4zkaoeqn8ly42s08uvr0
```

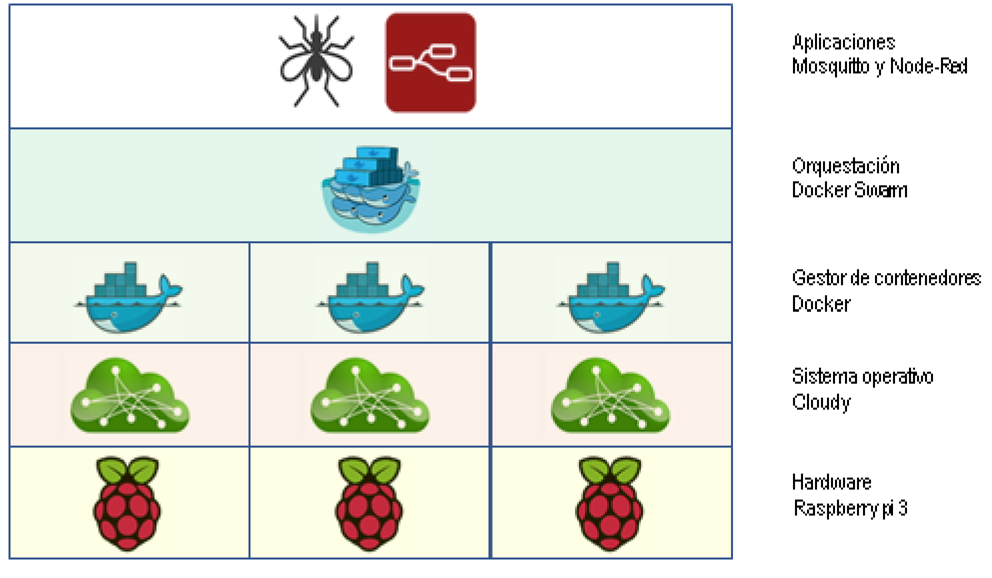
3.4.3.1 Ejecución de servicios

Docker Swarm ofrece los servicios de sus contenedores a través de todos los hosts del clúster y cuando existen varios contenedores para un mismo servicio, Swarm accede a cualquiera de ellos de manera aleatoria. Esto implica que tendremos las siguientes restricciones para ejecutar los servicios ‘mqtt’ y ‘Node-Red’ en nuestra microCloud:

- Solo podemos usar un contenedor por cada servicio de nuestra microCloud en todo el clúster (Figura 50). Si usásemos varios, podría ejecutarse una parte del trabajo en un contenedor y otra parte en otro debido al balanceo de carga interno.

- Necesitamos algún mecanismo para no acceder directamente a un servicio en una IP fija de un nodo en concreto. Por esa razón en nuestro caso seguiremos necesitando la funcionalidad de IP Failover que nos proporciona keepalived (Figura 48) y accederemos siempre a los servicios a través de la IP flotante.

Figura 50: Nodos en Clúster con Swarm



En caso de fallo del nodo donde se esté ejecutando uno de los contenedores, el Swarm de 3 'masters' rearrancará automáticamente el contenedor en cualquiera de los nodos supervivientes. De esta manera tenemos redundancia usando la mitad de los recursos que la solución con keepalived y nodos independientes que vimos en la primera parte de este capítulo (Figura 38).

Antes de ejecutar los servicios de nuestra microCloud en el Swarm, no debemos olvidar de parar los contenedores que creamos en la primera fase ya que si no los eliminamos, los puertos publicados interferirán con los servicios que hemos definido en el Swarm.

Desde la opción 'Enterprise Cloud - Docker Compose' de Cloudy crearemos un nuevo proyecto Docker compose que llamaremos 'SmartHome_service_public' con el siguiente fichero de definición de servicios:

```
version: "3"
services:
  mqtt:
    image: robotany/mosquitto-rpi:latest
    ports:
      - "1883:1883"
    restart: always

  node-red:
    image: easypi/node-red-arm:latest
    ports:
      - "1880:1880"
    restart: always
```

Aprovecharemos la recreación de los contenedores como servicios para cambiar la imagen de Node-Red que usamos con Docker. En este caso elegiremos la imagen 'easypi/node-red-arm' ya que esta imagen ocupa la cuarta parte de memoria que la imagen anterior.

Una vez creado y lanzado el proyecto desde la interfaz web de Cloudy, podemos comprobar cómo se ha creado un fichero llamado '/etc/cloudy/docker-compose/projects/SmartHome_service_public/docker-compose.yml'.

Como Swarm permite crear servicios a partir de este tipo de ficheros **[41],** usaremos la funcionalidad Docker Compose de Cloudy para crear y testear los contenedores y una vez verificado su funcionamiento desplegarlos en el Swarm con el comando 'docker stack deploy' usando el mismo fichero YAML.

```
$ cd /etc/cloudy/docker-compose/projects/SmartHome_service_public
$ sudo docker-compose down
$ sudo docker stack deploy -c docker-compose.yml --resolve-image never SmartH
ome_service_public
```

He tenido que usar la opción '--resolve-image never' para evitar que algunos contenedores del stack se queden sin arrancar. Por ejemplo, el contenedor Mosquitto tiene arquitectura ARM (arm) pero Swarm no lo reconoce como la misma arquitectura ARM de la Raspberry pi 3 (armv71).

En nuestro caso comprobamos que se crea un único contenedor por cada servicio y que el servicio se puede acceder desde cualquiera de los nodos del clúster. El nuevo contenedor de Node-Red arranca en blanco, lo que nos obliga a volver configurarlo y a cargar todos los flujos.

```
$ sudo docker service ls
ID                  NAME                                MODE                REPLICAS
IMAGE                            PORTS
29me3fphlxbr        SmartHome_service_public_mqtt       replicated          1/1
robotany/mosquitto-rpi:latest    *:1883->1883/tcp
h54zweuwh9yt        SmartHome_service_public_node-red   replicated          1/1
easypi/node-red-arm:latest       *:1880->1880/tcp
```

Al cargar la configuración de Node-Red, comprobamos que ahora podemos acceder a los contenedores usando el nombre del servicio definido en el fichero YAML como un registro de DNS. En nuestro caso reconfiguramos el bróker MQTT con el nombre 'mqtt' para acceder al servicio desde el contenedor Node-Red en cualquier host.

3.4.3.2 Publicación de servicios

Para poder publicar estos nuevos servicios para los usuarios externos a través de Serf, modificaremos el script de Python que creamos en el apartado 3.3.3.3 de este capítulo ya que los servicios creados con 'docker deploy' ya no aparecen listados como contenedores en la API de Docker.

En el script enumeraremos los servicios que se ejecutan en el clúster Swarm y publicaremos automáticamente en la IP alternativa los servicios cuyo nombre contenga la cadena 'public', todos los demás servicios, serán considerados como servicios internos y no se publicarán mediante Serf.

Las líneas que hay que añadir al script '/usr/local/python/regpublic.py' para publicar los servicios Swarm son las siguientes:

```
 # Publica servicios Docker swarm
for service in client.services.list():
        if ('service' in service.name):
                strport=str(service.attrs['Endpoint']['Ports'][0]['PublishedP
ort'])
                if ('public' in service.name) and strport:
                        cmd='publish_alt'
                        tipo='Service'
                        img = ""
                        subprocess.call(['/usr/sbin/avahi-ps', cmd, service.n
ame, tipo, strport, ""])
```

3.4.3.3 Problema de configuración de los servicios Swarm

Una vez arrancados nuestros servicios en el Swarm, observamos un comportamiento no deseado durante las pruebas de reinicio de los nodos:

Cada vez que reiniciamos un nodo con Node-Red y el contenedor se desplaza a otro nodo, pierde toda su configuración, incluido nodos añadidos y flujos programados. Este comportamiento es especialmente problemático porque al perder la configuración de Node-Red perdemos toda la programación de la microCloud. Básicamente todo deja de funcionar.

Esto ocurre porque cuando Swarm ajusta la configuración de los servicios al estado del clúster y desplaza un contenedor a un nuevo nodo, el contenedor arranca como si fuera la primera vez que se ejecuta (apartado 2.3.6), creando un nuevo volumen anónimo con un nombre aleatorio.

Incluso si nuestro contenedor de Node-Red arrancase en un nodo con un volumen anónimo que fue creado anteriormente por otro Node-Red, no tiene ningún medio de reconocer y usar ese volumen anónimo.

Una solución rápida para evitar este problema es el de especificar volúmenes con nombre **[42]** en la definición de servicios, de esta manera el contenedor usará el nuevo volumen con nombre en vez de un volumen anónimo.

Por ejemplo, podemos crear el volumen 'Node-red_config' en cada uno de los nodos de nuestro clúster e indicar a Swarm que los contenedores Node-Red que se inicien siempre usen ese volumen.

```
$ sudo docker volume create node-red_config
node-red_config
$ sudo docker volume list
DRIVER              VOLUME NAME
local               node-red_config
```

Al arrancar el contenedor en un nuevo nodo, este buscará un volumen local con nombre 'node-red_config', en caso de que ya exista el volumen, Node-Red lo usará para su configuración, sino, se creará un nuevo volumen en blanco con ese nombre. Al configurar el contenedor Node-Red la configuración se guardará en el volumen con nombre y podrá reusarse por cualquier otro contenedor de Node-Red.

Pero hay un inconveniente: tenemos que mover el contenedor a todos los nodos del clúster y desde allí cargar en cada uno la configuración y los flujos de Node-Red en el volumen ‘node-red_config’. Este procedimiento debería hacerse cada vez que cambiásemos la configuración de Node-Red.

Esta solución se vuelve impracticable cuando el número de nodos o la frecuencia de las modificaciones se incrementa. Por esa razón tenemos que buscar una solución de replicación automática de datos entre los nodos.

3.4.3.4 Replicación de datos entre Nodos

La solución de replicación entre nodos que necesitamos deberá cumplir los siguientes requerimientos:

- Tiene que ser una solución descentralizada. No queremos introducir ningún hardware adicional e incrementar puntos de fallo.

- Tiene que replicar carpetas de uno a muchos. Un número indeterminado de nodos tiene que recibir las réplicas desde un único nodo.

- Tiene que ser replicación a nivel de fichero. No interesa soluciones complejas con sistemas de ficheros en clúster o replicación por bloques

En un principio, parece que podemos resolver el problema ejecutando scripts con ‘rsync’ en los nodos de nuestro clúster, pero pronto nos damos cuenta de que debido a la descentralización de Swarm, lo ideal sería que los nodos se replicasen en una topología de malla completamente conectada (Figura 51).

Figura 51: Malla completamente conectada

Con esta topología el número de conexiones aumenta exponencialmente con el número de nodos (n): conexiones = n*(n-1)/2 y además, las actualizaciones se pueden efectuar desde cualquier nodo y se tienen que propagar a los demás de una manera eficiente.

De esta manera llegamos a la conclusión que necesitamos buscar una solución de replicación descentralizada que se pueda escalar masivamente para manejar este nivel de complejidad.

Tras repasar las opciones existentes, decidimos usar el software Syncthing. Syncthing es una aplicación descentralizada de replicación de ficheros, está integrada en Cloudy, pero la ejecutaremos como un servicio Swarm global en nuestro clúster.

Para evitar la mala práctica de acceder directamente a volúmenes de Docker desde el Host, crearemos volúmenes montados en la configuración de los contenedores que queremos replicar en una nueva carpeta que llamaremos '/usr/local/docker-config'.

Configuraremos Syncthing de manera que se dedique a replicar todos los datos contenidos en 'docker-config', así copiaremos automáticamente entre los nodos cualquier volumen Docker montado que apunte a esa carpeta. Una vez hecha esta configuración inicial no tendremos que configurar nada más.

Desgraciadamente, algunos contenedores se ejecutan como 'root', por lo que no tendremos más remedio que dar acceso privilegiado al contenedor Syncthing para acceder a todos los datos de las carpetas.

Usaremos una imagen de Syncthing para arquitectura ARM llamada 'lsioarmhf/syncthing'. La desplegaremos en el Swarm como un servicio global. Un servicio global ejecuta un contenedor en cada uno de los nodos del clúster.

Crearemos un fichero de definición de servicios YAML para ejecutar el servicio Syncthing en el path '/etc/cloudy/docker-compose/projects/synthing_service'.

```
version: "3.2"
services:
  syncthing:
    image: lsioarmhf/syncthing:latest
    environment:
      - PGID=0
      - PUID=0
      - UMASK_SET=022
    volumes:
      - syncthing_config:/config
      - /usr/local/docker-config:/sync
    networks:
      - syncnet
    ports:
      - target: 8384
        published: 8384
        mode: host
    deploy:
      mode: global
volumes:
  syncthing_config:
networks:
  syncnet:
```

La configuración de Syncthing se guardará en un volumen con nombre, llamado 'syncthing_config' que se creará automáticamente en cada nodo.

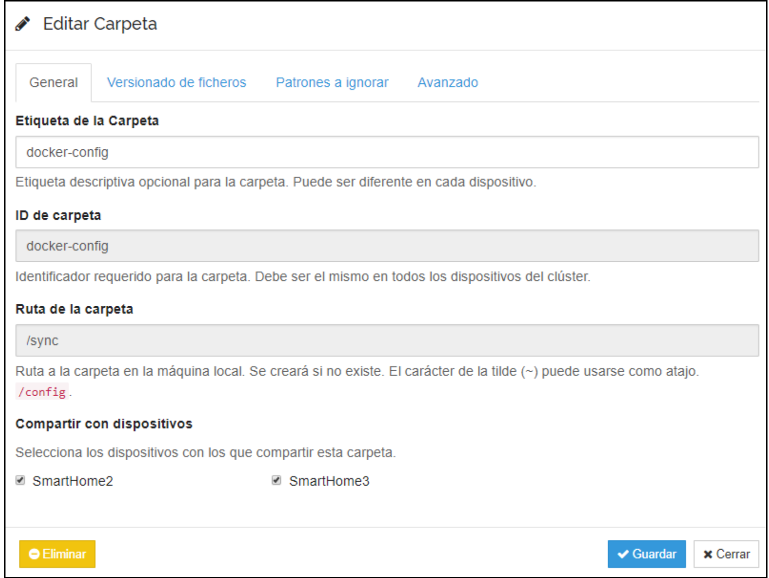
Usaremos la opción ‘mode: host’ en la configuración del puerto para ligar el contenedor al host donde se esté ejecutando, ya que syncthing tiene que identificar unívocamente el nodo donde replica los datos.

Lanzamos nuestro servicio con el comando ‘docker stack deploy’:

```
$ cd /etc/cloudy/docker-compose/projects/syncthing_service
$ sudo docker stack deploy -c docker-compose.yml --resolve-image never
syncthing_service
```

Solo queda crear la carpeta ‘/usr/local/docker-config’ y configurar el nombre de Host en cada syncthing de cada nodo. Una vez hecho esto, configuraremos la carpeta en ‘SmartHome1’ (Figura 52) y la replicaremos entre todos los nodos.

Figura 52: Configuración de carpeta docker-config



3.4.3.5 Configuración de los servicios de la microCloud

Una vez puesto en marcha el mecanismo de replicación de las configuraciones de los contenedores entre los nodos, modificaremos la definición de los servicios de la microCloud en el fichero YAML para usar un volumen montado en la configuración del servicio Node-red.

```
version: “3”
services:
  mqtt:
    image: robotany/mosquitto-rpi:latest
    ports:
      - "1883:1883"
  node-red:
    image: easypi/node-red-arm:latest
    volumes:
      - /usr/local/docker-config/node-red:/root/.node-red
    ports:
      - "1880:1880"
```

Crearemos una carpeta llamada ‘node-red’ dentro de la carpeta replicada ‘/usr/local/docker-config’ y volveremos a desplegar los servicios de nuestra microCloud con el fichero YAML y el comando ‘stack deploy’.

```
$ cd /etc/cloudy/docker-compose/projects/SmartHome_service_public
# docker stack deploy -c docker-compose.yml --resolve-image never SmartHome_
service_public
```

Reconfiguramos el servicio ‘Node-Red’ por última vez teniendo cuidado de configurar el nombre por defecto de los flujos como ‘flows.json’ en el fichero ‘settings.js’ del contenedor, ya que el comportamiento por defecto es el de usar el hostname, que es dinámico en los contenedores Docker.

Y finalmente, reiniciaremos cada nodo uno después de otro para comprobar como el contenedor Node-Red puede desplazase a cualquiera de los nodos del Swarm conservando su configuración.

3.4.3.6 Integración de Swarm en la interfaz de Cloudy

Para integrar la funcionalidad Swarm en Cloudy necesitaremos crear un módulo en la interfaz web para gestionar los nodos del clúster. Las opciones del nuevo módulo y los comando asociados serían los siguientes:

- Opción ‘Swarm - Create’: pide el campo ‘address’ (dirección IP), ejecuta el comando ‘docker swarm init –advertise-addr <address>’.
- Opción ‘Swarm - Join’: pide el campo ‘token’, ejecuta el comando ‘docker swarm join –token <token>’. Según el token, el nodo será ‘Worker’ o ‘Manager’.
- Opción ‘Swarm - Leave’: ejecuta el comando ‘docker swarm leave’ para desconectar el nodo actual del Swarm.
- Opción ‘Swarm - List tokens’: ejecuta los comandos ‘docker swarm join-token manager’ y ‘docker swarm join-token worker’ y muestra los tokens en pantalla. Estos tokens se compartirán y son los que hay que indicar en la opción ‘Swarm - Join’ para añadir un nodo a un clúster existente.
- Opción ‘Swarm - Nodes’: esta opción solo aparecerá si el nodo es un ‘Manager’, ejecuta el comando ‘docker node ls’ que muestra la lista de nodos del clúster.

  Por cada nodo <id> de la lista aparecerá un botón con las acciones:

  - Botón ‘Delete’: ejecuta el comando ‘docker node rm <id>’
  - Botón ‘Promote’: ejecuta el comando ‘docker node promote <id>’
  - Botón ‘Demote’: ejecuta el comando ‘docker node demote <id>’

- Opción ‘Swarm Execute stack’: pide el campo ‘proyecto’ y ejecuta un proyecto de Docker Compose con el comando ‘docker stack deploy -c docker-compose.yml –resolve-image never <Nombre Proyecto>’.

- Opción ‘Swarm - List stacks’: ejecuta el comando ‘docker stack ls’ y por cada stack <id> de la lista aparecerá un botón con las acciones:

    - Botón ‘Delete’: ejecuta el comando ‘docker stack rm <id>’

3.5 Extensión de la microCloud

En este apartado estudiaremos como extender nuestra microCloud mediante extensión del Swarm a nodos externos con una VPN o mediante programación distribuida con la ayuda de ‘DNR-Editor’.

3.5.1 Extensión del Swarm

Hasta ahora nos hemos limitado a ejecutar nuestra microCloud en una misma ubicación con un único propietario. Analizaremos las implicaciones de usar un Swarm con varios propietarios, en múltiples ubicaciones y con distintas arquitecturas.

3.5.1.1 Docker Swarm con varios propietarios

En este apartado analizaremos qué ocurre cuando los nodos pertenecen a distintos propietarios. Entendemos que todos los nodos ya están conectados en un clúster, y que solo tenemos acceso a nuestro nodo local con funcionalidad ‘Manager’. Estudiaremos la versión Community Edition de Docker.

En este proyecto, hemos usado Docker con usuarios privilegiados ejecutándolo con ‘sudo’, pero incluso si usásemos un usuario no privilegiado perteneciente al grupo ‘docker’, el poder ejecutar el comando ‘docker run’ nos da fácilmente acceso administrador a nuestro equipo **[43]**. Por ejemplo:

```
$ docker run --rm -v /:/host busybox sh -c 'echo myname ALL=(ALL:ALL)
NOPASSWD | tee -a /host/etc/sudoers'
$ sudo ...
```

Aún en este caso, la facilidad de obtener privilegios de administrador con Docker en nuestro equipo local nos es problemática porque, para gestionar un Swarm, no necesitamos tener acceso privilegiado a los demás nodos.

El problema es que desde un nodo ‘manager’ se tiene acceso completo a todos los recursos de todos los equipos del Swarm. (almacenamiento, CPU, memoria, puertos...) por lo que es trivial hacer ataques de denegación de servicio a cualquier nodo del clúster.

Además, desde nuestro nodo ‘manager’ se puede conseguir acceso privilegiado a cualquiera de los contenedores del Swarm: podríamos instalar los

paquetes que quisiéramos en un contenedor para subvertir su funcionalidad o sencillamente lanzar un contenedor con malware en nodos ajenos.

En los servicios Cloud de contenedores comerciales, se combina el uso de Swarm con máquinas virtuales. Los contenedores se escalan y se orquestan con Docker Swarm, pero el control de recursos y la manera de que varios usuarios compartan un equipo físico se solventa usando máquinas virtuales.

Como nuestros nodos están basados en Raspberry, no podemos usar virtualización completa por lo que en nuestro caso la granularidad de asignación de recursos es por nodo completo.

La forma de compartir recursos sin virtualización en una red comunitaria de forma segura seria asignar una Raspberry que no esté conectada a nuestra Red Local y que se limitase a dar servicio a la red externa.

Aun así, seguiremos teniendo problemas de propiedad: hemos visto que necesitamos al menos 3 nodos 'manager' en Docker Swarm para implementar alta disponibilidad. Cualquiera de los propietarios de un nodo 'manager' tiene acceso a los contenedores creados en el Swarm por los demás usuarios.

Los nodos 'worker' del Swarm, por el contrario, no tienen opción de gestionar el clúster y se limitan a proporcionar recursos sin ninguna posibilidad de controlar a los propietarios de los nodos 'manager'.

Esta problemática se está empezando a afrontar en la versión comercial de Docker, la Enterprise Edition desde la 17.06 implementando RBAC (Role Based Access Controls) a nivel de API.

3.5.1.2 Docker Swarm en múltiples ubicaciones

En mi opinión tiene sentido tener varias ubicaciones distintas, pero únicamente con nodos que pertenezcan al mismo propietario. Un caso de uso sería por ejemplo un particular que tiene varias casas, o una empresa con varias delegaciones.

Por esa razón, en vez de dar acceso público y abierto a un clúster Swarm, lo idóneo es compartirlo mediante una red VPN.

3.5.1.3 Conectar Swarm mediante una VPN

Para conectar los nodos entre sí a través de una VPN debemos primero averiguar qué puertos y protocolos son necesarios para la comunicación del Swarm según la documentación de Docker **[44]** (Figura 53).

Figura 53: Puertos y Protocolos de Swarm

Protocol	Port	Description
TCP	2377	cluster management communications
TCP UDP	7946	Communication among nodes
UDP	4789	Overlay network traffic

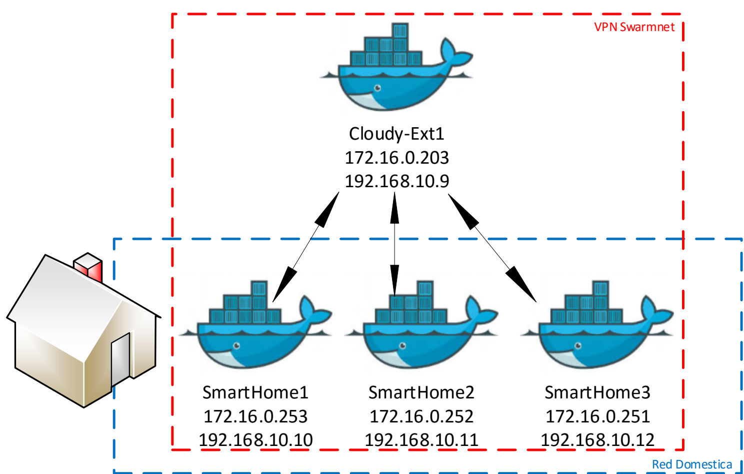
Como Swarm usa el protocolo UDP para sus comunicaciones, tenemos que elegir una VPN que permita conectar los nodos a nivel de capa 2 ya que los paquetes UDP no se transmiten entre diferentes subredes.

Usaremos como VPN el software TincVPN que permite crear redes virtuales en la capa 2 de TCP/IP. TincVPN ha sido usado previamente en Cloudy para publicar servicios entre nodos mediante Avahi, es una implementación de mesh VPN flexible y distribuida que se ajusta a las necesidades de Swarm.

Añadiremos a nuestra microCloud el nodo externo 'Cloudy-Ext1' que configuramos en la Fase 1. Una vez creada la VPN de nivel 2, podremos conectar con Bridging el nodo 'Cloudy-Ext1' como 'worker' a los demás nodos del clúster Swarm.

De esta manera tendremos alta disponibilidad incluso si se pierde la conexión con el nodo externo ya que en ese caso nuestro Swarm solo perderá un 'worker' y conservará la alta disponibilidad con los tres 'managers'. La VPN que crearemos con tinc se llamará 'swarmnet' (Figura 54).

Figura 54: VPN Swarmnet



Primero instalaremos el paquete 'tinc' en todos nuestros nodos y crearemos en cada nodo una carpeta llamada '/etc/tinc/swarmnet' donde guardaremos los ficheros de configuración de nuestra VPN. Seguiremos el ejemplo de red con Bridging que está disponible en la web de TincVPN **[45]**.

En cada fichero Host dentro de '/etc/tinc/swarmnet/hosts' tendremos la dirección publica ('172.16.0.x') de cada nodo y su clave pública. En '/etc/tinc/swarmnet' tendremos el fichero 'rsa_key.priv' que contendrá la clave privada del nodo y el fichero 'tinc.conf'.

```
Name = <Nombre_nodo>
AddressFamily = ipv4
Mode = switch
Interface = tun0
```

Donde <Nombre_nodo> es el nombre del nodo. Los nodos internos tendrán todos está línea para conectarse al nodo externo.

```
ConnectTo = Cloudy-Ext1
```

El fichero ‘/etc/tinc/swarmnet/tinc-up’ será el siguiente:

```
#!/bin/sh
ifconfig $INTERFACE 0.0.0.0
brctl addif br0 $INTERFACE
ifconfig $INTERFACE up
```

El fichero ‘/etc/tinc/swarmnet/tinc-down’ será el siguiente:

```
#!/bin/sh
ifconfig $INTERFACE down
```

Para hacer las pruebas de conexión, crearemos los bridges y manipularemos la configuración de red con un script. De esta manera podremos volver a la configuración original fácilmente solo reiniciando el nodo. Este es el script ‘startbr’ que usaremos para probar temporalmente la VPN.

```
brctl addbr br0
ifconfig br0 <IP_Interna> netmask 255.255.255.0
ifconfig eth0 0.0.0.0
brctl addif br0 eth0
ifconfig eth0 up
route add default gw 192.168.10.1
service tinc restart
```

Donde <IP_Interna> es la dirección IP privada (192.168.10.x). Únicamente en el nodo ‘Cloudy-Ext1’ añadiremos la siguiente línea ya que, al no estar detrás de un firewall, la IP publica se debe incluir en el bridge.

```
ip address add 172.16.0.203/24 dev br0
```

Una vez puesta en marcha nuestra VPN, comprobamos que existe inestabilidad (bloqueos y perdida de paquetes entre un 20% y 40%). Una vez investigado y resuelto este problema (ver Anexo 4 de este documento), seguimos con la implementación de la VPN.

Agregaremos ‘Cloudy-Ext1’ a nuestro Swam como ‘worker’.

```
joe@Cloudy-Ext1$ sudo docker swarm join --token SWMTKN-1-2efg7dcdk05lergpzz9a
jtvj4cn5oehrqkv9dh3c5b8xncun41-3vqpbeomlkbtnvdwrmjlpsdou 192.168.10.10:2377
```

Y comprobamos desde un ‘manager’ que se ha añadido correctamente al clúster Swarm.

```
joe@SmartHome1 $ sudo docker node ls
ID                            HOSTNAME       STATUS         AVAILABILITY      MANAGE
1esr58sm6f6xs04l0br368y8l     Cloudy-Ext1    Ready          Active            Reacha
oijscqhgpbbk2t68yrseqk3ku *   SmartHome1     Ready          Active            Reacha
skdx03l26yach8g3u03tj8ylq     SmartHome2     Ready          Active            Reacha
g3lr17vxg7si3f35y645r2ujd     SmartHome3     Ready          Active            Leader
```

Al incluir el nuevo nodo se intentan ejecutar automáticamente las imágenes del Swarm en formato ARM en 'Cloudy-Ext1', pero fallan al ser este un nodo con arquitectura x86. Para evitar este problema implementaremos etiquetas en todos los nodos para diferenciar su arquitectura con restricciones.

```
$ sudo docker node update --label-add arc=arm SmartHome1
$ sudo docker node update --label-add arc=arm SmartHome2
$ sudo docker node update --label-add arc=arm SmartHome3
$ sudo docker node update --label-add arc=x86 Cloudy-Ext1
```

Y modificaremos nuestros servicios para ejecutar imágenes diferentes en las distintas arquitecturas. Por ejemplo, el servicio de replicación syncthing:

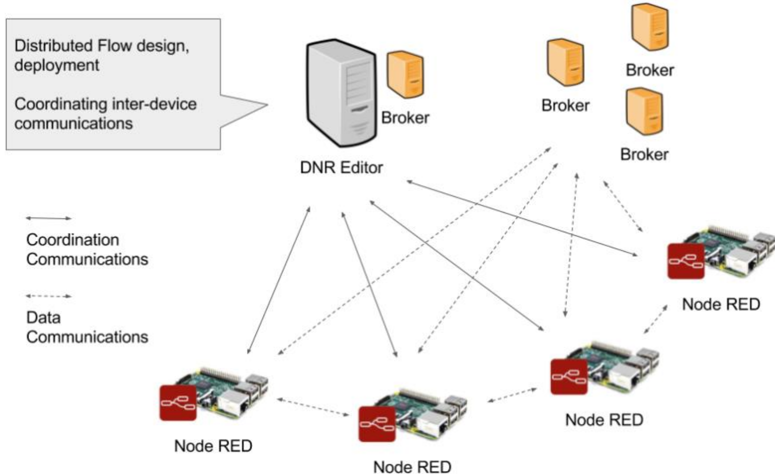
```
version: "3.2"

services:
  syncthing-arm:
    image: lsioarmhf/syncthing:latest
    environment:
      - PGID=0
      - PUID=0
      - UMASK_SET=022
    volumes:
      - syncthing_config:/config
      - /usr/local/docker-config:/sync
    networks:
      - syncnet
    ports:
      - target: 8384
        published: 8384
        mode: host
    deploy:
      mode: global
      placement:
        constraints:
         - node.labels.arc == arm

  syncthing-x86:
    image: syncthing/syncthing:latest
    environment:
      - PGID=0
      - PUID=0
      - UMASK_SET=022
    volumes:
      - syncthing_config:/config
      - /usr/local/docker-config:/sync
    networks:
      - syncnet
    ports:
      - target: 8384
        published: 8384
        mode: host
    deploy:
      mode: global
      placement:
        constraints:
         - node.labels.arc == x86
```

3.5.2 Programación distribuida con DNR-Editor

DNR-Editor o Distributed Node-Red es una extensión de Node-Red que permite ejecutar flujos de Node-Red de manera distribuida **[46]**. DNR-Editor se ejecuta en un servidor maestro, que distribuye los flujos a instalaciones existentes de Node-Red registradas previamente (Figura 55).

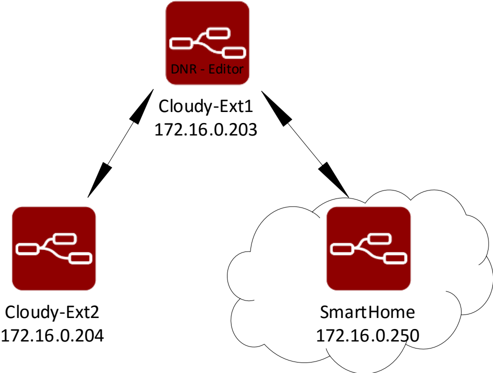
Para registrar una instalación de Node-Red a DNR-Editor, solo necesitamos instalar un nodo llamado 'Node-red-contrib-dnr' y configurar los parámetros necesarios, por lo que podemos usarlo directamente con nuestros nodos Cloudy sin necesidad de modificar nuestros contenedores de Node-Red.

Figura 55: Instalación de DNR-Editor



Investigaremos la viabilidad de DRN-Editor para implementar soluciones de Fog computing con instalaciones Linux de distintas arquitecturas: Instalaremos DNR Editor en el nodo x86 virtual 'Cloudy-Ext1' y desde allí accederemos al otro nodo x86 'Cloudy-Ext2' y a nuestra microCloud ARM (Figura 56).

Figura 56: Arquitectura DNR-Editor



Crearemos un Dockerfile para crear nuestra imagen a partir del contenedor oficial de node.js, exponiendo el puerto 1818.

```
FROM node:stretch
EXPOSE 1818
```

Y creamos la imagen intermedia para modificarla manualmente

```
$ sudo docker build -t node-test .
$ sudo docker run -it -u 0 node-test /bin/sh
# npm install -g dnr-editor
```

Tras instalar la aplicación y modificar la configuración, solo nos queda guardar los cambios que hemos efectuado en el contenedor en la imagen definitiva que llamaremos ‘dnr-editor’.

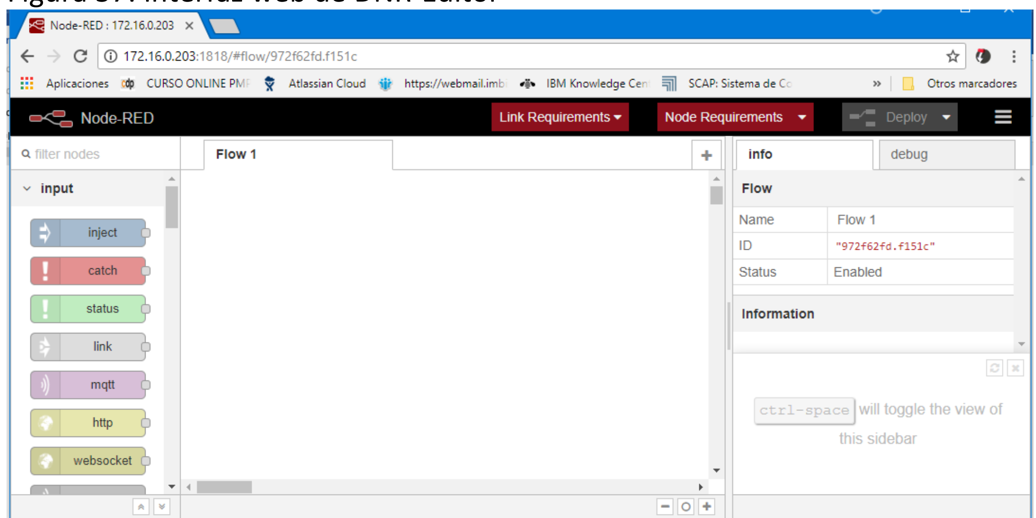
```
$ sudo docker ps | grep node-test | awk '{print $1}'
$ 824f4a9a52af
$ sudo docker commit 824f4a9a52af dnr-editor
```

Y con la nueva imagen crearemos un nuevo proyecto ‘Docker-compose’ para ejecutar el servicio en el nodo local.

```
version: "3"

services:
  dnr-editor:
    image: dnr-editor
    volumes:
      - "dnr-editor_config:/root/.node-red"
    ports:
      - 1818:1818
    command: "dnr-editor"

volumes:
  dnr-editor_config:
```

Tras ejecutar ‘docker-compose up’ comprobaremos que los servicios funcionan correctamente conectándonos desde un navegador a la interfaz web de DNR-Editor en ‘http://172.16.0.203:1818’ (Figura 57).

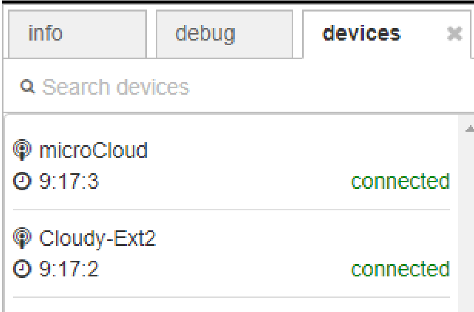
Figura 57: Interfaz web de DNR-Editor



Solo queda instalar el nodo ‘node-red-contrib-dnr’ en cada instalación de Node-red que queremos controlar desde DNR-Editor (Cloudy-Ext2 y nuestra

microCloud), copiar la semilla y verificar que las instalaciones aparecen registradas en la pestaña 'devices' (Figura 58).

Figura 58: Devices en DNR-Editor



Una vez instalados, añadiremos una configuración MQTT en cada Node-Red y crearemos un flujo de prueba en DNR-Editor (Figura 59). Con restricciones 'External' que hace referencia a 'Cloudy-Ext2' y microCloud que hace referencia al Node-Red de nuestro Clúster Swarm.

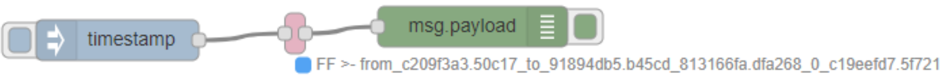
Figura 59: Flujo distribuido en DNR-Editor

Comprobaremos como se crea un flujo distribuido en 'Cloudy-Ext2' (Figura 60).

Figura 60: Flujo en Cloudy-Ext2

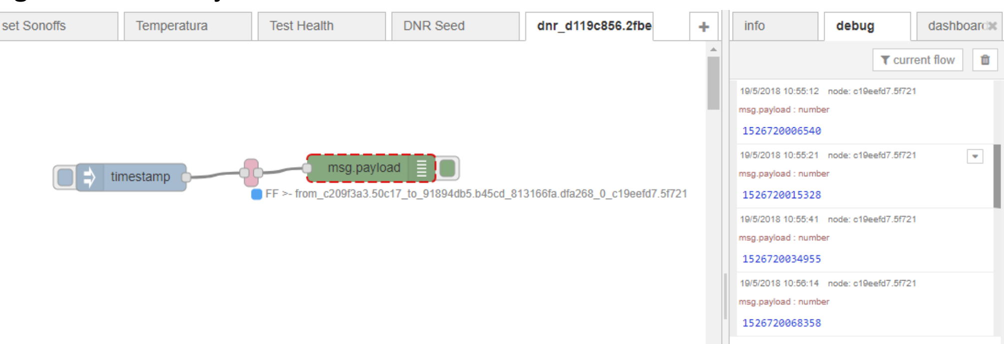
Y un flujo distribuido idéntico en nuestra microCloud (Figura 61).

Figura 61: Flujo en microCloud



Al inyectar el Timestamp manualmente en 'Cloudy-Ext2', podemos comprobar como recibimos el mensaje en nuestra microCloud (Figura 62).

Figura 62: Mensajes recibidos en microCloud



3.6 Conclusiones de la fase 2

Una vez finalizada la segunda fase de nuestro proyecto, podemos inferir las siguientes conclusiones.

3.6.1 Implementación de alta disponibilidad en redes domesticas

Hemos ampliado la microCloud de la primera fase para obtener un sistema en alta disponibilidad en redes domesticas por un coste razonable.

Podemos obtener un sistema redundante con Docker y keepalived con un clúster de dos nodos a costa de duplicar el consumo de recursos de nuestro sistema. También podemos usar syncthing en este clúster para replicar datos.

A partir de 3 nodos, sin embargo, se pueden optimizar el uso de recursos del clúster con Docker Swarm, pero esto implica una complejidad y una inversión significativa que puede desanimar algunos usuarios que no necesiten tantos recursos.

Docker Swarm no ofrece redundancia por sí solo, ya que necesita programas de balanceo o de IP failover como keepalived para ofrecer alta disponibilidad en los servicios del clúster.

3.6.2 Replicación de configuración para servicios Swarm

Durante la implementación de nuestra microCloud, hemos comprobado que Docker Swarm está enfocado a la ejecución de servicios sin estado, ya que no tiene mecanismos propios para replicar la configuración o datos generados por los contenedores.

Para guardar los estados de los contenedores Docker necesita usar servicios complementarios tales como almacenamiento compartido o replicación de datos entre los nodos del clúster.

En nuestra microCloud, los datos son principalmente ficheros de texto que se actualizan de vez en cuando y no tenemos ningún problema de concurrencia porque solo existe una única instancia de cada contenedor. En este caso Syncthing se ha revelado como la solución idónea para replicar esos datos.

3.6.3 Integración de Docker Swarm en la interfaz de Cloudy

Docker Swarm se puede integrar fácilmente en la interfaz de Cloudy, añadiendo un módulo que siga la filosofía de facilidad de uso para el usuario que tiene el sistema operativo.

Sin embargo, hay que buscar una nueva forma de publicar los servicios de Swarm, ya que todos los servicios publicados se publican en todos los nodos y además Swarm necesita mecanismos de balanceo o de IP failover que implican el uso de IPs flotantes. De hecho la IP flotante debería ser la única manera de acceder al Swarm.

Por esa razón será necesario añadir mecanismos alternativos de publicación de servicios con Serf parecidos a los que se han implementado informalmente para efectuar este proyecto.

3.6.4 Extensión de la microCloud

Hemos demostrado que la microCloud se puede extender fácilmente, siempre y cuando se mantenga los nodos 'manager' del Swarm bajo control directo de un único propietario, esto implica que los nodos externos sean añadidos como 'worker' bajo control total del propietario de los nodos 'manager'.

Para poder usar Docker Swarm en un mismo equipo con múltiples propietarios actualmente necesitamos usar entornos virtualizados para compartir los recursos o esperar que se desarrollen las funcionalidades necesarias que empiezan a añadir en la versión comercial de Docker.

DNR-Editor es un proyecto atractivo para añadir funcionalidad de programación distribuida a Node-Red en entornos con una gran cantidad de instalaciones, pero es un proyecto con un número muy pequeño de contribuidores y con muy poca documentación. Por lo que se debe considerar una solución experimental.

4 Tercera fase: Aplicaciones Avanzadas

4.1 Monitorización y Análisis

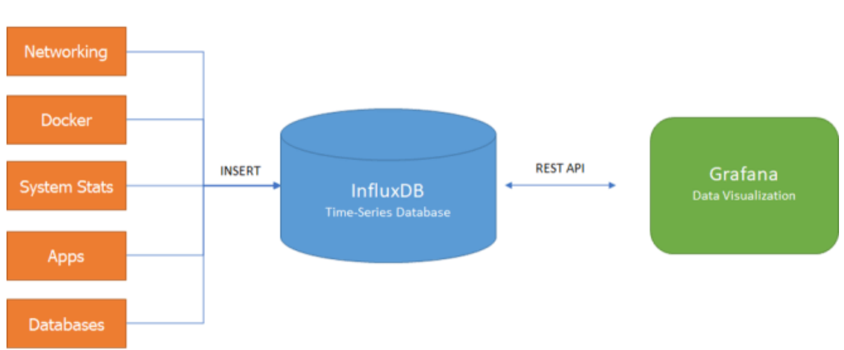
Una de las aplicaciones avanzadas que vamos a explorar es el poder efectuar monitorización y análisis de datos en la capa de Edge del IoT. Recordemos que la capa de Edge se encuentra en la frontera entre nuestra red local y la red externa. Nuestra microCloud abierta nos permite desarrollar esta funcionalidad.

El poder ejecutar este tipo de analíticas localmente nos permitirá prescindir de las funciones más avanzadas que nos puede proporcionar un proveedor Cloud para nuestros componentes de red. En este apartado vamos a comprobar si podemos obtener una solución funcional con la potencia de nuestro clúster.

4.1.1 Componentes

Los componentes de monitorización serán telegraf, influxdb y grafana (TIG, por las iniciales de cada aplicación) **[48]** tal como podemos ver en la Figura 63. Ejecutaremos los distintos componentes en contenedores Docker como un stack de servicios de Swarm que interactúan entre sí.

Figura 63: Arquitectura de monitorización



Telegraf es un agente que debe ejecutarse en cada nodo del clúster, se encargará de recolectar toda la información a nivel de nodo, incluyendo las métricas relativas a los contenedores de Docker.

InfluxDB es una base de datos basadas en series de tiempos. Su cometido es almacenar y optimizar la recuperación datos organizados en mediciones efectuadas en intervalos regulares. Servirá para recolectar todos los datos obtenidos por telegraf.

Grafana es un entorno grafico que nos permite crear Dashboards o cuadros de mando a partir de entre otros, los datos almacenados en InfluxDB **[49]**.

Nos basaremos en un stack ya existente de esta solución para entornos x86 **[50]**, pero adaptaremos las imágenes Docker para poder usar las mismas aplicaciones en entornos de arquitectura ARM como las Raspberry pi 3.

4.1.2 Monitorización de Swarm con TIG

Modificaremos el fichero ‘docker-compose.yml’ para adaptarlo a nuestra arquitectura ARM y guardar la configuración en volúmenes montados en la carpeta ‘docker-config’ que se replica con syncthing y usaremos ‘hostname’ en telegraf para poder usar el nombre de Nodo Swarm en las estadísticas.

```yaml
version: "3.2"

services:
  telegraf:
    image: arm32v7/telegraf:1.3
    hostname: '{{.Node.Hostname}}'
    networks:
      - tig-net
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
      - /usr/local/docker-config/telegraf:/etc/telegraf
    deploy:
      restart_policy:
        condition: on-failure
      mode: global

  influxdb:
    image: arm32v7/influxdb:1.2
    ports:
      - "8083:8083"
      - "8086:8086"
    environment:
      - INFLUXDB_ADMIN_ENABLED=true
    networks:
      - tig-net
    volumes:
      - /usr/local/docker-config/influxdb:/var/lib/influxdb
    deploy:
      restart_policy:
        condition: on-failure

  grafana:
    image: fg2it/grafana-armhf:v4.3.2
    ports:
      - "3000:3000"
    networks:
      - tig-net
    volumes:
      - /usr/local/docker-config/grafana:/var/lib/grafana
    deploy:
      restart_policy:
        condition: on-failure
      placement:
        constraints:
          - node.role == manager

networks:
  tig-net:
    driver: overlay
```

De esta manera la monitorización y estadística funcionará en alta disponibilidad, soportando incluso la pérdida de uno de los nodos del Swarm.

Una vez lanzado el stack con ‘docker deploy’ como ‘tig_service’, comprobamos que se crean los servicios correctamente entre todos los nodos del Swarm.

```
joe@SmartHome1:/etc/cloudy/docker-compose/projects/tig_service $ sudo docker service ls
ID              NAME                            MODE         REPLICAS
xhzoefyi61ys    SmartHome_service_public_mqtt   replicated   1/1
vmuv56wl4z0r    SmartHome_service_public_node-red replicated 1/1
r25v69lpw025    syncthing_service_syncthing-arm global       3/3
5vjgwws41ti8    tig_service_grafana             replicated   1/1
oop9hh29filj    tig_service_influxdb            replicated   1/1
d22q7etuz7xd    tig_service_telegraf            global       3/3
```

Configuraremos los componentes siguiendo los pasos incluidos en la documentación y finalmente adaptaremos el Dashboard propuesto para poder crear un cuadro de mando adaptado a nuestras necesidades (Figura 64).

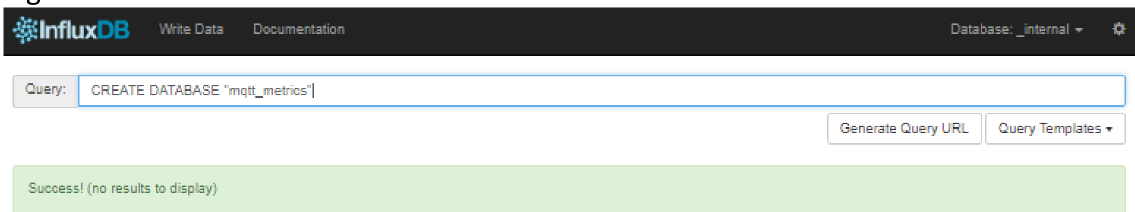
Figura 64: Swarm Dashboard

4.1.3 Monitorización de la microCloud

Una vez monitorizado el clúster Swarm, en un segundo paso, usaremos Node-Red para recopilar unas estadísticas de ejemplo de nuestra microCloud.

Primero crearemos una base de datos en influxDB para guardar las métricas que usaremos en MQTT. Nos conectaremos a ‘http://192.168.10.10:8083’ y crearemos la base de datos ‘mqtt_metrics’ (Figura 65).

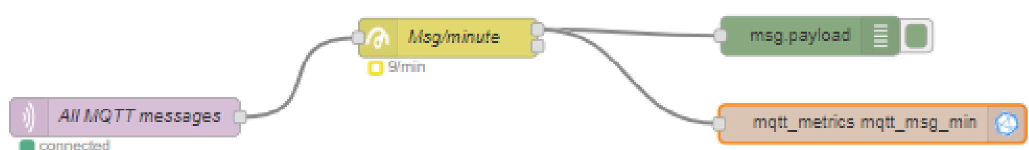
Figura 65: Crear BD influxDB



Instalaremos en Node-Red los nodos ‘node-red-contrib-influxdb’ y el nodo ‘node-red-contrib-msg-speed’ y crearemos una pestaña que llamaremos ‘estadisticas’ donde agruparemos los distintos flujos que crearemos para alimentar la base de datos influxDB .

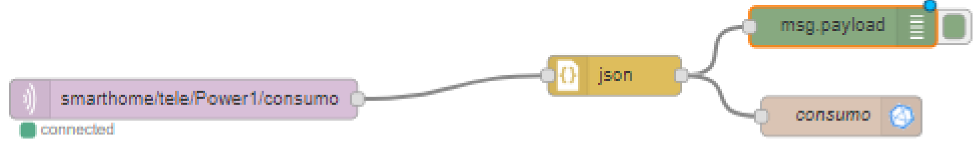
El primer flujo que crearemos necesitará la instalación del nodo ‘node-red-contrib-msg-speed’. El flujo recibirá todos los mensajes MQTT de Mosquitto, calculará el número de mensajes por minuto y lo guardará en la métrica ‘mqtt_msg_min’.

Figura 66: Estadística de mensajes por minuto



Otra estadística que recogeremos es el consumo de Power1 para guardar un histórico de los datos. En este caso escucharemos el mensaje MQTT con tópico ‘smarthome/tele/Power1/consumo’ que el dispositivo sonoff ‘Power1’ envía periódicamente al servidor Mosquitto para anunciar su consumo actual. Una vez recibida la lectura, la guardaremos en influxDB (Figura 67).

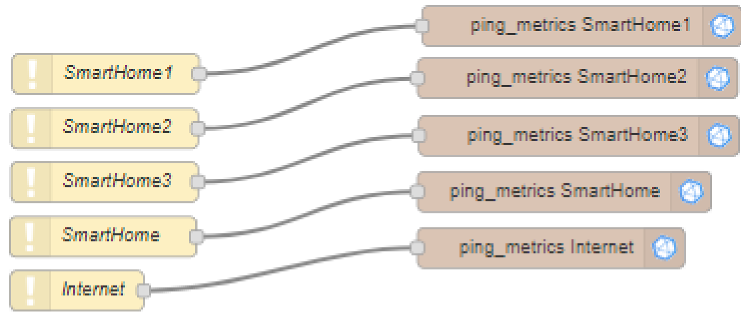
Figura 67: Estadística de Consumo de Sonoff



Finalmente recogeremos varias estadísticas de red para recopilar la disponibilidad de varias direcciones IP de nuestra microCloud y guardarlas en influxDB. En este caso crearemos una base de datos que llamaremos ‘ping_metrics’.

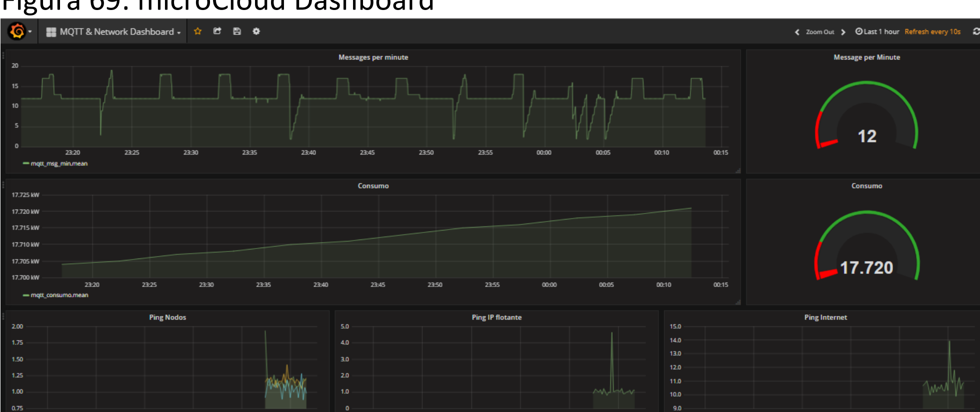
Para comprobar la disponibilidad de cada nodo, de la IP flotante y de Internet , instalaremos el nodo ‘node-red-contrib-ping’, que ejecuta el comando ping a intervalos regulares contra la dirección IP que le indiquemos. Crearemos también varios flujos en la pestaña ‘Estadisticas’, uno por cada métrica que queramos guardar en la base de datos (Figura 68).

Figura 68: Estadística de Pings



Para visualizar las estadísticas recopiladas en influxDB crearemos un Dashboard en grafana que llamaremos ‘microCloud Dashboard’ (Figura 69).

Figura 69: microCloud Dashboard



Estos son solo una muestra de las estadísticas que podemos crear con nuestro stack TIG. Guardaremos la definición de los flujos y de los Dashboard en los ficheros listados en el Anexo 5 de esta memoria.

4.2 Serverless computing con Openfaas

Para aprovechar los recursos ociosos de nuestro clúster Swarm, vamos a ejecutar OpenFaaS. La arquitectura FaaS (Function as a Service) pretende ir un paso más allá de los contenedores, creando un entorno en el que el cliente ejecuta funciones stateless disponibles en la Cloud.

Esta arquitectura abstrae completamente los servidores para el cliente y el creador de la aplicación siendo los servicios automáticamente escalados según las necesidades de los usuarios **[51]**.

OpenFaaS es un framework para crear funciones serverless con los orquestadores Kubernetes o Swarm. OpenFaaS proporciona las siguientes funcionalidades:

- Facilidad de uso con un portal de usuario e instalación con un click
- Posibilidad de escribir las funciones en cualquier lenguaje del sistema operativo soportado en el contenedor.
- Portabilidad, se ejecuta de forma nativa en Swarm o Kubernetes en Cloud privadas o públicas.
- Interfaz CLI con formato YAML para definir funciones y plantillas.
- Auto escalado de las funciones según el incremento de la demanda.
- Recopilación de estadísticas con Prometheus, es una Base de Datos de series de tiempos como influxDB que está integrada en OpenFaaS.

En nuestro caso podremos ejecutar funciones desde Node-red con el nodo 'node-red-contrib-openfaas' y visualizar las estadísticas en Grafana ya que podemos configurar Prometheus como una fuente de datos.

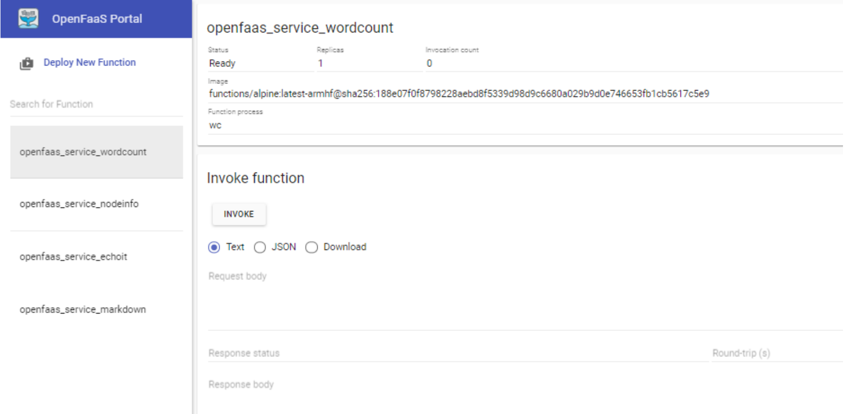
4.2.1 Instalar OpenFaaS

Seguiremos el tutorial para instalar OpenFaaS en arquitectura ARM que está incluido en el sitio GitHub de OpenFaaS **[52].** Al igual que TIG, OpenFaaS en un conjunto de servicios que se instala como un stack de Swarm.

Basta crear un fichero docker-compose.yml a partir del fichero 'https://github.com/openfaas/faas/blob/master/docker-compose.armhf.yml' y lanzarlo en el Swarm con el comando 'docker stack deploy'.

De esta manera se creará el stack OpenFaaS y algunas funciones de prueba. Podemos comprobar que se ejecuta correctamente el stack accediendo al portal OpenFaaS en el puerto 8080 del Swarm (Figura 70).

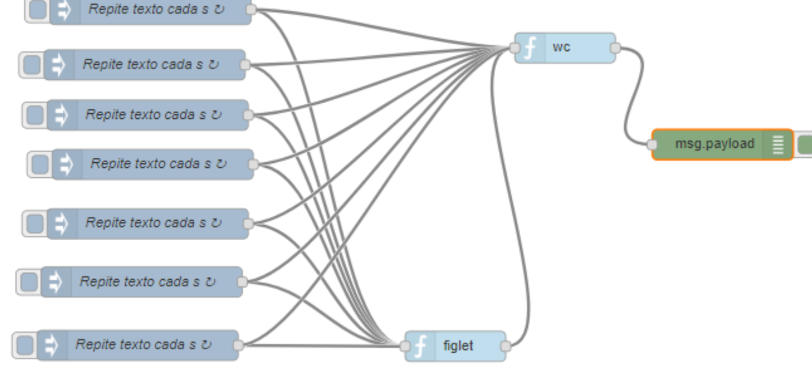
Figura 70: Portal OpenFaaS



Para usar OpenFaaS desde Node-Red instalaremos el nodo 'node-red-contrib-openfaas' que nos permitirá enviar mensajes de Node-Red como entrada a las funciones OpenFaas y recibir la salida procesada como mensajes.

En un principio para probar OpenFaaS ejecutaremos un sencillo flujo de Node-Red con la función 'wc' que encapsula el comando 'wc' de Linux y la función 'figlet' que transforma texto a logos ASCII (Figura 71).

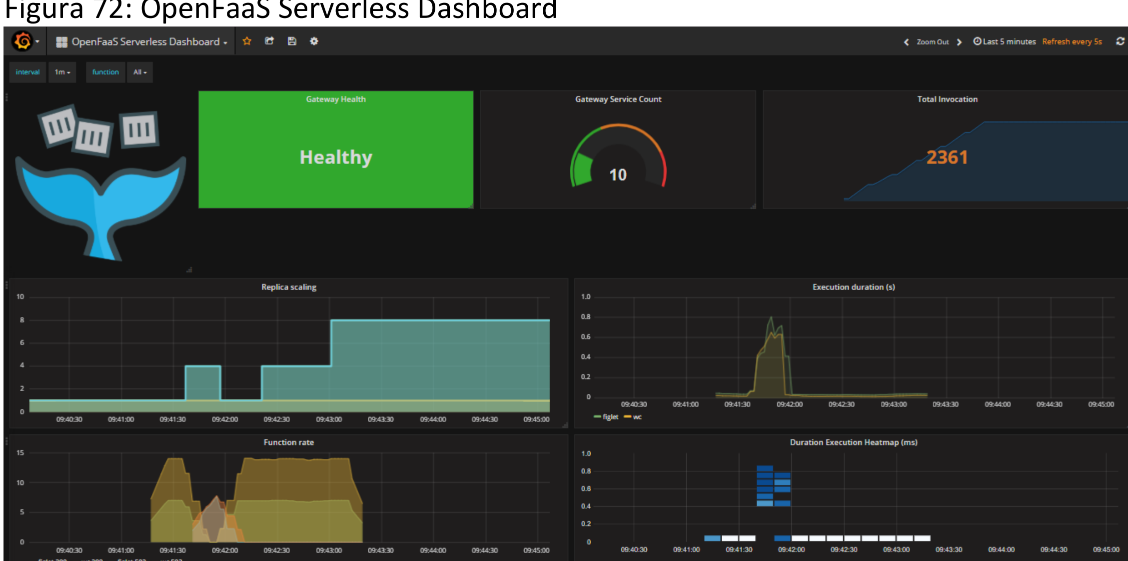
Figura 71: Flujo de prueba de OpenFaaS



Cada nodo ‘inject’ de este flujo, enviará una cadena de texto larga a la función ‘figlet’ y a la función ‘wc’ de esta manera generaremos una demanda de potencia de computación que será cubierta por OpenFaaS escalando las funciones dentro del Swarm.

Cargaremos un Dashboard en Grafana para visualizar las estadísticas de monitorización de OpenFaaS que se recogen en Prometheus (Figura 72).

Figura 72: OpenFaaS Serverless Dashboard



Comprobamos que según aumenta la demanda, OpenFaaS va creando contenedores para ejecutar las funciones de prueba en nuestro Swarm.

4.2.2 Crear una función Serverless de ejemplo

Crearemos una función OpenFaaS con el comando ‘jp2a’ **[53]** para convertir imágenes jpeg a ficheros ASCII. Primero crearemos un fichero Dockerfile que llamaremos ‘Dockerfile.armhf’ para crear una imagen con el sistema operativo Linux Debian y el comando ‘jp2a’.

```
FROM armv7/armhf-debian

RUN apt-get update
RUN apt-get -y install jp2a

ADD  https://github.com/openfaas/faas/releases/download/0.8.0/fwatchdog-armhf
/usr/bin/fwatchdog
RUN chmod +x /usr/bin/fwatchdog

ENV TERM dumb
ENV fprocess "jp2a"

EXPOSE 8080

HEALTHCHECK --interval=1s CMD [ -e /tmp/.lock ] || exit 1
CMD [ "/usr/bin/fwatchdog"]
```

Crearemos la imagen con el comando ‘docker build’

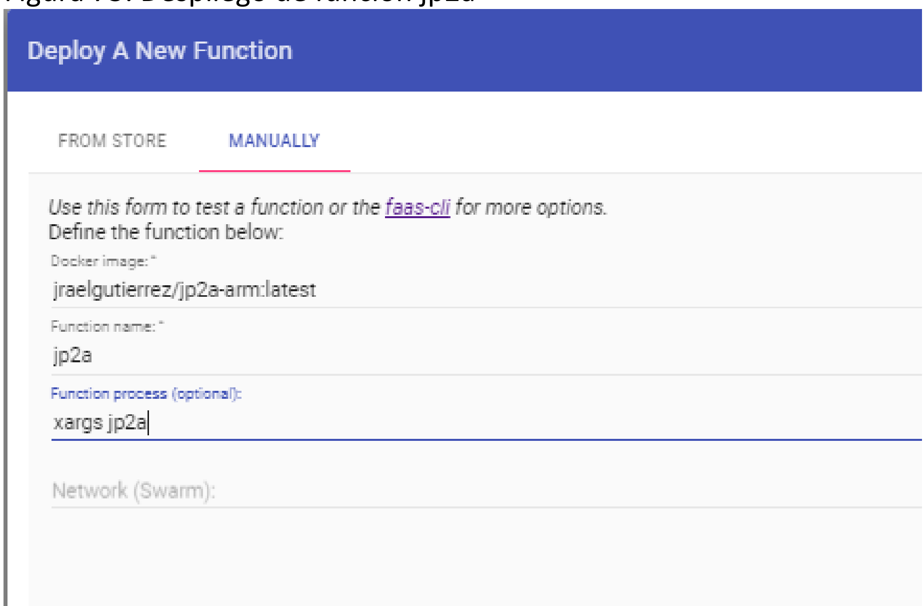
```
sudo docker build  .  -f Dockerfile.armhf -t jp2a
```

Guardaremos la imagen creada en Docker Hub para poder usarla posteriormente en cualquiera de los nodos. Primero haremos Login en Docker Hub, etiquetaremos la imagen y la subiremos al repositorio.

```
joe@SmartHome1:~/functions/jp2a $ sudo docker login
Login with your Docker ID to push and pull images from Docker Hub. If you
don't have a Docker ID, head over to https://hub.docker.com to create one.
Username (jraelgutierrez):
Password:
Login Succeeded
joe@SmartHome1:~/functions/curl $ sudo docker tag jpa2a  jraelgutierrez/jp2a-
arm
joe@SmartHome1:~/functions/curl $ sudo docker push jraelgutierrez/jp2a-arm
```

Finalmente, solo nos queda crear la función con la nueva imagen desde la UI de OpenFaaS (Figura 73).

Figura 73: Despliego de función jp2a



Podemos ejecutar la función con el comando ‘curl’ y una URL de ejemplo que apunta a una imagen en formato jpeg

```
curl 192.168.10.15:8080/function/jp2a -d "--color --width=80 https://upload.w
ikimedia.org/wikipedia/commons/thumb/4/4d/Cat_November_2010-1a.jpg/220px-Cat_
November_2010-1a.jpg"
```

Y comprobamos que la salida es correcta. En este caso el fichero se volcará a la salida estándar, pero podemos redirigir la salida a un fichero de texto para comparar el resultado con la imagen original (Figura 74).

Figura 74: Resultado de la función jp2a

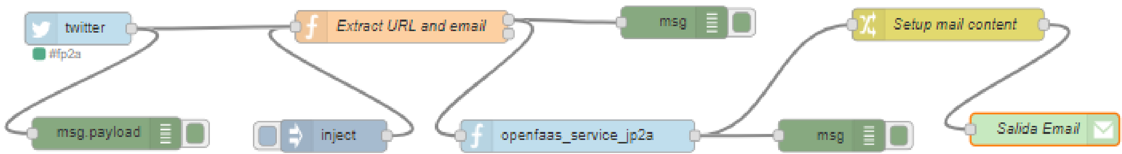
4.2.3 Crear una aplicación Serverless

Para crear la aplicación Serverless, usaremos la función 'jp2a' que hemos creado en el apartado anterior para hacer crear un sencillo servicio de conversión de imágenes JPEG a ASCII.

Crearemos un flujo Node-red (Figura 75) en el que esperamos que alguien envíe por twitter un mensaje con hastag '#jp2a' y un texto como 'enviar <url-imagen-jpeg> a correo <email>', si el mensaje contiene una URL valida, la función OpenFaaS 'jp2a' convertirá la imagen en texto.

El texto generado se enviará por correo al email que hemos indicado en el twitter. Este servicio se puede usar desde cualquier cliente twitter, y puede escalar según el número de usuarios que lo use en un instante.

Figura 75: Flujo de prueba OpenFaaS con jp2a



Este flujo demuestra el gran poder de integración de servicios que proporciona Node-red, además el flujo propuesto se puede adaptar posteriormente para recibir la cadena de entrada a través de email, un Web Service o sencillamente un pequeño formulario web que se puede generar a través de los componentes de Dashboard del propio Node-Red.

4.3 Conclusiones de la fase 3

Esta tercera fase ha sido especialmente satisfactoria al constatar lo rápido que se puede crear o adaptar sistemas complejos con varios contenedores con un único fichero de configuración en un entorno con alta disponibilidad.

Ha quedado demostrado que nuestra microCloud abierta, se puede escalar para ejecutar funciones avanzadas aparte de realizar la función de pasarela IoT. El sistema nos ha permitido desplegar aplicaciones complejas compuestas por grupos de servicios tales como analítica y visualización de datos o una arquitectura FaaS.

El stack de aplicaciones TIG ha demostrado la capacidad de recopilar, analizar y visualizar datos de forma sencilla, con el apoyo de una comunidad de usuarios que comparten información y cuadros de mando de forma abierta. Es notable la facilidad con la hemos podido instalar un cuadro de mando para OpenFaaS.

Es de resaltar el gran potencial de OpenFaaS para proporcionar una arquitectura FaaS en redes domesticas o comunitarias. En un futuro, FaaS nos permitirá desarrollar un tipo de aplicaciones a nivel comunitario que no se han podido crear hasta ahora salvo en grandes entornos corporativos.

5 Valoración de la solución desarrollada

En este apartado valoraremos la microCloud desarrollada durante el proyecto y las implicaciones de los resultados en diferentes ámbitos de aplicación tales como uso en redes domésticas y uso en redes comunitarias.

5.1 microCloud en redes domésticas

Raspberry pi y la reciente disponibilidad de Docker y Swarm para arquitecturas ARM ha permitido el que tengamos acceso a la posibilidad de ejecutar un clúster de tres nodos Docker en alta disponibilidad en un entorno de red doméstica por un precio de menos de 150 Euros.

La alta disponibilidad nos permite tener servicios funcionando casi continuadamente y efectuar operaciones nunca vistas antes en entornos caseros tales como actualizar equipos y sistemas operativos sin pérdida de servicio.

La encapsulación de servicios en los contenedores nos permite reemplazar fácilmente el hardware y el software de forma independiente de nuestra configuración. También nos permite hacer coexistir de forma fácil servicios que coinciden en el mismo puerto sin tocar la configuración del servicio.

La recuperación de un fallo completo de un nodo se cifra en segundos. Esto permite desarrollar un nuevo tipo de aplicaciones críticas en el hogar para el que antes sencillamente no existía un entorno capaz de ejecutarlas.

La escalabilidad y versatilidad de la microCloud nos da la capacidad de desplegar entornos complejos. Esto permite ejecutar aplicaciones de varios niveles incluso a usuarios no técnicos, por ejemplo, podríamos ejecutar todos los niveles del Stack IoT en nuestro entorno local.

5.2 microCloud en redes comunitarias

En todo caso primero necesitaremos resolver el problema de pérdida de paquetes de red que hemos descubierto debido a la incompatibilidad entre Avahi, Serf, Raspbian y el hardware de la Raspberry ya que Serf es imprescindible para publicar servicios en redes comunitarias

También hay que replantearse la forma de publicar servicios, ya que usando una microCloud con alta disponibilidad los servicios no están ligados obligatoriamente a un nodo en concreto sino a una IP flotante que puede desplazarse entre varios nodos.

La primera aportación que puede proporcionar nuestra microCloud a las redes comunitarias es, al igual que en redes domésticas, el ofrecer servicios complejos en alta disponibilidad con un mínimo coste y esfuerzo.

En redes comunitarias centralizadas podemos extender nuestra microCloud a múltiples ubicaciones con el uso de una VPN, creando uno o varios clústeres

Swarm distribuidos que pueden extender cualquier servicio de forma masiva por la infraestructura de red..

En redes más descentralizadas, podemos compartir datos y sincronía entre microClouds pertenecientes a propietarios distintos, en este caso usaremos el protocolo MQTT junto con Node-Red, que nos da flexibilidad, facilidad de uso y seguridad para controlar exactamente los datos y funciones que queremos hacer públicos.

En caso de querer proporcionar potencia de computación a otros usuarios, hay que tener en cuenta de que, en este caso, la unidad mínima para compartir entre usuarios es un nodo entero, que corresponde a una Raspberry pi. Esto será así hasta que Docker incorpore la posibilidad de crear usuarios y roles de acceso a nivel de contenedor, volúmenes y redes.

La posibilidad de ejecutar una arquitectura FaaS en la microCloud puede ser la base para crear un nuevo tipo de aplicaciones masivamente escalables que soporten un gran número de usuarios de forma sencilla y distribuida.

A la hora de crear servicios públicos en una red comunitaria, tenemos la complejidad añadida de configurar el firewall para permitir publicar los servicios externos. Sería interesante añadir una funcionalidad para generar automáticamente las reglas del firewall que nos faciliten la publicación de servicios de los contenedores Docker.

6 Conclusiones Finales

En la primera fase, hemos creado un sistema completamiento abierto utilizando y combinando distintos proyectos de software libre y hardware abierto, que nos han proporcionado una libertad total para usar nuestros datos y nuestros dispositivos.

En la segunda fase hemos investigado el uso de herramientas y tecnologías avanzadas para hacer progresar nuestra instalación a otro nivel, implementando una solución efectiva de alta disponibilidad y estudiando la extensión de la microCloud mediante nodos externos y programación distribuida.

En la tercera fase, hemos explorado nuevas aplicaciones para la infraestructura que hemos desarrollado en las fases anteriores. Poniendo en marcha soluciones de monitorización y análisis con grafana y de infraestructuras FaaS con OpenFaaS.

Tal como hemos reiterado varias veces durante el proyecto, la idea de plantear varias fases ha sido un completo acierto, que nos ha permitido obtener una solución funcional desde la primera fase, lo que nos ha proporcionado una base para ir perfeccionando nuestra microCloud.

Este es el espíritu real del software abierto que era el objetivo de nuestro proyecto: coger una solución limitada, superarla y compartir el esfuerzo entre todos los demás usuarios para hacer avanzar la comunidad en su conjunto.

Como punto negativo los problemas de Red que nos hemos encontrado en la ejecución de Raspbian 9 en Raspberry pi, nos ha hecho perder tiempo y me deja con un sabor agridulce al no poder haber resuelto completamente el problema de compatibilidad con Serf por falta de tiempo al acercarse la fecha de entrega del proyecto.

La planificación se ha mantenido de forma escrupulosa salvo en el último tramo del proyecto debido a los problemas de red citados anteriormente, sin embargo, al tener un margen de 8 días antes de la entrega final del proyecto este retraso no ha supuesto ningún inconveniente.

La microCloud tras finalizar la tercera fase está ejecutando unos 20 contenedores usando menos de un 6 por ciento de CPU en cada nodo con un uso entre un 30% y un 50% de la memoria. El Swarm final de 3 nodos está compuesto de 12 cores con 3GB de memoria en total.

Nuestra microCloud es completamente funcional, pero debido a la corta duración del proyecto, se han quedado algunos temas pendientes tales como asegurar algunas aplicaciones añadiéndole credenciales de acceso (por ejemplo, Node-red y Syncthing), y estudiar si se puede dejar de usar un usuario privilegiado para replicar los datos de todos los contenedores.

7 Glosario

ARM — ARM es una arquitectura RISC (Reduced Instruction Set Computer u Ordenador con Conjunto Reducido de Instrucciones) desarrollada por ARM Holdings.

API — Acrónimo de Aplicación Programming Interface o Interfaz de programación de la aplicación. Es un conjunto de rutinas que provee acceso a funciones de un determinado software

Avahi — Avahi es un protocolo libre que permite a los programas publicar y descubrir servicios y hosts que se ejecutan en una red local.

Bridging — Interconectar segmentos de red haciendo la transferencia de datos de una red hacia otra con base en la dirección física de destino de cada paquete, formando efectivamente una única subred.

Bróker de Mensajes o Bróker — Un bróker de mensajería es un mecanismo mediador de la comunicación entre aplicaciones, permitiendo minimizar el grado de conocimiento mutuo que estas aplicaciones necesitan tener, para poder intercambiar mensajes, implementando así efectivamente su desacoplamiento.

Cloud o Cloud Computing — La computación en la nube (del inglés Cloud computing), conocida también como simplemente Cloud o "la nube", es un paradigma que permite ofrecer servicios de computación a través de una red, que usualmente es Internet.

Cloudy — Cloudy es una distribución GNU/Linux basada en Debian, destinada a los usuarios, para fomentar la transición y adopción del entorno Cloud en las redes comunitarias.

Clúster — es un conjunto de dos o más máquinas que se caracterizan por mantener una serie de servicios compartidos y por estar constantemente monitorizándose entre sí.

Cronograma — Un cronograma es, en gestión de proyectos, una lista de todos los elementos terminales de un proyecto con sus fechas previstas de comienzo y final.

CWAN — Acrónimo de Community Wide Area Network. Es una red comunitaria que une varias redes locales para compartir servicios entre todos sus usuarios de forma libre. Un ejemplo de este tipo de redes es Guifi.net

Deviot — Plugin de desarrollo IoT para el editor Sublime Text que gestiona el entorno de desarrollo PlatformIO.

Diagrama de Gantt El diagrama de Gantt es una herramienta gráfica cuyo objetivo es exponer el tiempo de dedicación previsto para diferentes tareas o actividades a lo largo de un tiempo total determinado.

Docker Docker es un proyecto de código abierto que automatiza el despliegue de aplicaciones dentro de contenedores de software, proporcionando una capa adicional de abstracción y automatización de Virtualización a nivel de sistema operativo en Linux.

Docker HUB Docker HUB es un servicio de registro de imágenes de contenedores en la Nube que permite compartir Imágenes y descargarlas directamente en un Host Docker.

DS18b20 El termómetro digital DS18B20 proporciona lecturas de temperatura de 9 a 12 bits (configurables) que indican la temperatura del dispositivo.

Edge Computing Es un método para optimizar los sistemas de computación en la nube al realizar el procesamiento de datos en equipos situados en el límite de la red local del usuario final, cerca de la fuente de los datos.

EDT Una estructura de descomposición del trabajo (EDT), también conocida por su nombre en inglés Work Breakdown Structure o WBS, es una herramienta fundamental que consiste en la descomposición jerárquica, orientada al entregable, del trabajo a ser ejecutado por el equipo de proyecto.

Enrutador o Router Un router —también conocido como enrutador — es un dispositivo cuya función principal consiste en enviar o encaminar paquetes de datos de una red a otra

Firmware El firmware es un programa informático que establece la lógica de más bajo nivel que controla los circuitos electrónicos de un dispositivo de cualquier tipo. Está fuertemente integrado con la electrónica del dispositivo, es el software que tiene directa interacción con el hardware, siendo así el encargado de controlarlo para ejecutar correctamente las instrucciones externa

Fog Computing Es una arquitectura que utiliza de manera colaborativa varios equipos situados cerca de los usuarios finales para proporcionar servicios de computación más potentes optimizando el uso de todos los recursos compartidos.

FTDI Es un dispositivo para la conversión de transmisiones serie RS-232 o TTL a señales USB desarrollado por la empresa FTDI (Future Technology Devices International). Se llama comúnmente a todos los dispositivos que tienen esta función.

FTP El Protocolo de transferencia de archivos (en inglés File Transfer Protocol o FTP), es un protocolo de red para la transferencia de archivos entre sistemas conectados a una red TCP

GitHub GitHub es una plataforma de desarrollo colaborativo para alojar proyectos utilizando el sistema de control de versiones Git.

GPIO GPIO (General Purpose Input/Output, Entrada/Salida de Propósito General) es un pin genérico en un chip, cuyo comportamiento (incluyendo si es un pin de entrada o salida) se puede controlar (programar) por el usuario en tiempo de ejecución.

Headless Termino informático que indica que un ordenador es capaz de funcionar sin pantalla ni teclado.

HTTP El Protocolo de transferencia de hipertexto (en inglés: Hypertext Transfer Protocol o HTTP) es el protocolo de comunicación que permite las transferencias de información en la World Wide Web

IoT Acrónimo de Internet of Things o Internet de las Cosas. Es un concepto que se refiere a la interconexión digital de objetos cotidianos a través de redes informáticas.

IoT World Forum El IoT World Forum (IoTWF) es un evento exclusivo de la industria, organizado por Cisco. IoTWF reúne a los líderes de la industria para colaborar, establecer redes, asociarse y resolver los desafíos que ofrece la IoT.

IP El protocolo de Internet (en inglés Internet Protocol o IP) es un protocolo de comunicación de datos digitales situado en la capa de red

IP Failover Es una técnica que consiste en trasladar automáticamente una IP a otro equipo en caso de malfuncionamiento del equipo principal

ISO La Organización Internacional de Normalización (originalmente en inglés: International Organization for Standardization, conocida por las siglas ISO) es una organización para la creación de estándares internacionales

Javascript JavaScript es un lenguaje de programación interpretado. Se define como orientado a objetos, basado en prototipos, imperativo, débilmente tipado y dinámico

LAN Una LAN o red de área local (por las siglas en inglés de Local Area Network) es una red de computadoras que abarca un área reducida a una casa, un departamento o un edificio.

Linux	Linux es un núcleo mayormente libre semejante al núcleo de Unix. Es uno de los principales ejemplos de software libre y de código abierto, está licenciado bajo la licencia GPL v2.
Login	En el ámbito de seguridad informática, login o logon (en español ingresar o entrar) es el proceso mediante el cual se controla el acceso individual a un sistema informático mediante la identificación el usuario.
LoT	Es el acrónimo de LAN of Things, o Red de Area Local de las cosas. En contraste con la IoT que conecta los dispositivos a nivel de Internet, conecta los dispositivos inteligentes a nivel de Area Local.
Lubuntu	Distribución de Linux ligera orientada a equipos con pocos recursos
M2M	M2M (machine to machine o 'máquina a máquina') es un concepto genérico que se refiere al intercambio de información o comunicación en formato de datos entre dos máquinas remotas.
microCloud	En contraste con la Cloud, es una Infraestructura ligera que ofrece servicios con equipos poco potentes y de bajo coste, normalmente a pocos usuarios dentro de una red local.
microSD	Tarjeta de memoria flash de tamaño de solo 15×11×1 milímetros,
modo Swarm de Docker	Es una funcionalidad de Docker que proporciona capacidades de orquestación, creación de clúster de Hosts Docker y programación de cargas de trabajo en contenedores
Mosquitto	Mosquitto o Eclipe Mosquitto, es un bróker de mensajes MQTT de código abierto que implementa el protocolo MQTT 3.1 y 3.1.1.
MQTT	MQTT acrónimo de Message Queuing Telemetry Transport es un estándar ISO de un protocolo de mensajes basado en publicación/subscripción sobre redes TCP/IP. Está diseñado para conectar dispositivos con bajos recursos en ubicaciones remotas con ancho de banda limitado.
NAT	Acrónimo del inglés Network Address translation, es un mecanismo utilizado por routers IP para intercambiar paquetes entre dos redes que asignan mutuamente direcciones incompatibles.
Node.js	Es un entorno en tiempo de ejecución multiplataforma de código abierto basado en el lenguaje de programación ECMAScript.

Node-Red Es una herramienta de programación visual basada en flujos desarrollada inicialmente por IBM.

NTP Network Time Protocol (NTP) es un protocolo de Internet para sincronizar los relojes de los sistemas informáticos a través del enrutamiento de paquetes en redes con latencia variable.

Open Source Open Source, Software Libre o Código Abierto es un modelo de desarrollo de software basado en la colaboración abierta entre los usuarios. El término «abierto» hace referencia a la libertad de poder modificar el código fuente del programa.

OTA Acrónimo de 'Over the Air', o comunicación inalámbrica.

Platformio Un ecosistema abierto para desarrollo de IoT, con un IDE multiplataforma, un depurador unificado, tests de desarrollo remotos y actualización de firmware integradas.

PMI El Project Management Institute (PMI) es una organización estadounidense sin fines de lucro que asocia a profesionales relacionados con la Gestión de Proyectos.

PMP Project Management Professional (PMP) es una certificación de gestión de proyectos ofrecida por el Project Management Institute(PMI).

Routing Mesh Funcionalidad de Docker Swarm que permite acceder a todos los contenedores desde un host desde cualquier nodo, distribuyendo la carga del servicio entre contenedores replicados.

Rsync Rsync es una aplicación libre para sistemas de tipo Unix y Microsoft Windows que ofrece transmisión eficiente de datos incrementales, permite sincronizar archivos y directorios entre dos máquinas de una red o entre dos ubicaciones en una misma máquina, minimizando el volumen de datos transferidos.

Safe@Office Safe@Office es una línea de firewall y de appliances VPN de la empresa Sofaware. Subsidiaria de la empresa Checkpoint

SBC Acrónimo de Single Board Computer, u Ordenador de tarjeta Única. Es un ordenador completo en un único circuito electrónico de tamaño reducido.

Serf Herramienta de creación de Clúster, descentralizada, con detección y tolerancia a fallos y alta disponibilidad.

SoC	Un sistema en chip o SoC (del inglés system on a chip o system on chip), describe la tendencia cada vez más frecuente de usar tecnologías de fabricación que integran todos o gran parte de los módulos que componen un computador o cualquier otro sistema informático o electrónico en un único circuito integrado o chip.
Sonoff	Gama de dispositivos domóticos de la empresa ITEAD basados en el SOC ESP8266
SSH	Secure Shell (SSH) es un protocolo de red cifrado para operar servicios de red de forma segura a través de una red no segura. La aplicación de ejemplo más conocida es para el inicio de sesión remoto en sistemas informáticos
Tasmota	O Sonoff-Tasmota, es un firmware alternativo para dispositivos basados en SOCs ESP8266. Soporta entre otros los dispositivos SonOffs de ITEAD.
TCP	Protocolo de control de transmisión (en inglés Transmission Control Protocol o TCP), es uno de los protocolos fundamentales en Internet, orientado a conexión y situado en la capa de transporte
TTL	TTL es la sigla en inglés de transistor-transistor logic, es decir, «lógica transistor a transistor». Es una tecnología de construcción de circuitos electrónicos digitales.
UDP	Es un protocolo del nivel de transporte basado en el intercambio de datagramas. Permite el envío de datagramas a través de la red sin que se haya establecido previamente una conexión.
USB	Es un bus de comunicaciones que sigue un estándar que define los cables, conectores y protocolos usados en un bus para conectar, comunicar y proveer de alimentación eléctrica entre computadoras, periféricos y dispositivos electrónicos.
WebDAV	Este protocolo proporciona funcionalidades para crear, cambiar y mover documentos en un servidor remoto (típicamente un servidor web).
WebSocket	WebSocket es una tecnología que proporciona un canal de comunicación bidireccional y full-duplex sobre un único socket TCP. Está diseñada para ser implementada en navegadores y servidores web, pero puede utilizarse por cualquier aplicación cliente/servidor.
Wifi	Wifi es una tecnología que permite la interconexión inalámbrica de equipos electrónicos cumpliendo el estándar IEE 802.11 relacionado con redes inalámbricas de área local.

WLAN	Una WLAN es una LAN que usa tecnología inalámbrica.
YAML	YAML es un formato de serialización de datos legible por humanos inspirado en lenguajes como XML, C, Python, Perl, así como el formato para correos electrónicos especificado en RFC 2822.
Zigbee	Zigbee es nombre de un conjunto de protocolos de alto nivel de comunicación inalámbrica para IoT, basada en el estándar IEEE 802.15.4 de redes inalámbricas de área personal (wireless personal área network, WPAN). Su objetivo son las aplicaciones que requieren comunicaciones seguras con baja tasa de envío de datos y maximización de la vida útil de sus baterías.

8 Bibliografía

[1] Williams, Sam (2002). Free as in Freedom: Richard Stallman's Crusade for Free Software. O'Reilly Media. ISBN 0-596-00287-4. Chapter 1. Available under the GFDL in both the initial O'Reilly edition (accessed on October 27, 2006) and the updated FAIFzilla edition. Retrieved October 27, 2006.

[2] Flinley, K. (25 de agosto de 2016). *LINUX TOOK OVER THE WEB. NOW, IT'S TAKING OVER THE WORLD*.
Recuperado el 1 de abril de 2018, de
https://www.wired.com/2016/08/linux-took-web-now-taking-world/

[3] Moya, P. (29 de febrero de 2016). *Todo sobre la nueva Raspberry Pi 3, más potente y conectada que nunca*.
Recuperado el 1 de abril de 2018, de
https://omicrono.elespanol.com/2016/02/raspberry-pi-3-model-b/

[4] Docker. (s.f.). *Docker Overview*.
Recuperado el 1 de abril de 2018, de
https://docs.docker.com/engine/docker-overview/

[5] Baig, R. &. (17 de diciembe de 2017). *Cloudy in guifi.net: Establishing and sustaining a community cloud as open commons.*
Recuperado el 1 de abril de 2018, de
https://doi.org/10.1016/j.future.2017.12.017

[6] Varios. (2013). *A Guide to the Project Management Body of Knowledge (PMBOK® Guide), Fifth Edition*. PMI Institute. ISBN-10: 1935589679; ISBN-13: 978-1935589679. Chapter 5. Scope Management.

[7] Cisco. (4 de junio de 2014). *Building the Internet of Things.*
Recuperado el 1 de abril de 2018, de
http://cdn.iotwf.com/resources/72/IoT_Reference_Model_04_June_2014.pdf

[8] Cisco. (2015). *Fog Computing and the Internet of Things: Extend the Cloud to Where the Things Are*.
Recuperado el 05 de junio de 2018 de
https://www.cisco.com/c/dam/en_us/solutions/trends/iot/docs/computing-overview.pdf

[9] Vera Control Ltd. (s.f.). *Vera Smarter Home Control.*
Recuperado el 2 de abril de 2018, de
http://getvera.com/

[10] ITEAD. (s.f.). *Sonoff Wiki*.
Recuperado el 2 de abril de 2018, de
https://www.itead.cc/wiki/Sonoff

[11] Andrew Banks, R. G. (29 de octubre de 2014). *MQTT Version 3.1.1.*
Recuperado el 1 de abril de 2018, de
http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.pdf

[12] Antonino Virgillito. (2003). *Publish/Subscribe Communication Systems: from Models to Applications*.
Recuperado el 5 de junio de 2018, de
https://www.cs.ucf.edu/~dcm/Teaching/COT4810-Spring2011/Literature/PublishSubscribe-thesis.pdf

[13] Checkpoint. (October 2008). *Check Point Safe@Office Internet Security Appliance User Guide Version 8.0*.
Recuperado el 5 de junio de 2018, de
http://dl3.checkpoint.com/paid/77/CheckPoint_Safe@Office_8_UserGuide.pdf?HashKey=1528157913_eafd155ac140171236ad6f1c4f632202&xtn=.pdf

[14] Expressif. (s.f.). *ESP8266EX Datasheet*.
Recuperado el 1 de abril de 2018, de
https://www.espressif.com/sites/default/files/documentation/0a-esp8266ex_datasheet_en.pdf

[15] Wemos. (s.f.). *D1 mini A mini wifi board with 4MB flash based on ESP-8266EX*. Recuperado el 2 de abril de 2018, de
https://wiki.wemos.cc/products:d1:d1_mini

[16] Arendst, T. (s.f.). *Provide ESP8266 based itead Sonoff with Web, MQTT and OTA firmware using Arduino IDE or PlatformIO*.
Recuperado el 1 de abril de 2018, de
https://github.com/arendst/Sonoff-Tasmota

[17] Pueyo, R. (s.f.). *A script to convert your Debian system into a Commnunity Networking cloud in a box*.
Recuperado el 2 de abril de 2018, de
https://github.com/Clommunity/cloudynitzar

[18] Docker. (s.f.). *Swarm mode overview*.
Recuperado el 1 de abril de 2018, de
https://docs.docker.com/engine/swarm/

[19] Suse. (22 de marzo de 2018). *Why do you want to invest in containers.*
Recuperado el 2 de abril de 2018 de
https://www.suse.com/media/presentation/1.2_why_do_you_want_to_invest_in_containers.pdf

[20] Light, R. A. (s.f.). *Mosquitto: server and client implementation of the MQTT protocol.* Recuperado el 1 de abril de 2018, de http://joss.theoj.org/papers/10.21105/joss.00265

[21] Varios. (s.f.). *Node-RED Flow-based programming for the Internet of Things*.
Recuperado el 1 de abril de 2018, de
https://nodered.org/

[22] Mostovoy, V. V. (4 de abril de 2018). *MQTT Dash*.
Recuperado el 2 de abril de 2018 de
https://play.google.com/store/apps/details?id=net.routix.mqttdash&hl=es_419

[23] Mackenzie, J. (2 de enero de 2017). *Headless Raspberry Pi Setup*.
Recuperado el 2 de abril de 2018, de
https://hackernoon.com/raspberry-pi-headless-install-462ccabd75d0

[24] Pueyo, Roger. (5 de marzo de 2018). *Cloudynitzar.sh*.
Recuperado el 2 de abril de 2018, de
https://github.com/Clommunity/cloudynitzar/blob/master/cloudynitzar.sh

[25] Varios. (s.f.). *The JavaFX based MQTT Client.*
Recuperado el 2 de abril de 2018, de
http://mqttfx.jensd.de/

[26] Varios. (s.f.). *ITEAD Smart Home*.
Recuperado el 2 de abril de 2018, de
https://www.itead.cc/smart-home.html

[27] Varios. (s.f.). *Hardware Preparation*.
Recuperado el 2 de abril de 2018, de
https://github.com/arendst/Sonoff-Tasmota/wiki/Hardware-Preparation

[28] Varios. (s.f.). *Cloudy en nodo cliente*.
Recuperado el 6 de junio de 2018, de
http://es.wiki.guifi.net/wiki/Cloudy_en_nodo_cliente

[29] Varios. (10 de Septiembre 2010). *CADAVER, POTENTE Y SENCILLO CLIENTE DE WEBDAV*.
Recuperado el 6 de junio de 2018, de
http://www.matrosphera.com/cadaver-cliente-webdav/

[30] Smyth, David G. (22 de febrero de 2016). *Configurar WebDAV en Windows 10*.
Recuperado el 6 de junio de 2018, de
https://www.smythsys.es/7421/configurar-weddav-en-windows-10/

[31] Varios. (s.f.). '*node-red-contrib-viera'*.
Recuperado el 6 de junio de 2018, de
https://www.npmjs.com/package/node-red-contrib-viera

[32] Varios. (s.f.). '*node-red-contrib-sensor-ds18b20'*.
Recuperado el 6 de junio de 2018, de
https://www.npmjs.com/package/node-red-contrib-sensor-ds18b20

[33] Cisco. (s.f.). *Funciones y Funcionalidad de Hot Standby Router Protocol*.
Recuperado el junio de 2018, de
https://www.cisco.com/c/es_mx/support/docs/ip/hot-standby-router-protocol-hsrp/9234-hsrpguidetoc.pdf

[34] Hinden, R. & Nokia, E. (2004). *Virtual Router Redundancy Protocol (VRRP)*.
Recuperado el 6 de junio de 2018, de
https://tools.ietf.org/html/rfc3768

[35] Danhieux, Pieter. (24 de junio de 2004). *CARP, The Free Fail-over Protocol*.
Recuperado el 6 de junio de 2018, de
https://cyber-defense.sans.org/resources/papers/gsec/carp-free-fail-over-protocol-106433

[36] Cassen, Alexandre. (2002). *Keepalived User Guide*.
Recuperado el 6 de junio de 2018, de
http://www.keepalived.org/pdf/UserGuide.pdf

[37] Bunner, Tobias. (29 de julio de 2013). *Keepalived Check and Notify Scripts*.
Recuperado el 6 de junio de 2018, de
https://tobrunet.ch/keepalived-check-and-notify-scripts/

[38] Docker. (s.f.). *Publish a port for a service*.
Recuperado el 7 de junio de 2018 de
https://docs.docker.com/engine/swarm/ingress/#publish-a-port-for-a-service

[39] Docker. (s.f.). *How nodes work* .
Recuperado el 7 de junio de 2018 de
https://docs.docker.com/engine/swarm/how-swarm-mode-works/nodes/

[40] Varios. (s.f.). *The Raft Consensus Algorithm*.
Recuperado el 7 de junio de 2018 de
https://raft.github.io/

[41] Docker. (s.f.). *Docker Cloud stack file YAML reference*.
Recuperado el 7 de junio de 2018 de
https://docs.docker.com/docker-cloud/apps/stack-yaml-reference/

[42] Docker. (s.f.). *More details about mount types*.
Recuperado el 7 de junio de 2018 de
https://docs.docker.com/storage/#more-details-about-mount-types

[43] Varios. (s.f.). *Docker multi-tenant security*.
Recuperado el 7 de junio de 2018 de
https://forums.docker.com/t/docker-multi-tenant-security/45173

[44] Docker. (s.f.). *Open protocols and ports between the hosts*.
Recuperado el 7 de junio de 2018 de
https://docs.docker.com/engine/swarm/swarm-tutorial/#open-protocols-and-ports-between-the-hosts

[45] Tinc. (s.f.). *Example: bridging Ethernet segments using tinc under Linux*.
Recuperado el 7 de junio de 2018 de
https://www.tinc-vpn.org/examples/bridging/

[46] Giang, Nam. (s.f.). *Distributed Node-RED (DNR) Editor*.
Recuperado el 7 de junio de 2018 de
https://github.com/namgk/dnr-editor

[47] Varios. (s.f.). *ethernet interface interrupts connection*.
Recuperado el 7 de junio de 2018 de
https://raspberrypi.stackexchange.com/questions/79359/ethernet-interface-interrupts-connection

[48] Nociones.de. (s.f.). *Monitoriza tu sistema con telegraf, grafana e influxdb*.
Recuperado el 7 de junio de 2018 de
https://www.nociones.de/monitoriza-tu-sistema-con-telegraf-grafana-e-influxdb/

[49] ichasco. (2 de abril de 2018). *TIG: Graficar métricas de hosts y docker con Telegraf, InfluxDB y Grafana*.
Recuperado el 7 de junio de 2018 de
https://blog.ichasco.com/tig-graficar-metricas-de-hosts-y-docker-con-telegraf-influxdb-y-grafana/

[50] Labouardy, Mohammed. (8 de noviembre de 2017). *Exploring Swarm & Container Overview Dashboard in Grafana*.
Recuperado el 7 de junio de 2018 de
http://www.blog.labouardy.com/swarm-cluster-monitoring-grafana/

[51] Roberts, Mike. (22 de mayo de 2018) . *Serverless Architectures.* Recuperado el 7 de junio de 2018 de https://martinfowler.com/articles/serverless.html

[52] OpenFaaS. (s.f.). Deployment guide for Docker Swarm on ARM. Recuperado el 7 de junio de 2018 de https://github.com/openfaas/faas/blob/master/guide/deployment_swarm_arm.md

[53] Larsen, Christian. (s.f.). *jp2a.* Recuperado el 7 de junio de 2018 de https://csl.name/jp2a/

9 Anexos de la Memoria

ANEXO 1: Configuración de la Red en la Fase 1

Safe@Office 500W 8.0.39x

Check Point — We Secure the Internet

Welcome | Reports | Logs | Security | Antivirus | Antispam | Services | Network | Setup | Users | VPN | Help

SofaWare Embedded

Status | Traffic | My Computers | Connections | Networks | Tunnels | Routing

Active Computers — Refresh | Node Limit

Bridge

Safe@Office — 192.168.10.1

LAN (Bridged to: Bridge)

Safe@Office — 00:24:1d:99:fa:16

SmartHome1 — 192.168.10.10 b8:27:eb:55:13:84 — Edit

WLAN (Bridged to: Bridge)

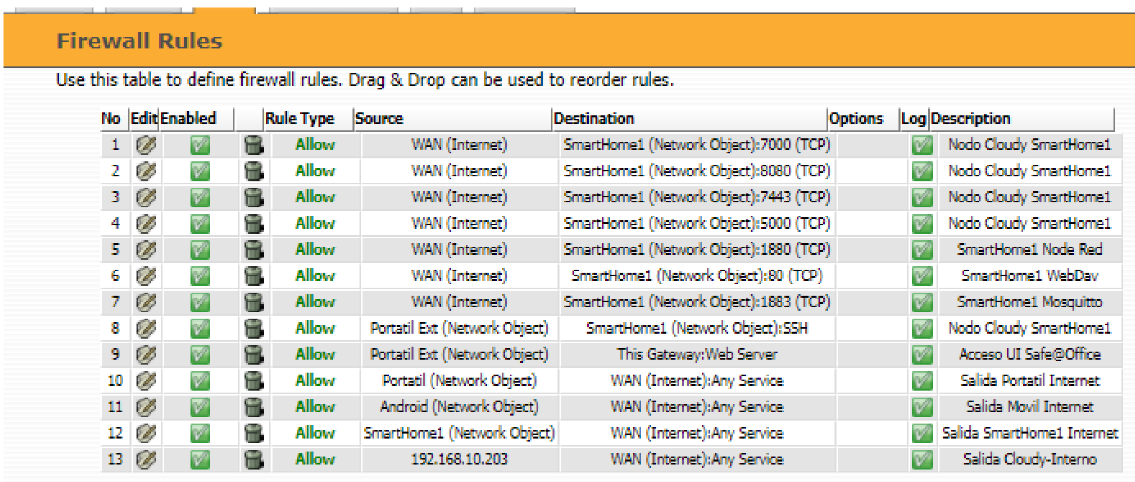
Safe@Office — 00:24:1d:99:fa:16

HomePlug1 — 192.168.10.100 (DHCP) ec:fa:bc:10:18:c1 — Signal: IIII (49dB) — Add

Inwall1 — 192.168.10.177 (DHCP) 68:c6:3a:8a:eb:45 — Signal: IIII (45dB) — Add

Power1 — 192.168.10.195 (DHCP) 2c:3a:e8:3b:3d:3c — Signal: IIII (60dB) — Add

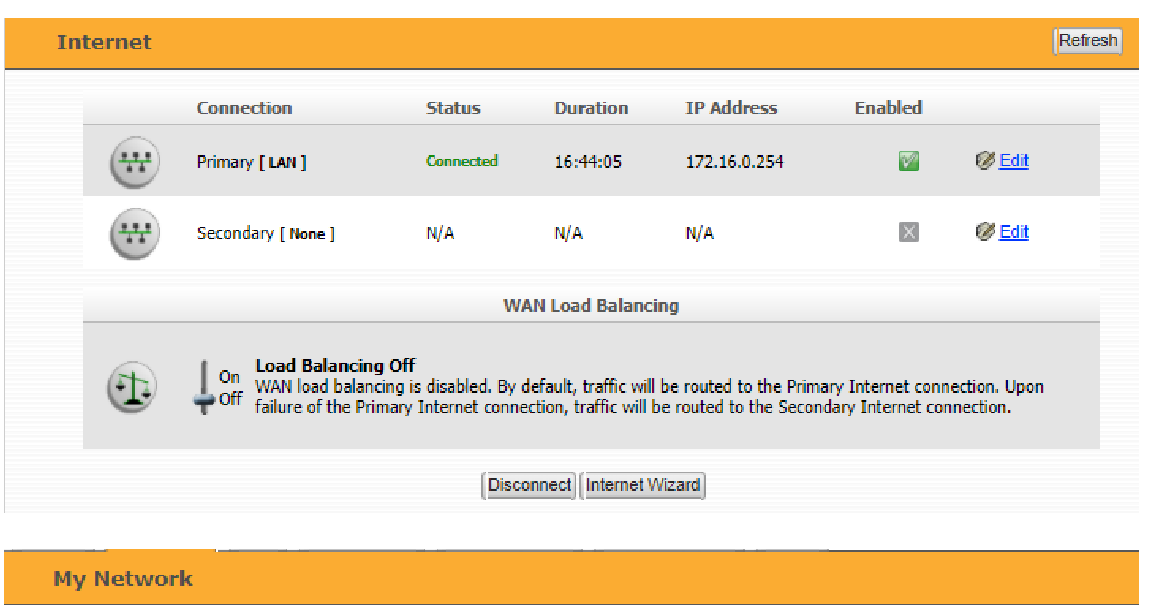
Android — 192.168.10.202 (DHCP) d0:9d:ab:96:f0:34 — Signal: IIII (45dB) — Edit



Firewall Rules

Use this table to define firewall rules. Drag & Drop can be used to reorder rules.

No	Edit	Enabled		Rule Type	Source	Destination	Options	Log	Description
1				Allow	WAN (Internet)	SmartHome1 (Network Object):7000 (TCP)			Nodo Cloudy SmartHome1
2				Allow	WAN (Internet)	SmartHome1 (Network Object):8080 (TCP)			Nodo Cloudy SmartHome1
3				Allow	WAN (Internet)	SmartHome1 (Network Object):7443 (TCP)			Nodo Cloudy SmartHome1
4				Allow	WAN (Internet)	SmartHome1 (Network Object):5000 (TCP)			Nodo Cloudy SmartHome1
5				Allow	WAN (Internet)	SmartHome1 (Network Object):1880 (TCP)			SmartHome1 Node Red
6				Allow	WAN (Internet)	SmartHome1 (Network Object):80 (TCP)			SmartHome1 WebDav
7				Allow	WAN (Internet)	SmartHome1 (Network Object):1883 (TCP)			SmartHome1 Mosquitto
8				Allow	Portatil Ext (Network Object)	SmartHome1 (Network Object):SSH			Nodo Cloudy SmartHome1
9				Allow	Portatil Ext (Network Object)	This Gateway:Web Server			Acceso UI Safe@Office
10				Allow	Portatil (Network Object)	WAN (Internet):Any Service			Salida Portatil Internet
11				Allow	Android (Network Object)	WAN (Internet):Any Service			Salida Movil Internet
12				Allow	SmartHome1 (Network Object)	WAN (Internet):Any Service			Salida SmartHome1 Internet
13				Allow	192.168.10.203	WAN (Internet):Any Service			Salida Cloudy-Interno



My Network

Network Name	Hide NAT	DHCP Server	IP Address	Subnet Mask		
Bridge			192.168.10.1	255.255.255.0	Erase	Edit
LAN	Enabled	Enabled				Edit
WLAN	Enabled	Enabled				Edit
DMZ [Disabled]						Edit
OfficeMode [Disabled]						Edit

Add Bridge

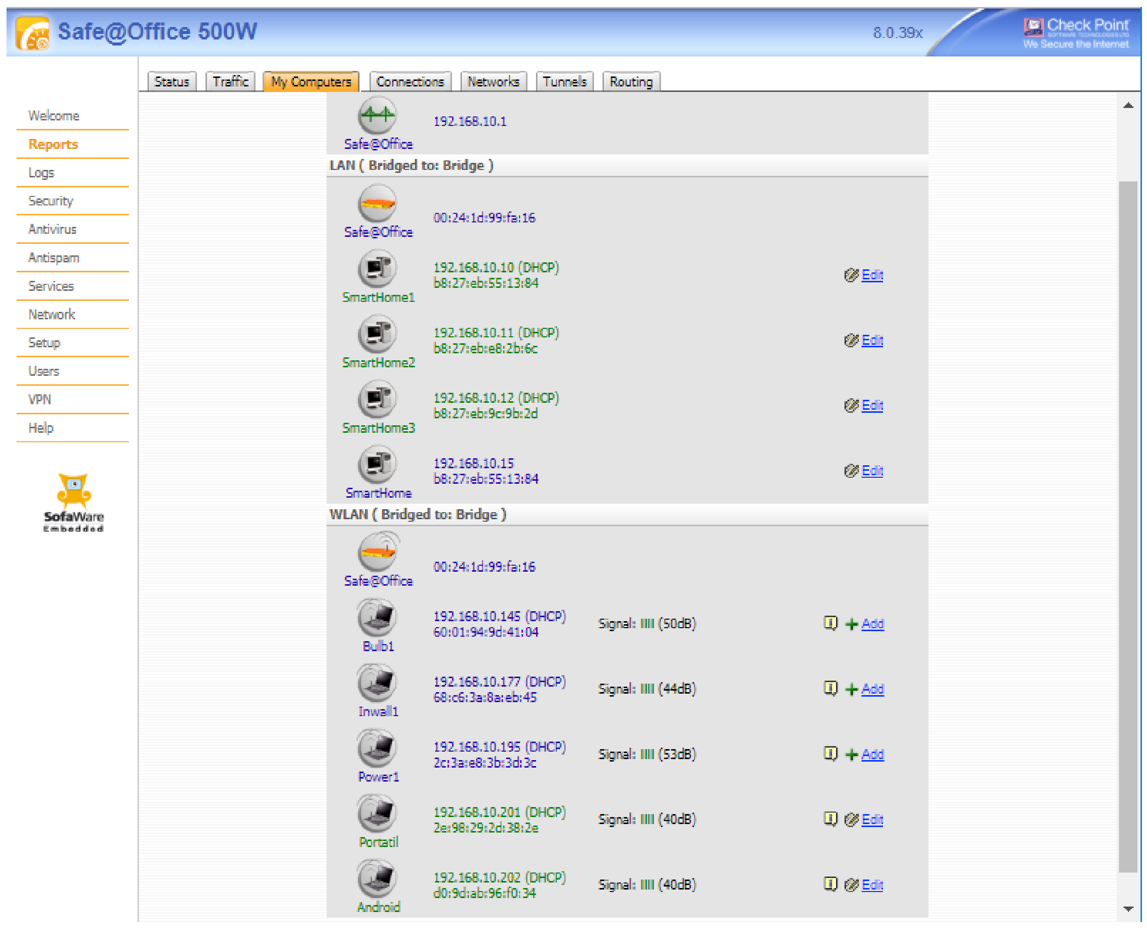
Ports

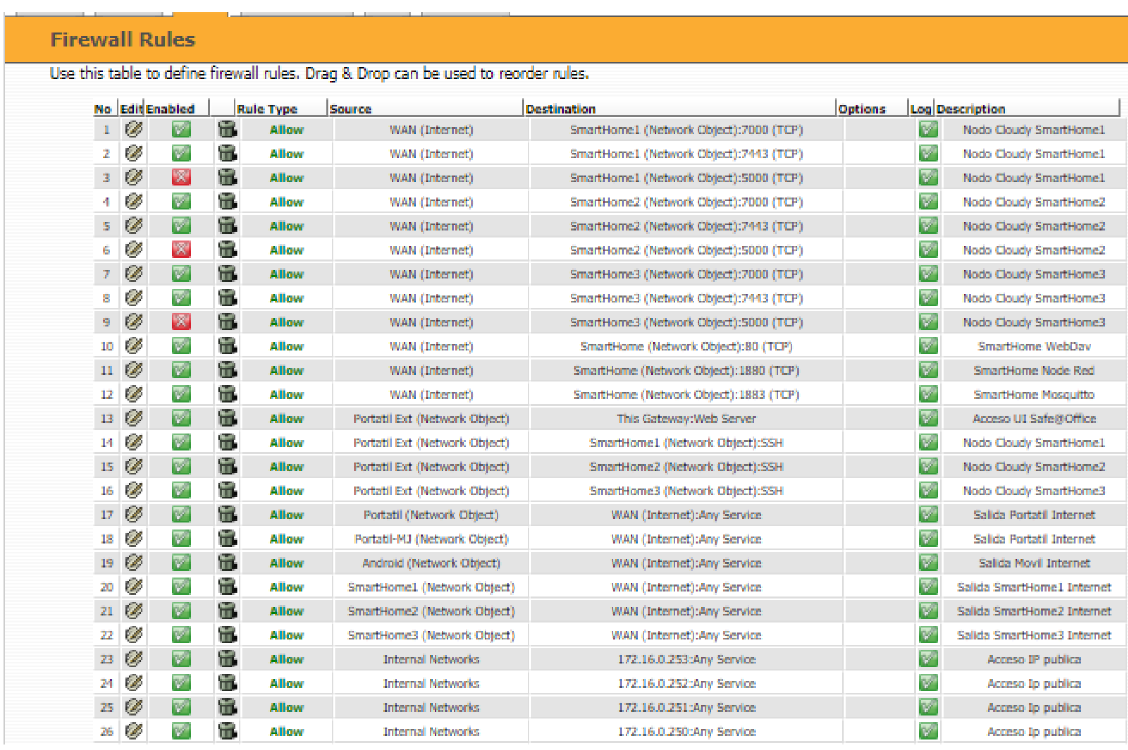
Refresh

Port	Assigned To	Status	
1	LAN	100 Mbps/Full Duplex	Edit
2	LAN	No Link	Edit
3	LAN	No Link	Edit
4	LAN	No Link	Edit
DMZ / WAN2	DMZ	Disabled	Edit
WAN	Internet	100 Mbps/Full Duplex	Edit
Serial	Console		Edit
USB	USB Devices	Not Connected	Edit

Default

ANEXO 2: Configuración de la Red en la Fase 2

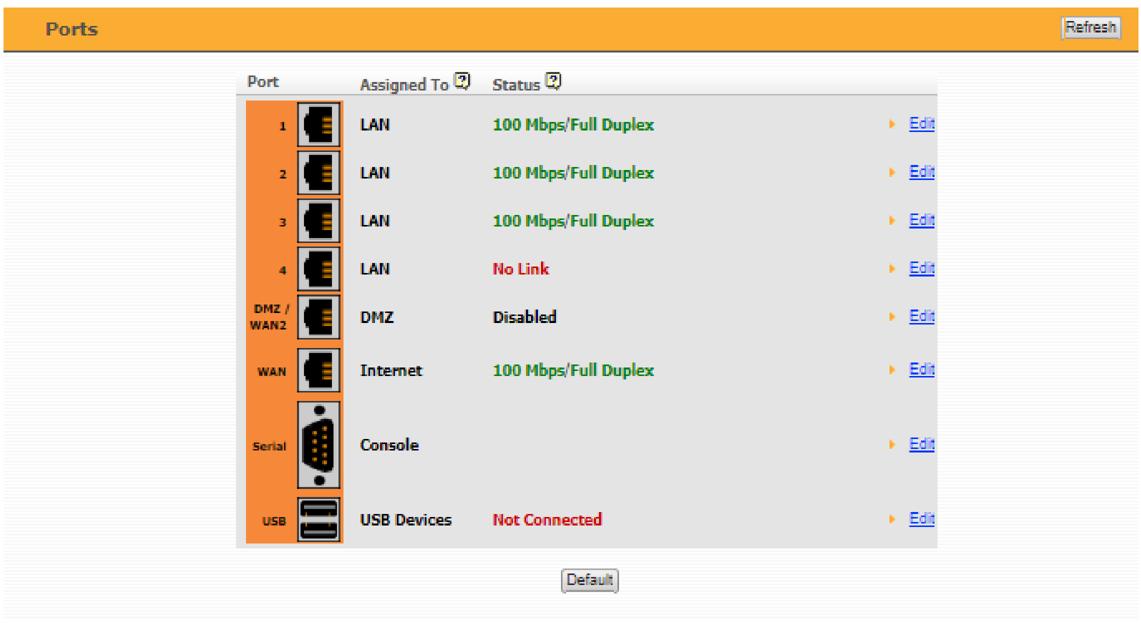




Firewall Rules

Use this table to define firewall rules. Drag & Drop can be used to reorder rules.

No	Edit	Enabled		Rule Type	Source	Destination	Options	Log	Description
1				Allow	WAN (Internet)	SmartHome1 (Network Object):7000 (TCP)			Nodo Cloudy SmartHome1
2				Allow	WAN (Internet)	SmartHome1 (Network Object):7443 (TCP)			Nodo Cloudy SmartHome1
3				Allow	WAN (Internet)	SmartHome1 (Network Object):5000 (TCP)			Nodo Cloudy SmartHome1
4				Allow	WAN (Internet)	SmartHome2 (Network Object):7000 (TCP)			Nodo Cloudy SmartHome2
5				Allow	WAN (Internet)	SmartHome2 (Network Object):7443 (TCP)			Nodo Cloudy SmartHome2
6				Allow	WAN (Internet)	SmartHome2 (Network Object):5000 (TCP)			Nodo Cloudy SmartHome2
7				Allow	WAN (Internet)	SmartHome3 (Network Object):7000 (TCP)			Nodo Cloudy SmartHome3
8				Allow	WAN (Internet)	SmartHome3 (Network Object):7443 (TCP)			Nodo Cloudy SmartHome3
9				Allow	WAN (Internet)	SmartHome3 (Network Object):5000 (TCP)			Nodo Cloudy SmartHome3
10				Allow	WAN (Internet)	SmartHome (Network Object):80 (TCP)			SmartHome WebDav
11				Allow	WAN (Internet)	SmartHome (Network Object):1880 (TCP)			SmartHome Node Red
12				Allow	WAN (Internet)	SmartHome (Network Object):1883 (TCP)			SmartHome Mosquitto
13				Allow	Portatil Ext (Network Object)	This Gateway:Web Server			Acceso UI Safe@Office
14				Allow	Portatil Ext (Network Object)	SmartHome1 (Network Object):SSH			Nodo Cloudy SmartHome1
15				Allow	Portatil Ext (Network Object)	SmartHome2 (Network Object):SSH			Nodo Cloudy SmartHome2
16				Allow	Portatil Ext (Network Object)	SmartHome3 (Network Object):SSH			Nodo Cloudy SmartHome3
17				Allow	Portatil (Network Object)	WAN (Internet):Any Service			Salida Portatil Internet
18				Allow	Portatil-MJ (Network Object)	WAN (Internet):Any Service			Salida Portatil Internet
19				Allow	Android (Network Object)	WAN (Internet):Any Service			Salida Movil Internet
20				Allow	SmartHome1 (Network Object)	WAN (Internet):Any Service			Salida SmartHome1 Internet
21				Allow	SmartHome2 (Network Object)	WAN (Internet):Any Service			Salida SmartHome2 Internet
22				Allow	SmartHome3 (Network Object)	WAN (Internet):Any Service			Salida SmartHome3 Internet
23				Allow	Internal Networks	172.16.0.253:Any Service			Acceso IP publica
24				Allow	Internal Networks	172.16.0.252:Any Service			Acceso Ip publica
25				Allow	Internal Networks	172.16.0.251:Any Service			Acceso Ip publica
26				Allow	Internal Networks	172.16.0.250:Any Service			Acceso Ip publica

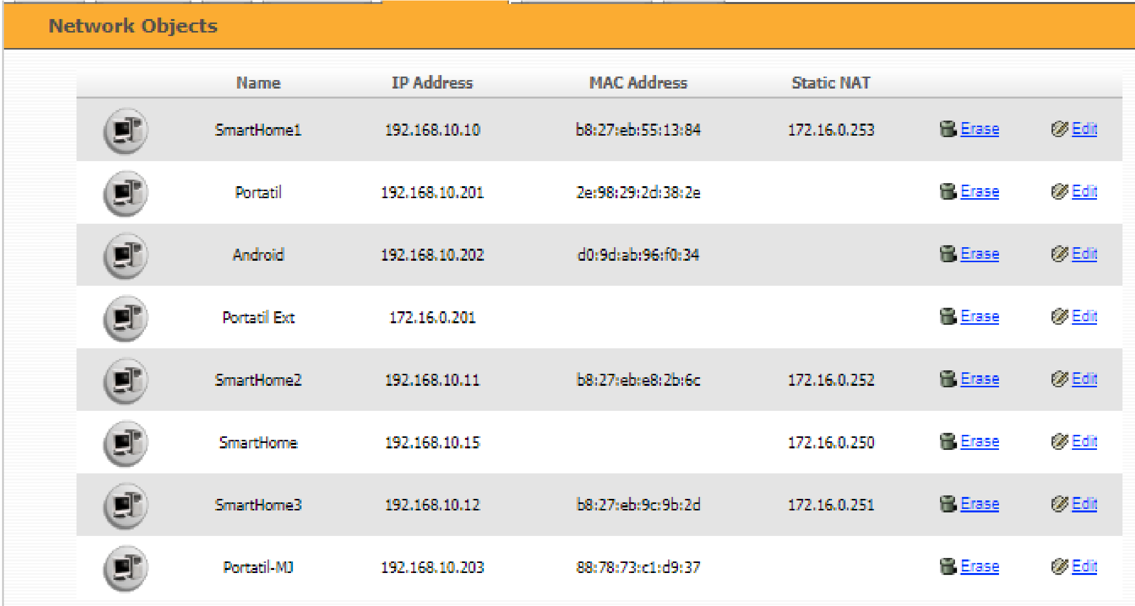


Ports

Refresh

Port	Assigned To	Status	
1	LAN	100 Mbps/Full Duplex	Edit
2	LAN	100 Mbps/Full Duplex	Edit
3	LAN	100 Mbps/Full Duplex	Edit
4	LAN	No Link	Edit
DMZ / WAN2	DMZ	Disabled	Edit
WAN	Internet	100 Mbps/Full Duplex	Edit
Serial	Console		Edit
USB	USB Devices	Not Connected	Edit

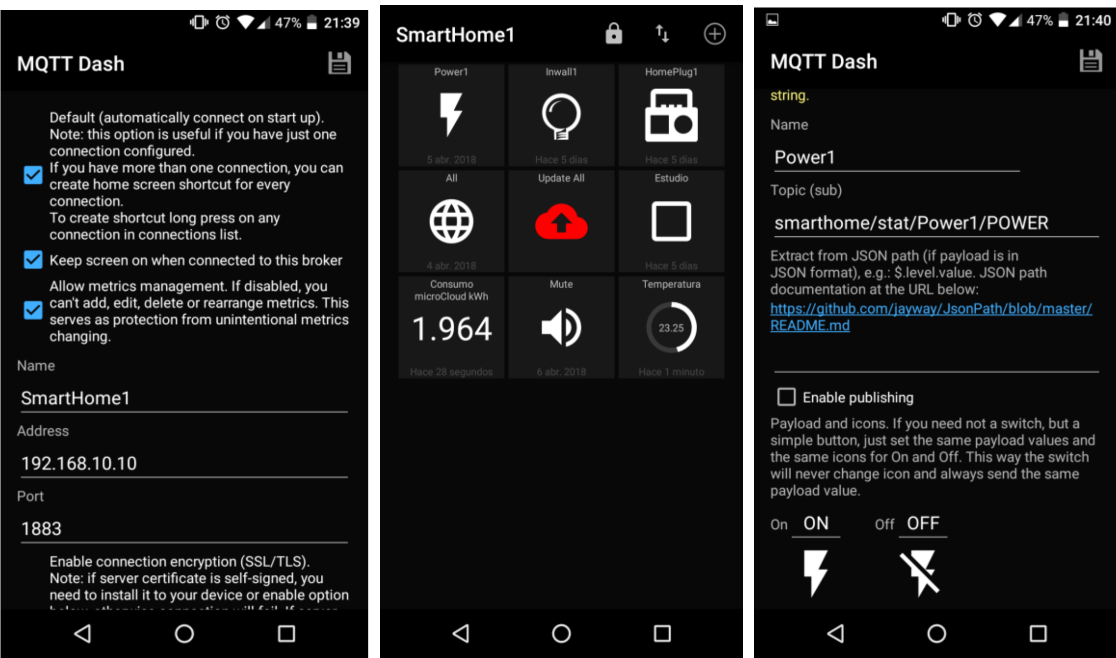
Default

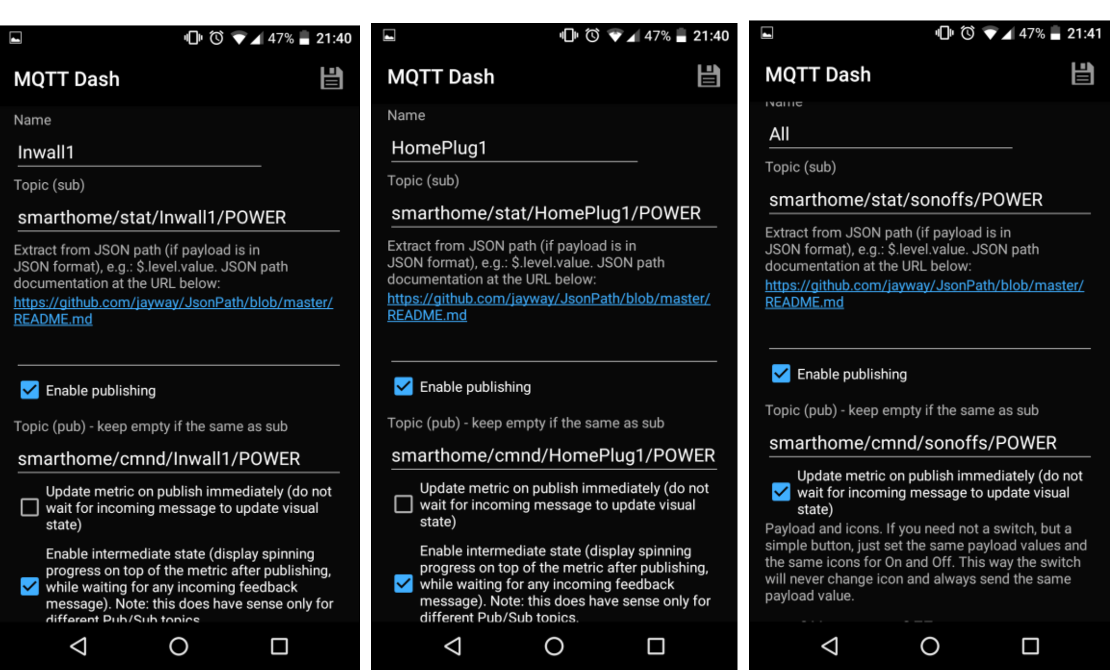


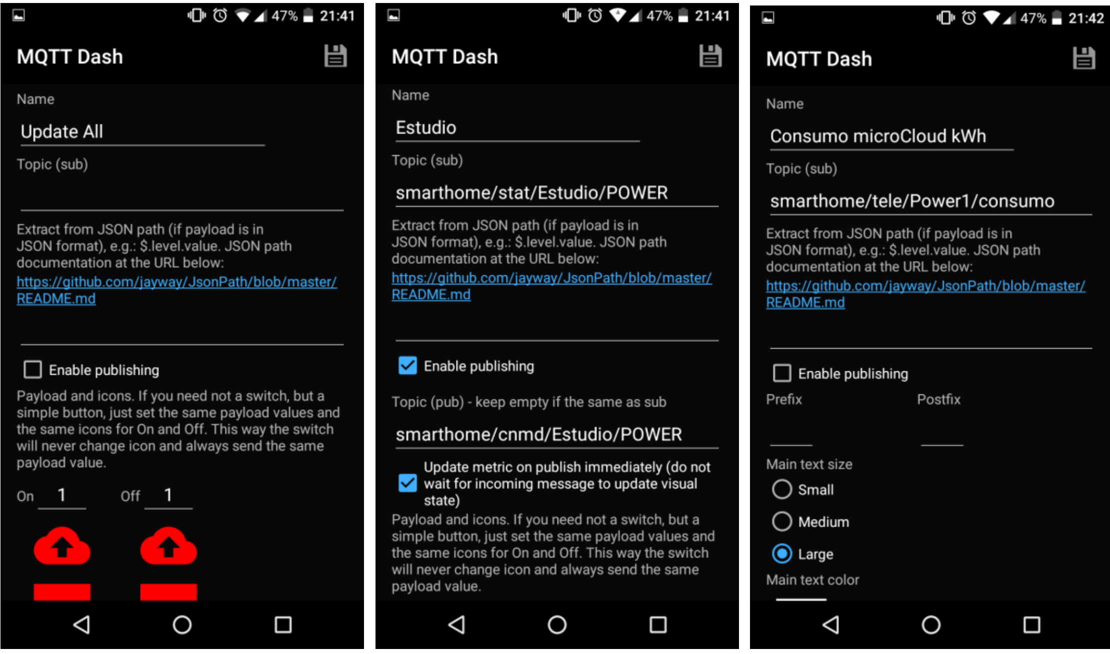
Network Objects

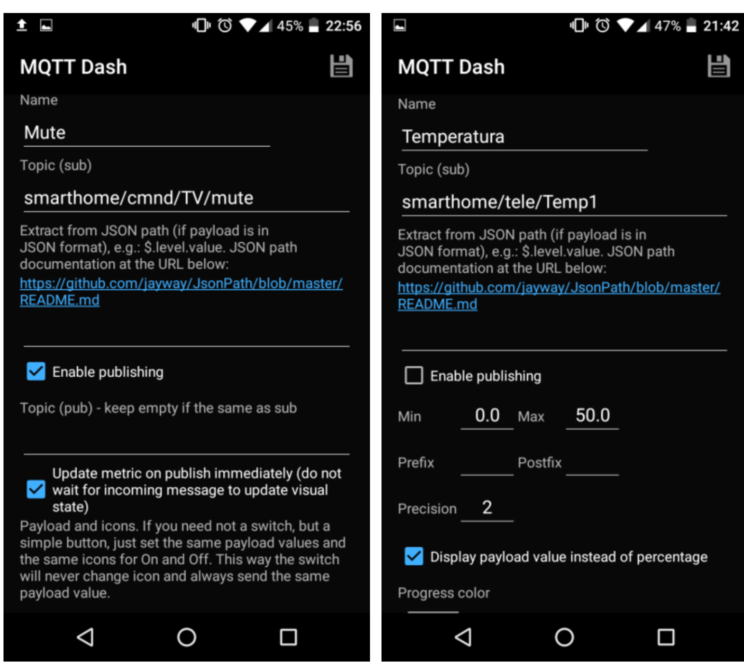
Name	IP Address	MAC Address	Static NAT		
SmartHome1	192.168.10.10	b8:27:eb:55:13:84	172.16.0.253	Erase	Edit
Portatil	192.168.10.201	2e:98:29:2d:38:2e		Erase	Edit
Android	192.168.10.202	d0:9d:ab:96:f0:34		Erase	Edit
Portatil Ext	172.16.0.201			Erase	Edit
SmartHome2	192.168.10.11	b8:27:eb:e8:2b:6c	172.16.0.252	Erase	Edit
SmartHome	192.168.10.15		172.16.0.250	Erase	Edit
SmartHome3	192.168.10.12	b8:27:eb:9c:9b:2d	172.16.0.251	Erase	Edit
Portatil-MJ	192.168.10.203	88:78:73:c1:d9:37		Erase	Edit

ANEXO 3: Configuración MQTT-Dash






47% 21:41
MQTT Dash
Name
Update All
Topic (sub)
Extract from JSON path (if payload is in JSON format), e.g.: $.level.value. JSON path documentation at the URL below:
https://github.com/jayway/JsonPath/blob/master/README.md
Enable publishing
Payload and icons. If you need not a switch, but a simple button, just set the same payload values and the same icons for On and Off. This way the switch will never change icon and always send the same payload value.
On 1
Off 1
47% 21:41
MQTT Dash
Name
Estudio
Topic (sub)
smarthome/stat/Estudio/POWER
Extract from JSON path (if payload is in JSON format), e.g.: $.level.value. JSON path documentation at the URL below:
https://github.com/jayway/JsonPath/blob/master/README.md
Enable publishing
Topic (pub) - keep empty if the same as sub
smarthome/cnmd/Estudio/POWER
Update metric on publish immediately (do not wait for incoming message to update visual state)
Payload and icons. If you need not a switch, but a simple button, just set the same payload values and the same icons for On and Off. This way the switch will never change icon and always send the same payload value.
47% 21:42
MQTT Dash
Name
Consumo microCloud kWh
Topic (sub)
smarthome/tele/Power1/consumo
Extract from JSON path (if payload is in JSON format), e.g.: $.level.value. JSON path documentation at the URL below:
https://github.com/jayway/JsonPath/blob/master/README.md
Enable publishing
Prefix
Postfix
Main text size
Small
Medium
Large
Main text color


45% 22:56
MQTT Dash
Name
Mute
Topic (sub)
smarthome/cmnd/TV/mute
Extract from JSON path (if payload is in JSON format), e.g.: $.level.value. JSON path documentation at the URL below:
https://github.com/jayway/JsonPath/blob/master/README.md
Enable publishing
Topic (pub) - keep empty if the same as sub
Update metric on publish immediately (do not wait for incoming message to update visual state)
Payload and icons. If you need not a switch, but a simple button, just set the same payload values and the same icons for On and Off. This way the switch will never change icon and always send the same payload value.
47% 21:42
MQTT Dash
Name
Temperatura
Topic (sub)
smarthome/tele/Temp1
Extract from JSON path (if payload is in JSON format), e.g.: $.level.value. JSON path documentation at the URL below:
https://github.com/jayway/JsonPath/blob/master/README.md
Enable publishing
Min 0.0
Max 50.0
Prefix
Postfix
Precision 2
Display payload value instead of percentage
Progress color

ANEXO 4: Problema de pérdida de paquetes

Durante la segunda fase del proyecto en el capítulo 3 de esta memoria, al extender la microCloud a nodos externos mediante TincVPN, detectamos problemas de pérdidas de paquetes de entre un 20% y un 40% que volvía nuestra microCloud inusable, provocando lentitud y perdida de conexión de la sesión SSH.

Tras investigar y revistar la configuración de la VPN a la situación inicial, comprobé que, en el caso de nuestro clúster de 3 nodos situados en la misma LAN, la perdida de paquetes se situaba entre el 10% y el 15%. En este caso no se llegaban a producir desconexiones, pero sí lentitud e inestabilidad en el clúster Swarm..

La inestabilidad del Swarm es provocada por la pérdida periódica de contacto entre los nodos. Esto genera contenedores parados que hay que ir limpiando cada cierto tiempo con los comandos 'docker system prune' y 'docker volume prune' para evitar agotar los recursos del clúster.

Al efectuar un ping continuo a los 3 nodos del clúster desde mi portátil en la misma LAN, comprobé que la perdida de paquetes se daba en algunas ocasiones simultáneamente en los 3 nodos.

Este problema además parece estar relacionado únicamente con la recepción de los paquetes de red, ya que un ping desde cualquiera de las Raspberry al gateway de red funciona correctamente sin ninguna pérdida de paquetes:

```
--- 192.168.10.1 ping statistics ---
102 packets transmitted, 102 received, 0% packet loss, time 101174ms
rtt min/avg/max/mdev = 5.534/5.780/8.502/0.461 ms
```

Pero un ping entre miembros del clúster muestra la pérdida de paquetes.

```
--- 192.168.10.11 ping statistics ---
112 packets transmitted, 101 received, 9% packet loss, time 115471ms
rtt min/avg/max/mdev = 0.495/0.663/2.669/0.215 ms
```

Además, esto ocurre en cualquiera de los 3 nodos por lo que podemos descartar problemas locales de cableado o de fuente de alimentación.

```
--- 192.168.10.12 ping statistics ---
138 packets transmitted, 122 received, 11% packet loss, time 142363ms
rtt min/avg/max/mdev = 0.472/0.652/0.836/0.078 ms
```

Al reiniciar cualquier nodo, el ping empieza respondiendo correctamente, y al cabo de algunos minutos empieza la perdida de paquetes. Al efectuar un shutdown se observa que durante la última parte del proceso antes del apagado no se produce perdida de paquetes. Esto nos da la pista de que el problema puede ser causado por un servicio de Cloudy.

Existe información en Internet que relaciona este problema con Avahi y una mala implementación del driver de red en Raspbian para multicast **[47]**, pero incluso al desactivar Avahi no se reduce la pérdida de paquetes. Al analizar la red con el comando ‘tcpdump’, comprobamos que la perdida ocurre tras el envío de paquetes UDP en el puerto 5000 que corresponde a Serf.

Concluyo que Serf está provocando la perdida de paquetes al interactuar los nodos entre sí en los anuncios en la LAN. Cuantos más nodos hay en una misma LAN, más perdida de paquetes de red tenemos. Por esa razón no hemos detectado el problema en la primera fase, y se ha agravado la perdida de paquetes al añadir el cuarto nodo Swarm con la VPN.

Tras deshabilitar Serf y Avahi en los 3 nodos ‘manager’, comprobamos que la pérdida de paquetes desaparece completamente. Debido al estado avanzado del proyecto y a la falta de tiempo para investigar más a fondo el problema, nos limitaremos a mantener desactivados Serf y Avahi para la última fase.

ANEXO 5: Descripción de ficheros adjuntos a la Memoria

- Carpeta: .
  Descripción: Documentos del proyecto
    - jraelgutierrez_memoria.pdf: memoria del proyecto
    - jraelgutierrez_presentacion.pdf: presentación del proyecto
    - jraelgutierrez_Informe_Evaluacion_TFG.pdf: Informe de autoevaluación del proyecto

- Carpeta: ./Dashboards
  Descripción: Configuraciones de Dashboards de Grafana
    - microCloud.json: cuadro de mando de la microCloud
    - OpenFaaS.json: cuadro de mando para OpenFaaS
    - Swarm.json: cuadro de mando de Swarm

- Carpeta: ./Docker-compose
  Descripción: Cada carpeta contiene un fichero configuración de servicios Swarm 'docker-compose.yml'.
    - openfaas_service: Servicios Swarm para OpenFaaS
    - SmartHome_service_public: microCloud IoT
    - Synthing_service: replicación de datos para Docker
    - Tig_service: Servicios para monitorización con TIG

- Carpeta: ./functions
  Descripción: Cada carpeta contiene un fichero DockerFile para crear imágenes de funciones para OpenFaaS.
    - Curl: una función que encapsula el comando 'curl'
    - jp2a: una función que transforma fichero JPG en texto ASCII
    - ResizeImageMagic: una función que reduce una imagen al 50%

- Carpeta: ./Node-Red
  Descripción: Configuración de flujos de Node-Red
    - All_flows.txt: todos los flujos de Node-Red usados en la memoria.

- Carpeta: ./Plan
  Descripción: EDT y planificación del proyecto con ms-project
    - edt_completa.xlsx: EDT con el detalle de las tareas del proyecto, incluyendo estimación de tiempo
    - edt_simplificada: EDT simplificada, incluyendo estimación de tiempo.
    - tfg_gantt.mpp: fichero Project con las tareas del proyecto definidas en horas.

- Carpeta: ./Tasmota
  Descripción: Ficheros de configuración del firmware Tasmota
  - platformio.ini: Fichero de configuración de Platform.io
  - user_config.h: Fichero de configuración de Tasmota

- Carpeta: ./scripts
  Descripción: Scripts desarrollados o modificados a lo largo del proyecto
  - avahi-ps: script modificado para publicación en IP alternativa
  - avahi-ps-serf: plug modificado de avahi-ps con serf
  - chkmqtt.sh: script para testear mqtt+node-red desde keepalived
  - regpublic.py: script para registrar servicios docker en Serf al reiniciar el nodo.

ANEXO 6: Enlaces de Video

- Video hablado de la presentación de la memoria: https://www.youtube.com/watch?v=TSrqNT5MEdY

- Video demo de la microCloud: https://www.youtube.com/watch?v=qtmvYINjx18