
# Aplicar Kubernetes sobre un entorno cloud comunitario y descentralizado

**Ismael Fernández Molina**

Grado de Ingeniería Informática

**Félix Freitag**

Data Entrega

**FICHA DEL TRABAJO FINAL**

| | |
|---|---|
| **Título del trabajo:** | Aplicar Kubernetes sobre un entorno cloud comunitario y descentralizado |
| **Nombre del autor:** | Ismael Fernández Molina |
| **Nombre del consultor:** | Félix Freitag |
| **Fecha de entrega (mm/aaaa):** | MM/AAAA |
| **Área del Trabajo Final:** | Sistemas Distribuidos |
| **Titulación:** | Grado de Ingeniería Informática |

**Resumen del Trabajo (máximo 250 palabras):**


Hace aproximadamente 5 años los contenedores y concretamente docker revolucionaron el despliegue de aplicaciones dando velocidad e inmutabilidad en el desarrollo de éstas.

Dado el rápido crecimiento en producción de está tecnología se creó la necesidad de orquestar contenedores, estos dada una especificación controlan/mantienen el estado de las aplicaciones. Además, haciendo uso de los cgroups podemos mantener aisladas más de una aplicación en una misma instancia.

Dentro de los muchos orquestadores de contenedores (docker swarm, Nomad, etc) encontramos Kubernetes, una de las tecnologías más extendidas hoy en día en el mundo de las aplicaciones web.

Nuestro objetivo es crear una red local de nodos Cloudy y Kubernetes. Además, queremos exponer a través de serf toda aplicación desplegada con Kubernetes y permitir el acceso desde la interfaz de Cloudy.

**Abstract (in English, 250 words or less):**

5 years ago, containers and specifically the platform named docker revolutionized how the apps will be deployed, giving velocity and immutability in them development.

Given the rapid growth in production of this technology the need to orchestrate containers was created. Given a specification or manifest file this orchestrators control/maintain the status of applications. In addition, by using cgroups we can keep more than one application isolated in the same instance.

Among the many container orchestrators (Docker Swarm, Nomad, etc) we find Kubernetes, one of the most widespread technologies today in the world of web applications.

Our goal is to build a local network of nodes with Cloudy and Kubernetes installed. Moreover, we would like to expose our services deployed with Kubernetes through Serf and allow access from the Cloudy's interface.

**Palabras clave (entre 4 y 8):**

kubernetes, Cloudy, edge, contenedores, operator

# Índice

**Lista de figuras**

# 1. Introducción del TFG

En este trabajo diseñaremos e implementaremos el despliegue de nodos distribuidos mediante Kubernetes, ofreciendo servicios en alta disponibilidad y tolerantes a fallos.

Todo los dispositivos que usaremos serán de bajo coste y basados en la distribución Cloudy.

Usaremos siempre proyectos Open Source tanto para Hardware como para Software.

## 1.1 Objetivos Generales y específicos

Los objetivos generales de este proyectos son los siguientes:

- Poder aplicar los conocimientos adquiridos durante la ingeniería para la resolución de problemas de forma eficiente creando y usando componentes informáticos.
- Poder usar y contribuir a proyectos FLOSS [1].
- Entender la aplicabilidad y las limitaciones de Kubernetes en entornos locales, distribuidos y heterogéneos.

Los objetivos específicos son los siguientes:

- Desplegar un clúster de Kubernetes en un entorno local en la distribución Cloudy de GNU/Linux.
- Conocer los distintos componentes de software Open Source necesarios para el despliegue de entornos distribuidos y en concreto Guifi.
- Desarrollar nuevos componentes de Kubernetes para dar soporte al despliegue de Guifi.

## 1.2 Descripción del proyecto

Realizaremos el proyecto en 4 fases, tratando de trabajar lo más ágil posible, iremos entregando cada 2 semanas “producto” evitando entregar el sistema con la última PAC.

Las fases son las siguientes:

1. Investigación y diseño.
2. Implementación en local.
3. Implementación de forma distribuida.
4. Investigación y extensión de funcionalidades.

A continuación expongo cuales son los hitos para cada fase.

### 1.2.1 Investigación y diseño

Está primera fase la dividiremos en 2:

- Preparación del entorno, instalando Cloudy en las 3 raspberries y en el ordenador.
- Aprenderemos como expone servicios Cloudy mediante Avahi y Serf.
- Aprendizaje de Kubernetes.

Objetivo: Tener el entorno listo y compartiendo servicios localmente.
Saber y reconocer todos los componentes de Kubernetes.

### 1.2.2 Implementación de un clúster en local

Montaremos un clúster de Kubernetes en local, el portátil hará de nodo máster, teniendo las 3 Raspberries cómo workers. De esta forma podremos tener alta disponibilidad de nuestras aplicaciones.

- Desplegamos Kubernetes y lo automatizamos con Ansible.
- Obtendremos métricas para ver el estado del clúster y nuestros servicios.

Objetivo: Haremos las pruebas necesarias para ver cómo de “resilent” es el clúster.

### 1.2.3 Implementación de forma distribuida

Estudiaremos si es posible y cómo exportar  kubernetes a nodos distribuidos GUIFI.

Objetivo: Kubernetes en GUIFI.

### 1.2.4 Investigación y nuevas funcionalidades del clúster

Mediante la implementación de controllers se pueden añadir funcionalidades al clúster de Kubernetes.

Objetivo: Ver si es posible y si tiene sentido implementar nuevos controllers en Kubernetes para usarlos desde GUIFI.

## 1.3 Recursos disponibles

En este apartado se indican los recursos disponibles tanto humanos como dispositivos para el proyecto.

### 1.3.1 Recursos humanos

El principal recurso humano de este trabajo, este mismo autor, está desarrollando un trabajo a tiempo completo, por lo que puede dedicar de forma rutinaria al proyecto 2 horas al día en días entre semana y de manera general las tardes del fin de semana, lo que supone unas 20 horas semanales.

La asignatura de TFG consta de 12 créditos que a razón de 25 horas por crédito suponen dedicar unas 300 horas a lo largo del cuatrimestre.

Todo lo anterior nos da unas 15 semanas, para completar el proyecto, lo que cuadra perfectamente con la entrega final del proyecto fijada para el día 09/06/2018.
No obstante lo indicado, se puede efectuar un aumento de las horas semanales para recuperar retrasos en el proyecto.

### 1.3.2 Recursos Físicos

- 3 Raspberry Pi modelo B+ las cuales harán de nodos localmente.
- 1 Raspberry pi 0 cómo VPN.
- MacBook Pro para el desarrollo de aplicaciones.
- Repositorio github.

### 1.4 Planificación del trabajo

Describimos la planificación del trabajo siguiendo las pautas fijadas en la UOC.

#### 1.4.1. Fecha de entregas

Siguiendo el calendario de la UOC, realizaré 4 entregas:

1. PAC1 – 04/03/2019
2. PAC2 – 14/04/2019
3. PAC3 – 19/05/2019
4. Entrega Final – 09/06/2019

#### 1.4.2 Entregas

- En la primera PAC se entregará el documento con el plan de trabajo, reflejando los hitos en cada punto de las fases.
- En la PAC número dos entregaremos los pasos documentados de cómo instalar en local todo lo necesario para correr Cloudy.
- Documentación sobre Kubernetes y Minikube. El estado del arte, momento actual de lo sistemas distribuidos y el porqué de la existencia de Kubernetes.
- En la tercera y previa entrega antes de finalizar el trabajo, entregaremos todas las pruebas realizadas en el clúster, cómo desplegar el clúster en k8s, y sí puede hacerlo o no en los nodos GUIFI.
- Entrega final.

### 1.4.3 Cronograma

Dada las entregas, he creado un diagrama de Gantt que me permita organizar en el tiempo las tareas  a realizar.

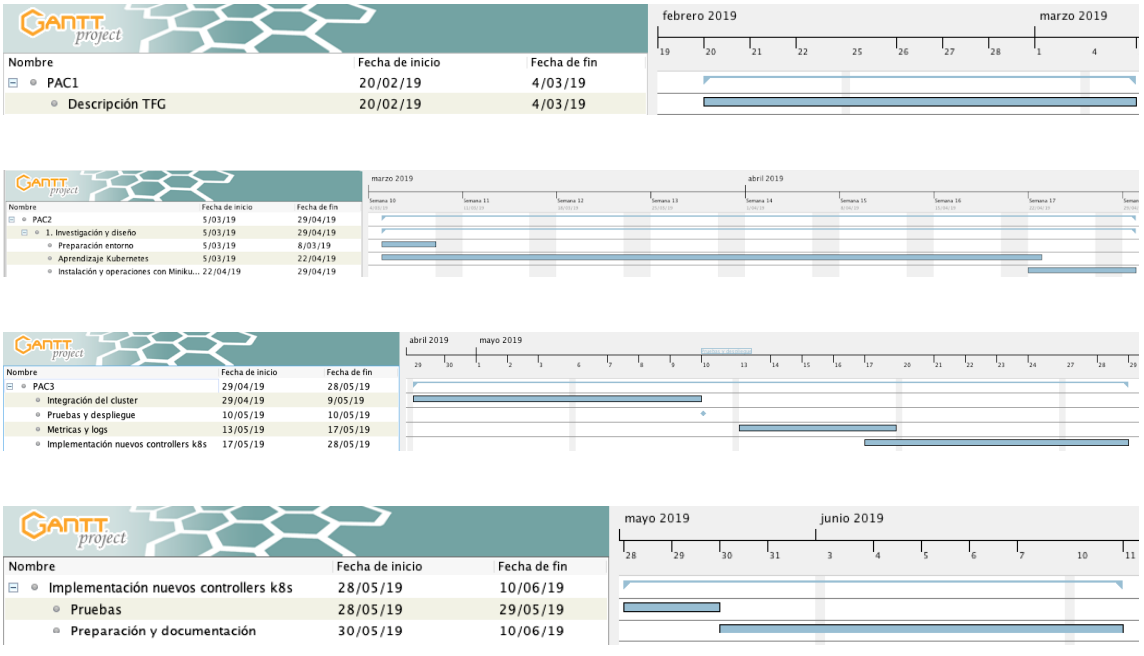
Figura 1: Estructura de desglose de tareas

| Nombre | Fecha de inicio | Fecha de fin |
|---|---|---|
| PAC1 | 20/02/19 | 4/03/19 |
| Descripción TFG | 20/02/19 | 4/03/19 |
| PAC2 | 5/03/19 | 29/04/19 |
| 1. Investigación y diseño | 5/03/19 | 29/04/19 |
| Preparación entorno | 5/03/19 | 8/03/19 |
| Aprendizaje Kubernetes | 5/03/19 | 22/04/19 |
| Instalación y operaciones con Minikube | 22/04/19 | 29/04/19 |
| PAC3 | 29/04/19 | 28/05/19 |
| Integración del cluster | 29/04/19 | 9/05/19 |
| Pruebas y despliegue | 10/05/19 | 10/05/19 |
| Metricas y logs | 13/05/19 | 17/05/19 |
| Implementación nuevos controllers k8s | 17/05/19 | 28/05/19 |
| Implementación nuevos controllers k8s | 28/05/19 | 10/06/19 |
| Pruebas | 28/05/19 | 29/05/19 |
| Preparación y documentación | 30/05/19 | 10/06/19 |

### 1.4.4 Diagrama de Gantt

He dividido el Diagrama de Gantt en las 4 entregas para poder verlo claramente.

Figura 2: Diagrama de Gantt



## 1.5 Breve sumario de productos obtenidos

Tal como hemos comentado en los apartados anteriores, y siguiendo los tiempos marcados, en la primera fase hemos realizado un informe sobre Cloudy y Kubernetes, hemos aprendido sobre la arquitectura de Kubernetes y realizado pruebas con Minikube.
Los informes y las pruebas se pueden encontrar anexadas a este documento.

En la fase de implementación hemos obtenido diferentes escenarios donde puede encajar Cloudy y Kubernetes.
Hemos desarrollado un pequeño operator que se puede encontrar en el repositorio de GitHub:

https://github.com/ismferd/serf-publisher

Y creado unos pequeños playbooks para facilitar el despliegue en nuestro clúster de forma fácil y automatizada:

https://github.com/ismferd/tfg-things
https://github.com/Clommunity/Cloudynitzar/pull/17

Finalmente, conseguimos el objetivo de exponer servicios en Cloudy mediante Serf desplegándose en Kubernetes.

Adjuntamos los enlaces a los vídeos de presentación y demostración.

Google drive: video presentación
Google drive: video demostración

## 1.6 Breve descripción de los otros capítulos de la memoria

El documento se acaba organizando en N capítulos donde se encuentra la siguiente información:

Investigación: Comprende los capítulos 2 y 3, donde explicamos las plataformas con las que vamos a trabajar.1

Diseño: En el capítulo 4 hablamos sobre los posibles escenarios y cómo encajan ambas plataformas.

Por último, en el capítulo 5 vemos la implementación y producto “final”. Hacemos una prueba de como exponemos servicios mediante las 2 plataformas juntas.

# 2. Primera fase: Investigación y diseño

En esta primera fase veremos:

- Qué es el Cloud Computing gestionado.
- Qué es Cloudy, para que sirve y cómo puede exponer servicios de forma local.
- Estudiaremos la arquitectura básica de Kubernetes.

## 2.1 Cloud Computing, Cloud Community Network y Containers

Los sistemas distribuidos en la actualidad están al alza, ya que nos permiten tener nuestras aplicaciones asegurando la alta disponibilidad y la tolerancia a fallos (siempre hablo de aplicaciones *stateless* en cualquier otro caso deberíamos de poner en la balanza la consistencia).

### 2.1.1 Cloud Computing

Es el nuevo modelo de plataforma y computación, actualmente mediante llamadas HTTP creamos nuestra infraestructura (PaaS), cuando antes se alquilaban *racks* e instalábamos nuestros servidores en *Data Centers.*

Empresas como AWS, Google o Digital Ocean han revolucionado el mercado de los sistemas distribuidos y contribuido a que todo el mundo pueda disponer de sus aplicaciones en disponibilidad o poder escalar las aplicaciones bajo demanda o siguiendo unas reglas de escalado (ejemplo: Cuando la CPU esté por encima del 70% añade una instancia más a pool de máquinas).

Éstas compañías han ido creciendo de forma exponencial, ya no sólo ofrecen  servicios de computación en la nube, si no que te permiten almacenar datos de forma escalable y segura. Por ejemplo en el caso de AWS tenemos el servicio de s3 (Amazon S3) es un servicio de almacenamiento de objetos que ofrece escalabilidad, disponibilidad de datos, seguridad,  rendimiento y ofrecer una durabilidad del 99,999999999 % [2].

Dada ésta fiabilidad y elasticidad que nos proporciona el cloud computing las empresas están haciendo migraciones y transformaciones de sus infraestructuras para usar proveedores Cloud.

Figura 3: Servicios de AWS


### 2.1.2 *Cloud Community Network*

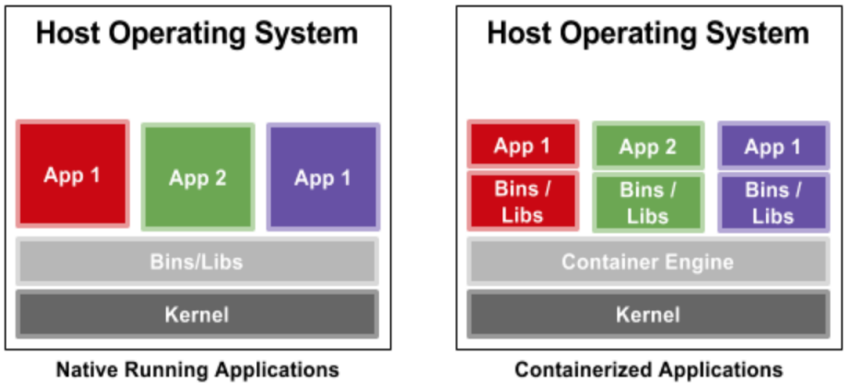
Antes de la adopción general de servicios en la nube (cloud services) los usuarios de redes comunitarias ya compartían o proveían servicios y recursos a la comunidad; sin embargo, estos usuarios eran una minoría. Una de las principales razones es la barrera tecnológica. Antes de proveer contenidos, los usuarios que quieren compartir información con la comunidad tienen primero que preocuparse de los aspectos técnicos como el despliegue/montaje de un servidor con un conjunto de servicios [3].
Soluciones cómo Cloudy facilitan que los usuarios puedan proveer servicios a la comunidad.

### 2.1.3 Containers

Desde aproximadamente hace 2 décadas se ha estado usando la contenerización de aplicaciones, esto quiere decir nuestra aplicación va a tener aislamiento de recursos, es decir, podemos definir límites de red, cpu y memoria para cada uno de nuestros procesos.

Dada esta tecnología y su crecimiento (actualmente hay proyectos como Docker, rkt, c-rio, …) podemos tener nuestra aplicación corriendo con un solo comando sin tener que preocuparnos de las dependencias, ocurriendo que podemos desplegar nuestra aplicación en cualquier entorno que soporte el motor de container seleccionado.

Figura 4: Arquitectura de aplicaciones conteinerizada y nativa




Dada la capacidad de aislamiento podemos tener más de un container corriendo en  nuestras instancias, pero cómo podemos hacer si tenemos varias aplicaciones web las cuales exponen todas en el puerto 443, para resolver este problema surgieron los orquestadores de containers.

### 2.1.4 Orquestadores de containers

Como comentamos en el apartado anterior este tipo de tecnología resuelve la exposición de varios containers en la misma instancia exponiendo el mismo puerto. Actualmente estos orquestadores hacen mucho más que orquestar, crean *health checks*, autoscaling, mantienen el estado deseado, se ocupan de buscar el mejor worker donde alojar nuestra aplicación, es decir, si tenemos una aplicación que consumirá 1vcpu y un 1GB de memoria RAM, nuestro orquestador se ocupará de buscar un hueco en el pool de nodos.

Normalmente este tipo de tecnología se basa en la arquitectura master-worker.
En los másters se guarda el estado deseado, el estado en el que debe de estar nuestro clúster, y en los nodos workers se haya la computación, es el lugar donde se alojan las aplicaciones.

Figura 5: Arquitectura de Kubernetes

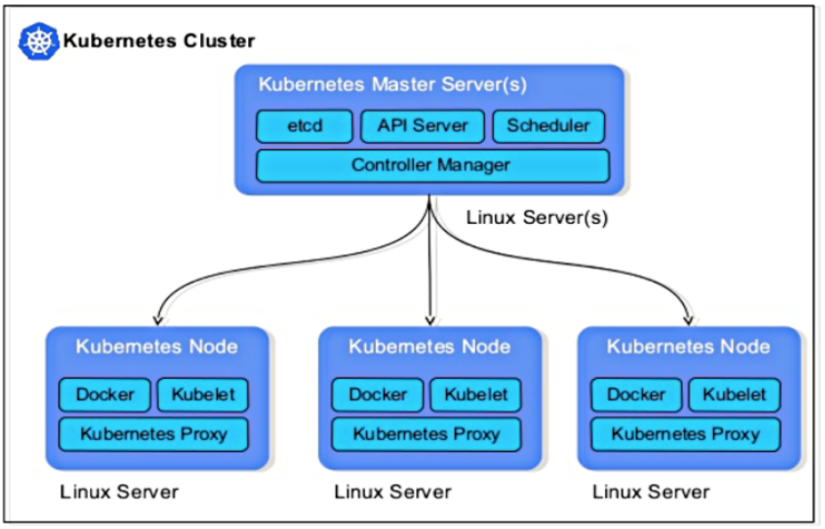
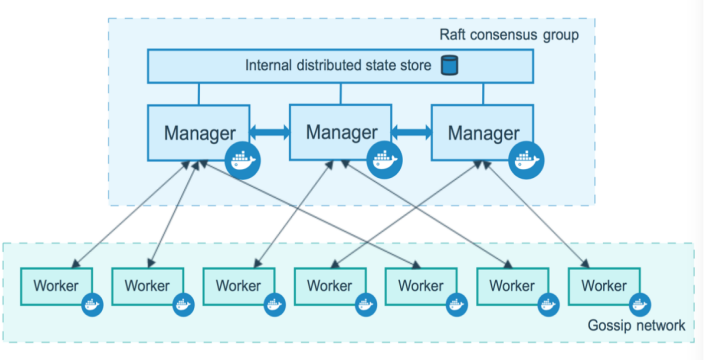



Figura 6: Arquitectura de Docker Swarm



## 2.2. Cloudy

### 2.2.1 Que es Cloudy

Cloudy es una distribución de GNU/Linux con licencia GPLv2 [4], está distribución está pensada para exponer y descubrir servicios en un red comunitaria (*Community Network Cloud*), estos servicios son expuestos a través de Serf [5].

### 2.2.2 Análisis de Cloudy

Probaremos Cloudy en un entorno de test el cual incluye:

- 3 raspberries con raspbian instalado.
- Portátil con ubuntu instalado.

En el primer contacto con Cloudy vemos que es realmente fácil de instalar de forma local y crear rápidamente una microCloud en el entorno de testing.

Si damos un vistazo al backoffice, expuesto por defecto en el puerto 7000, observamos la capacidad de poder instalar y compartir una serie de servicios que automáticamente se expondrán a través de Serf. También podemos hacerlo desde un terminal y de hecho aquí es donde se puede sacar toda la potencia de Cloudy, ya que podemos exponer cualquier servicio.

Como hemos comentado Cloudy expone los servicios a través de Serf. Serf mediante el protocolo Gossip [6] mantiene el clúster actualizado, es decir, cada vez que hay una modificación en alguno de los nodos pertenecientes al clúster de Serf, será éste quien mediante tags propague el cambio al resto de nodos.

### 2.2.3 Conclusiones

Después instalar Cloudy en nuestro entorno local, observamos que:

- Cloudy es muy fácil de instalar y empezar a usar.
- Exponemos de forma rápida los servicios que nos ofrece Cloudy desde su *backoffice*.

- Dado el orígen de Serf (Distribuido, descentralizado, tolerante a fallos y de alta disponibilidad), podemos tener consistencia eventual en nuestro nodo.
- Dificultad si queremos exponer servicios que no están en la web de administración, debemos de indagar cómo *Avahi* habla con Serf, y lanzar el comando desde la *command line.*

En el **Anexo I: Primeros pasos con Cloudy** se añade toda la información necesaria sobre las pruebas realizadas en el entorno de test:

- Instalación de Cloudy en varios nodos de nuestra red local.
- Exposición de servicios a través del backoffice.
- Exposición de servicios desde el terminal.
- Cómo trabaja Serf por debajo de Cloudy.

## 2.3 Kubernetes

### 2.3.1 Qué es Kubernetes

Kubernetes es un proyecto open source el cual nos permite automatizar, escalar y orquestar el despliegue de contenedores.

### 2.3.2 Análisis de Kubernetes

Kubernetes [7] nos provee con herramientas para automatizar la distribución de aplicaciones a un clúster de servidores, asegurando una utilización más eficiente del hardware, comparada con la que podemos conseguir de manera tradicional.

Kubernetes se coloca delante de un conjunto de servidores para que nosotros los veamos como un solo servidor, aunque detrás puede haber decenas o miles de servidores.

Para conseguir esto, Kubernetes expone una API Restful que podemos utilizar para desplegar de forma sencilla nuestra aplicación, y que hará que el propio sistema se ocupe de arreglar fallos en servidores, de la monitorización de la aplicación, etc. En vez de desplegar nuestra aplicación a todos los servidores, solo tendremos que decidir cuantas replicas de la aplicación deben ejecutarse al mismo tiempo, y Kubernetes se encargará de que ese estado se cumpla [8].

La API de Kubernetes nos permite modelar todo el ciclo de vida de las aplicaciones que se ejecutarán en el clúster de servidores. Es una API

Restful a la que podemos enviar peticiones describiendo cómo queremos desplegar. Podemos utilizar cualquier cliente HTTP para enviarle peticiones (curl, wget, Postman...), pero el propio Kubernetes trae una herramienta de línea de comandos para facilitar la interacción con la API. Se llama kubectl [9].

### 2.3.3 Conclusiones

En mi opinión Kubernetes no es sólo un orquestador de contenedores, es una plataforma la cual nos abstrae completamente de nuestro provider ya sea cloud u on-premise, nos permite definir el estado que queremos de nuestras aplicaciones y dados unos bucles de conciliación va intentar aplicar siempre que sea posible.

Y no sólo eso, ahora veremos nuestra computación como un conjunto de recursos, es decir, Kubernetes va a ver la cantidad total de la que disponemos en nuestros nodos y será él quien se encargue de ver donde es posible desplegar nuestra aplicación.

Finalizando, podemos decir que el paradigma de sistemas ha cambiado, antiguamente teníamos servidores los cuales cuidábamos como mascotas, actualmente estos servidores son vistos como *cattle* (rebaño), en los que gracias a la inmutabilidad podemos hacer, que de forma elástica y dependiendo de la carga, nuestra cantidad de servidores crecerá o decrecerá.

En el **Anexo II: Empezando con Kubernetes** se añade toda la información sobre los primeros pasos con Minikube [10] o como desplegar todo Kubernetes en un mismo nodo.

# 3. Qué necesitamos saber de Kubernetes

Más allá de saber el tipo de arquitectura, o que es una plataforma que se ocupa de la orquestación de contenedores, en éste capítulo introduciremos a más bajo nivel los siguientes componentes de Kubernetes:

- Control Plane.
- Accediendo al clúster.
- Alta disponibilidad y escalabilidad.
- Helm (Kubernetes package manager).

## 3.1 Kubernetes Control Plane

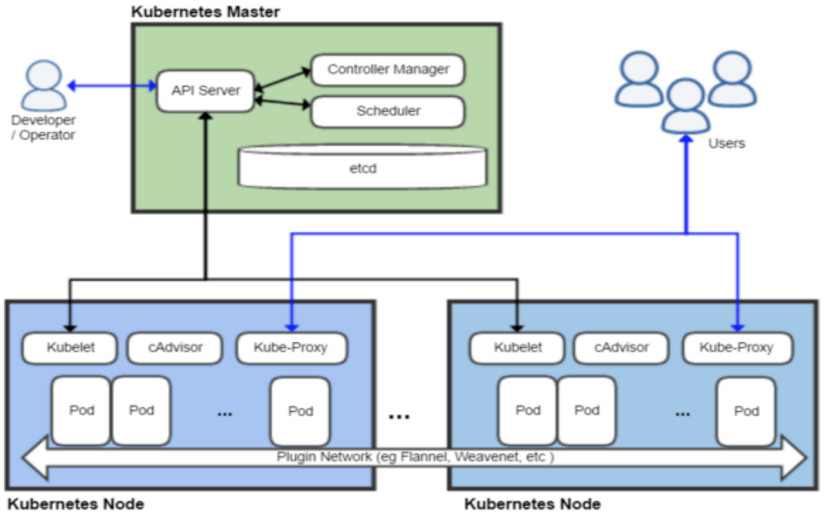
Kubernetes Control Plane [11] mantiene la información de todos los objetos del clúster de Kubernetes. Mediante bucles de conciliación controla su estado. Éstos bucles responden a los cambios en el clúster, manteniendo el estado deseado.

Por ejemplo, si usamos la API para crear o modificar un objeto Deployment, realmente estamos creando un nuevo estado. El Control Plane registrará estas nuevas acciones y actuará consiguiendo el estado deseado por el usuario.

Control Plane tiene 2 componentes base que le permiten llevar a cabo su cometido:

- Master Node [12].
- Worker Node [13].

Figura 7: Infraestructura de Kubernetes.




### 3.1.1 Kubernetes Master Node

El nodo/nodos master se encarga de mantener nuestro estado deseado.

Cuando hablamos con la API usando algún cliente como kubectl o realizando peticiones HTTP directamente, nos comunicamos con el nodo master, el cual se encargará de guardar estas peticiones.

Los procesos que se ocupan de controlar el estado del clúster son:

- kube-apiserver [14]: Es el elemento que se encarga de validar y configurar cómo serán las peticiones REST que se harán desde los clientes. El servidor HTTP que sirve la API.
- kube-controller-manager [15]: Es el *daemon* que se encarga de estar vigilando el estado del clúster y responde creando/aplicando los nuevos cambios gracias a la comunicación con el apiserver.
- kube-scheduler [16]: Este *daemon* es el encargado de ver en qué nodo encajaría nuestra aplicación y como debe de distribuir las aplicaciones en los nodos.
- Etcd: Es la base de datos que mantiene el estado del clúster.

Cuando arrancamos el nodo Master, coge toda la información para arrancar estos procesos con nuestra configuración desde los ficheros *manifest* situados en "`/etc/kubernetes/manifest`".

```
pi@raspberry-1:/etc/kubernetes $ cat manifests/
etcd.yaml                  kube-apiserver.yaml          kube-controller-manager.yaml  kube-scheduler.yaml
```

Los parámetros pueden ser *override* desde la línea de comandos cuando lanzamos el comando *kubeadm* [17].

### 3.1.2 Kubernetes Worker Node

El resto de nodos que contendrán las aplicaciones corriendo ejecutan 2 procesos:

- kubelet [18]: Es el proceso que se encarga de mantener el estado de trabajo así como de mantener el nodo en cuestión. Es el proceso que habla con el nodo master.
- kube-proxy [19]: Proceso cuyo objetivo es crear el enrutado de tráfico a los servicios que están corriendo en el nodo y del balanceo de carga.

## 3.2 Accediendo al clúster

En este punto veremos cómo tanto las aplicaciones como los usuarios deben y pueden autenticarse para poder realizar acciones sobre los recursos del clúster.

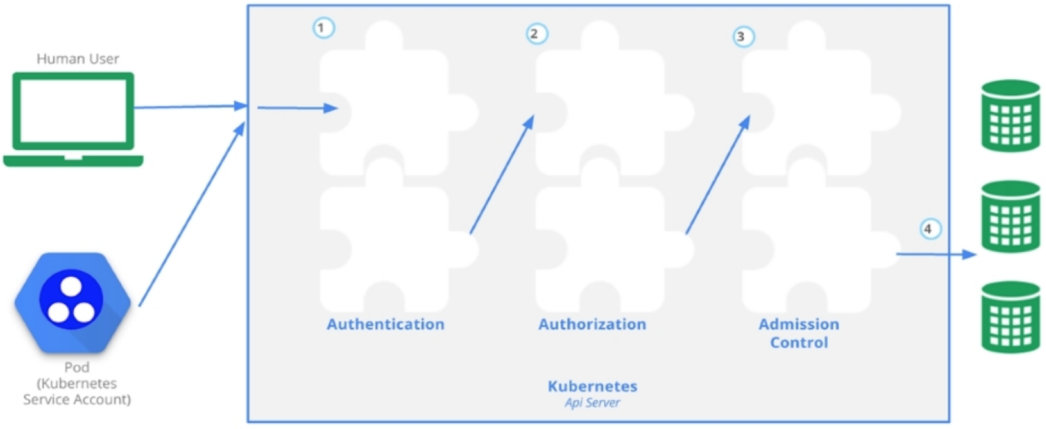
Para asegurar que podemos realizar este tipo de control, Kubernetes nos proporciona las herramientas comentadas a continuación.

### 3.2.1 UserAccount y ServiceAccount

Para poder autenticarnos y autorizarnos, Kubernetes hace distinciones entre *UserAccount y ServiceAccount* . El primero es para darle permisos a los "humanos" usuarios que van a hacer uso del clúster. El segundo es para nuestras aplicaciones "Pods".

- *ServiceAccount*: Son roles cuyo ámbito se limita a los *namespaces*. Estos roles serán asumidos por los *pods* para poder mantener una política de acceso dentro del *namespace*.
  Por defecto, Kubernetes crea un *ServiceAccount* al crear el namespace.
- *UserAccount*: Al igual que los *ServiceAccount* los *UserAccount* nos permiten limitar mediante roles y políticas las acciones de los usuarios para los diferentes recursos de Kubernetes.

Figura 8: Fases autenticación y autorización.




### 3.2.2 RBAC

RBAC (Role Based Access Control), es una de las herramientas que actualmente ofrece Kubernetes (versión 1.8 o superior) para la autorización y acceso a los recursos del clúster.
RBAC mediante el uso de roles hace este control granulado. Existen dos tipos de Role:

- Role: Se limita al namespace, por lo que podremos repetir el nombre del role el diferentes namespaces
- ClusterRole: Su alcance es el clúster, así que tendremos la limitación de nombres por clúster

#### 3.2.2.1 Role

Como hemos comentado los roles nos limitan el acceso del usuario o de las aplicaciones al namespace. En el siguiente ejemplo creamos un Role que actuará sobre el namespace Cloudy como administrador.

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
    name: user-control
    namespace: cloudy
rules:
- apiGroups: [""]
  resources: ["*"]
  verbs: ["*"]
```

Podríamos limitar solo hacer las acciones GET y POST sobre PODS de la siguiente forma:

```
± |master U:3 ?:5 ✗| → cat serviceaacount_example_limit.yaml
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
    name: user-control
    namespace: cloudy
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["GET","POST"]
```

Para aplicar los roles a los usuarios/serviceaccount necesitamos crear el objeto tipo Rolebinding:

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: user-rolebinding
  namespace: cloudy
subjects:
- kind: User
  name: ismael
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: user-control
  apiGroup: rbac.authorization.k8s.io
```

Una vez creado los objetos Role y RoleBinding estamos asignando los permisos para ese servicio/usuario en el namespaces.
Los usuarios pueden tener diferentes permisos en los diferentes namespaces, es decir, más de un role por usuario.

#### 3.2.2.2 Cluster Role

El fin es el mismo que el Role pero aplicado al clúster. Tendremos usuarios con limitaciones y estas limitaciones vienen dadas por un ClusterRole. Para añadir esta serie permisos granulares necesitamos un ClusterRoleBinding.

## 3.3 Alta disponibilidad y escalabilidad.

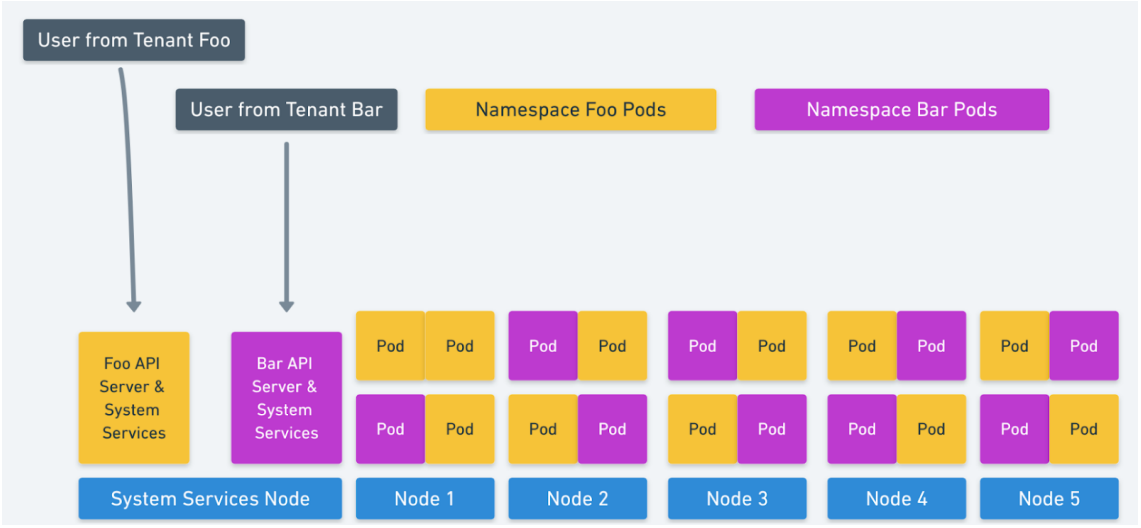
En éste capítulo veremos cómo Kubernetes gestiona las aplicaciones y cómo podemos escalarlas para mantener la disponibilidad.

### 3.3.1 Gestión de aplicaciones

Como hemos comentado en varias ocasiones Kubernetes gestiona el clúster como una masa de recursos, en la cual y gracias al kube-scheduler sabe la capacidad de los workers en todo momento.

Nuestras aplicaciones como ya sabemos se despliegan en pods y estos en namespaces. Los namespaces son separaciones lógicas que necesitamos para mantener, por ejemplo, los entornos de desarrollo, preproducción y producción de forma aislada.
Esta capa llamada namespace nos abstrae totalmente de donde realmente se ejecuta “físicamente” nuestra app.

Figura 9: Visualización de los namespaces como cross-node




Por lo que, aunque hayamos escalado o desplegado varias aplicaciones en el mismo namespace, nuestra aplicación acabará distribuida por los n nodos que tengamos, dotándola de alta disponibilidad.

### 3.3.2 Escalando / Auto escalando nuestra aplicación

Primero, para conseguir alta disponibilidad necesitamos tener varias replicas de nuestra aplicación, para conseguir este objetivo en el objeto deployment debemos de indicar el número de réplicas deseado.

En la siguiente imagen observamos el fichero manifest de un objeto deployment, en el cual, observamos lo siguientes parámetros:

- Namespace: Indica en que namespace se desplegará nuestra app.
- Replicas: Indica el número de copias que tendrá nuestra aplicación dentro del clúster.

```
pi@raspberrypi:~ $ cat nginx.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  namespace: cloudy
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

Éste número de copias puede cambiar "*on-demand*" mediante linea de comandos o modificando y aplicando el manifest.
Éste tipo de escalado es conocido cómo HPA [20] *(Horizontal pod autoscaling).*

Pero es *autoscaling* realmente? La respuesta es NO. Para poder conseguir *autoscaling* necesitamos de métricas y un *threshold* para delimitar cuándo es necesario incrementar o decrementar el número de copias.

En la imagen a continuación vemos como declaramos un objeto tipo *HorizontalPodAutoscaler*.

```
pi@raspberrypi:~ $ cat nginx_autoscaling.yaml
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: nginx
  namespace: cloudy
spec:
  maxReplicas: 4
  minReplicas: 2
  targetCPUUtilizationPercentage: 90
  scaleTargetRef:
    apiVersion: v1
    kind: ReplicationController
    name: nginx
```

De los parámetros cabe destacar:

- namespace: El ámbito donde el autoscaling actuará
- maxReplicas: El número máximo de réplicas deseado.
- minReplicas: El número mínimo de réplicas deseado.
- targetCPUUtilizationPercentatge: La métrica y threshold para escalar.

Con todas estas herramientas ya podemos tener nuestras aplicaciones en alta disponibilidad y escalar a medida (podemos escalar por estas *native metrics* [21] o por *custom metrics*)

Comentar también que la estrategia de escalado es haciendo rolling-update [22] para evitar el *downtime.*

### 3.4 Helm (Kubernetes package manager)

Uno de los proyectos/plugins más importantes y de más uso es HELM [23]. Y es que, nos facilita mucho la vida a la hora de gestionar nuestras aplicaciones al igual que hace APT o YUM.

Con HELM podemos:

- Compartir nuestras aplicaciones.
- Aplicaciones fáciles de actualizar y mantener
- Centralizamos el componente con permisos para instalar las aplicaciones.

### 3.5 Conclusiones

Si hacemos un poco de recapitulación, Kubernetes está creciendo de forma considerable si hace dos años se encargaba solo de orquestar containers a día de hoy encontramos:

- Autoescalabilidad: Dando fiabilidad y resiliencia al clúster.
- Seguridad: Mediante Role y ClusterRole tendremos controlado el acceso de forma granulada tanto de aplicaciones como usuarios sobre los recursos del clúster.
- Estabilidad y tolerancia a particiones: Dada la arquitectura de Kubernetes, podríamos perder el nodo master que nuestras aplicaciones no caerían.
- Usabilidad: Mediante plugins como HELM podemos desplegar y mantener nuestras aplicaciones de forma sencilla e intuitiva.

# 4. Diseño: Integración entre  Kubernetes y Cloudy

El objetivo de éste punto es aclarar los problemas, ver qué soluciones podemos aplicar para usar Kubernetes y Cloudy y por último ver si está solución es válida.

## 4.1 Escenarios propuestos

Exponemos varios escenarios en los que Cloudy puede beneficiarse de K8s.

### 4.1.1 – Escenario 1: Red Local

#### 4.1.1.1 Objetivo

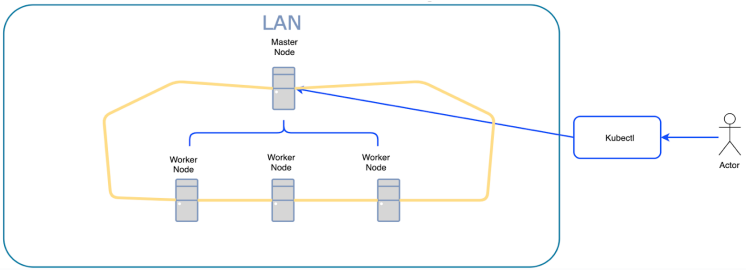
Teniendo desplegado un clúster de kubernetes en una red local queremos que Cloudy haga el autodiscovery de las aplicaciones.

Para ello hacemos las siguientes asunciones:

- Tenemos una red local del tipo C (192.168.1.1/24)
- Todos nuestros nodos están Cloudynizados [24].
- Instalamos versión 1.14 de k8s en la cual tendremos un clúster formado por:
    - 1 o más Nodos Master
    - 3 o más Nodos Workers

En la siguiente imagen se observa un *overview* de éste primer escenario

Figura 10: Escenario LAN

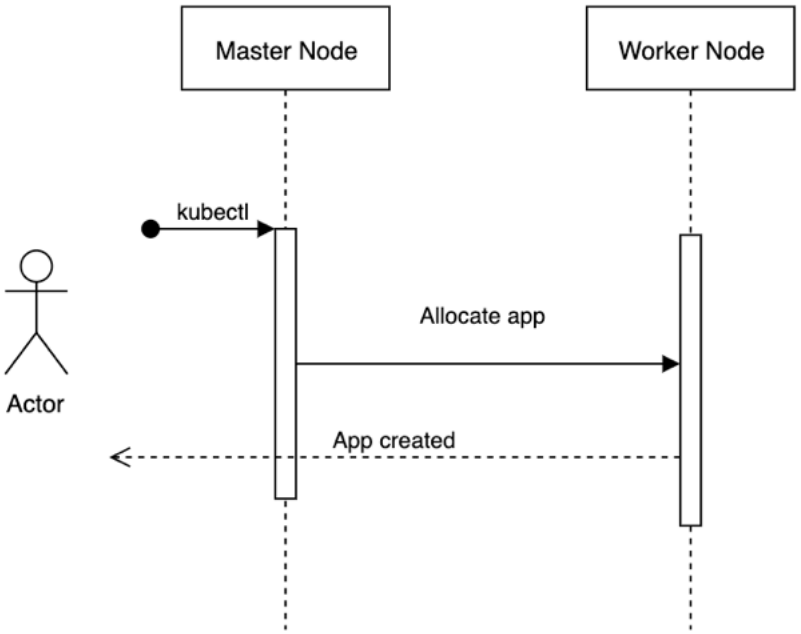



### 4.1.1.2 Diagrama de flujo

En primera instancia, el cliente hablará con la API de Kubernetes indicando que servicio y en qué estado lo quiere. Dicho estado se efectuará mediante una petición POST y un fichero *manifest*.

En este momento el nodo master mediante el proceso *kube-scheduler* aloja nuestra aplicación en el nodo donde mejor encaje.

Figura 11: Diagrama de flujo usuario creando app en  kubernetes

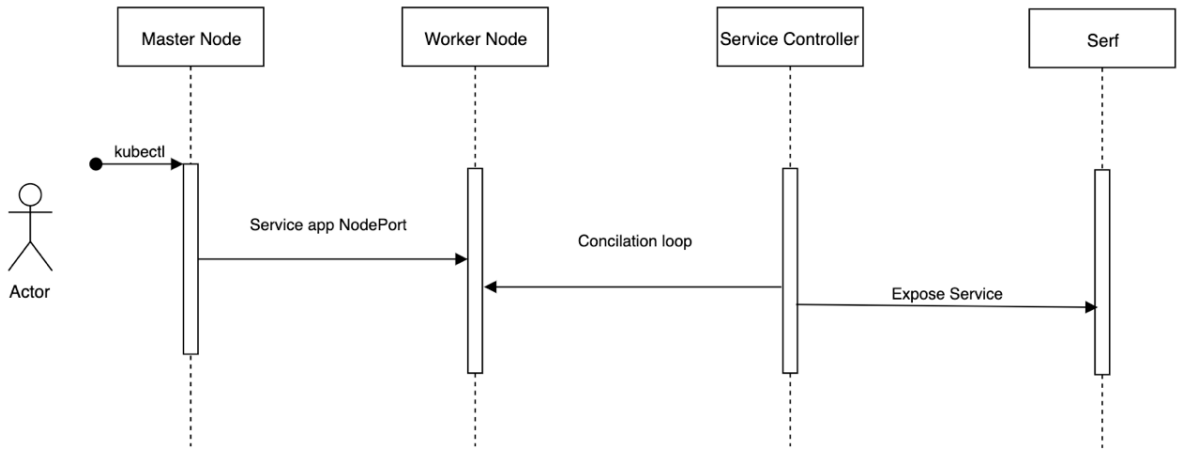



### 4.1.1.3 Exponiendo nuestra aplicación mediante SERF

Kubernetes permite añadir nuevas *features* y extender así sus funcionalidades para cumplir con nuestros requisitos.

Nosotros vamos a extender las funcionalidades de Kubernetes de modo que,  cuando llegue el evento de creación de un nuevo objeto tipo *Service*, nuestra aplicación hará una llamada al sistema pasándole los parámetros a Avahi para que la exponga mediante Serf.

Figura 12**:** Diagrama de flujo usuario creando app en  kubernetes y exponiendo a través de SERF




En la figura anterior se observan los pasos explicados, donde básicamente:

- Añadimos un nuevo Service a Kubernetes mediante la API.
- Al controller le llega el cambio de estado (add new Service).
- El controller llama a Avahi el cual expone mediante Serf nuestra app al resto de nodos.

#### 4.1.1.4 Cómo conseguiremos el objetivo del escenario

Para llegar al objetivo propuesto, lo primero que debemos hacer es poder hablar con la API de Kubernetes bien desde Cloudy o bien desde la línea de comandos.

Necesitaremos un sistema de autenticación/autorización para decir al clúster quien soy y que puedo hacer.
Al ser todo en local, controlado y centralizado hacemos uso de la autenticación mediante *Static Token File* [25]. Este fichero está compuesto por:

```
token,user,uid,"group1,group2,group3"
```

El fichero se define en el master node:

```
root@raspberry-1:/etc/kubernetes# ls -al token.csv && cat token.csv
-rwxr-xr-x 1 root root 39 May 20 18:05 token.csv
TokenOfTheJ0n,ismael,1
yeahyeah,jose,2
```

Además, para decirle a Kubernetes el sistema de autenticación debemos de indicar lo bien mediante línea de comandos o especificando lo en el fichero manifest de la API:

```
    - --token-auth-file=/etc/kubernetes/token.csv
```

Tendremos que montar el volumen para que el fichero se lea desde dentro del clúster:

```
 volumeMounts:
 - mountPath: /etc/kubernetes/
   name: token
   readOnly: true
```

```
volumes:
- hostPath:
    path: /etc/kubernetes
    type: DirectoryOrCreate
  name: token
```

Configuramos el fichero config para poder hacer peticiones a kubernetes:

```
apiVersion: v1
clusters:
- cluster:
    server: https://192.168.1.40:6443
  name: cluster-token
contexts:
- context:
    cluster: cluster-token
    user: ismael
  name: context-token
current-context: context-token
kind: Config
preferences: {}
users:
- name: ismael
  user:
    token: TokenOfTheJ0n
```

Ahora ya podemos hacer peticiones al clúster con nuestro usuario.

Teniendo la autenticación resuelta vamos a darle permisos a al usuario para que sólo pueda realizar acciones sobre el namespace “Cloudy”.

Desde el nodo master, creamos el nuevo namespaces Cloudy.

```
pi@raspberry-1:~ $ kubectl create namespace cloudy
```

Mediante RBAC (*role-based access control*) [26] podemos controlar la autorización de forma granulada en el clúster.
Aplicamos el siguiente Manifest que contiene el role RBAC:

```
pi@raspberry-1:~/kubernetes/roles $ cat user-cloudy.yaml
kind: Role
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  namespace: cloudy
  name: ismael-role
rules:
- apiGroups: ["*"]
  resources: ["*"]
  verbs: ["*"]
---
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: ismael-rolebinding
  namespace: cloudy
subjects:
- kind: User
  name: ismael
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: ismael-role
  apiGroup: rbac.authorization.k8s.io
```

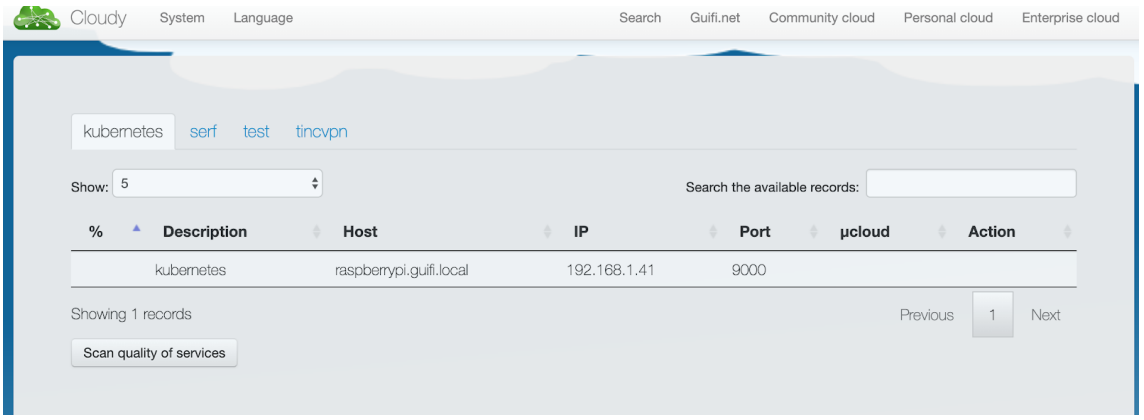
Como se observa en la imagen previa, estamos dando permisos de administrador al usuario ismael en el el namespace Cloudy (hemos explicado RBAC en el punto 3.2 de ésta memória)
Ahora tenemos acceso al namespace.

```
2019-06-02 10:27:26  MacBook-Pro-de-Ismael-2 in ~/vpn
○ → kubectl get pods serf-publisher-58c777678d-gj7tz  --insecure-skip-tls-verify=true  -o wide -n cloudy
NAME                              READY     STATUS    RESTARTS   AGE       IP            NODE          NOMINATED NODE   READINESS GATES
serf-publisher-58c777678d-gj7tz   1/1       Running   0          4d14h     10.244.1.72   raspberrypi   <none>           <none>
```

Y acceso denegado al resto del clúster.

```
2019-06-02 10:27:34  MacBook-Pro-de-Ismael-2 in ~/vpn
○ → kubectl get pods serf-publisher-58c777678d-gj7tz  --insecure-skip-tls-verify=true  -o wide
No resources found.
Error from server (Forbidden): pods "serf-publisher-58c777678d-gj7tz" is forbidden: User "ismael" cannot get resource "pods" in API group "" in the namespace "default"
```

Una vez tenemos acceso al clúster, debemos de poder exponer los servicios que tenemos corriendo en Kubernetes a Cloudy.
Hacemos uso de nuestro controller serf-publisher [27].



Ya tenemos nuestro servicio de prueba expuesto a través de Cloudy y corriendo en Kubernetes.

#### 4.1.1.5 Conclusión

Sin tener muchas dificultades y conociendo las arquitecturas tanto de Cloudy como Kubernetes podemos exponer nuestros servicios teniendo en cuenta:

- Es una red local, estamos dando servicio en nuestra casa o mediante VPN, pero siempre conociendo el dominio y exponiendo aplicaciones nuestras.
- Normalmente tendremos latencias bajas.
- Control total para escalar el clúster.
- Control total para parar el clúster.

### 4.1.2 Escenario 2: Extrapolado a Cloudy Distribuido con un clúster de Kubernetes

#### 4.1.2.1 Objetivo

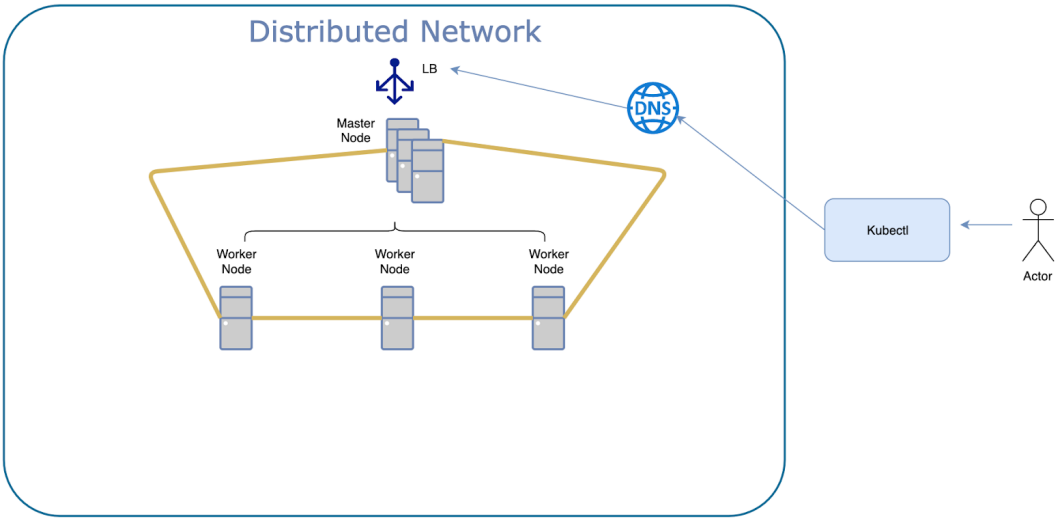
Usar Kubernetes en Cloudy de forma distribuida, soportar diferentes dominios y mantener el clúster con alta disponibilidad y tolerancia a particiones.

Para ello hacemos las siguientes asunciones:

- Tenemos una red distribuida y heterogénea.
- Todos nuestros nodos están Cloudynizados.
- Instalamos versión 1.14 de k8s en la cual tendremos un clúster formado por:
    - N Nodos Master
    - N Nodos Workers
- Debemos de asumir que nuestras aplicaciones no tienen estado, es decir, son *stateless.*
- Todas las aplicaciones deben de estar disponibles mediante Cloudy.

En la siguiente imagen se observa un *overview* del segundo escenario.

Figura 13: Escenario Red Distribuida




***Nota:*** *No expongo los diagramas de flujo ya que son iguales que en el primer escenario.*

#### 4.1.2.2 Cómo conseguiremos el objetivo del escenario

Al igual que en la red local, necesitaremos accesos a la API de Kubernetes.
Se deberán de crear usuarios y namespaces específicos para los diferentes dominios, es decir, se creará un namespace y un usuario por dominio, de forma que sólo el usuario propietario del namespace tendrá permisos CRUD [28] sobre sus aplicaciones.

Todo nodo Cloudy que use Kubernetes deberá ser parte del clúster de Kubernetes, con ello aseguramos al menos una copia de nuestra aplicación estará siempre disponible. Esta acción la llevaremos a cabo *"taggeando"* [29] tanto nuestras aplicaciones como nuestros nodos.

Cualquier nodo Cloudy debe de poder ser master en Kubernetes para mantener la alta disponibilidad del clúster.

Mediante el controller creado serf-publisher, expondremos las aplicaciones al resto de nodos Cloudy.

Además se deberá crear la siguiente infraestructura:

- Balanceador de carga sobre los nodos master para evitar hacer un DDOS a la API.

- Creación de un DNS para hablar humanamente con el clúster.

#### 4.1.2.3 Viabilidad del escenario

En este escenario se observan varios problemas que nos obligan a decir que es **NO** es un escenario viable.

- No se puede asegurar que los nodos master estén siempre activos ya que al ser un cloud colaborativo los sistemas Edge pueden estar o no disponibles, con lo que nuestra aplicación podría estar en un estado diferente al deseado.
- Seguridad, los nodos Masters van a poder administrar, actualizar, parar y borrar nuestras aplicaciones para sus necesidades. Es decir, se pierde el control de nuestra disponibilidad.

#### 4.1.2.4 Conclusiones

Siendo un escenario fácil de implementar e implantar vemos que no es viable, ya que no nos proporciona la seguridad adecuada para cumplir con el objetivo de alta disponibilidad de nuestra aplicación.

Lo bueno de éste escenario es que si se consigue la posibilidad de tener HA y resiliencia del clúster estaríamos usando toda la potencia de Kubernetes:

*“Utilizo el clúster como una masa de recursos, no importa donde, importa que el estado deseado se aplique siempre que se pueda”*

### 4.1.3 Escenario 3: Extrapolado a Cloudy Distribuido con N clusters de Kubernetes (Multiclúster)

#### 4.1.3.1 Objetivo

Mantenemos el objetivo del Escenario anterior:
Usar Kubernetes en Cloudy de forma distribuida, soportar diferentes dominios y mantener el clúster con alta disponibilidad y tolerancia a particiones.
Todas las aplicaciones deben de estar disponibles mediante Cloudy.

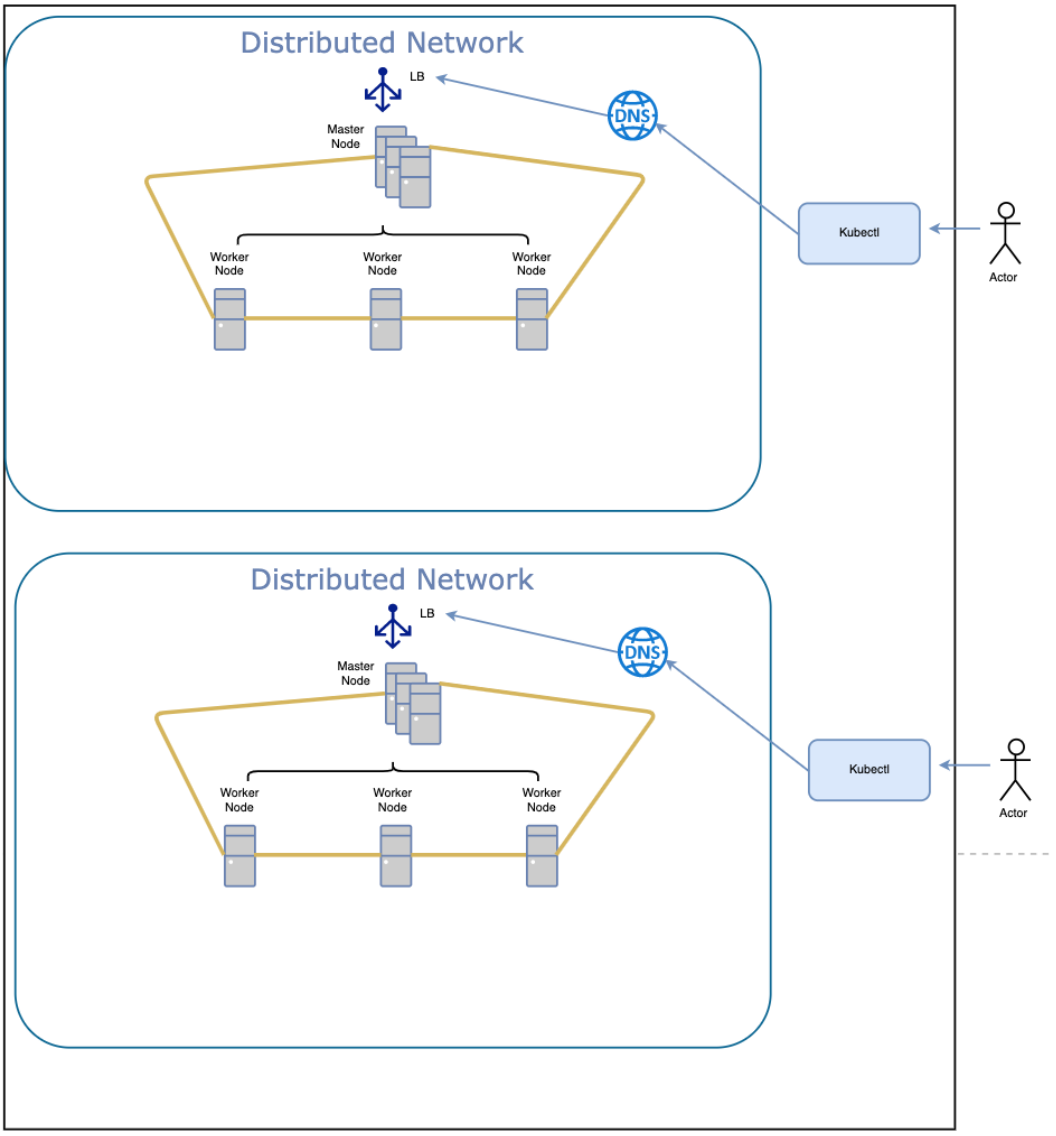
Para ello hacemos las siguientes asunciones:

- Tenemos una red distribuida y heterogénea.
- Todos nuestros nodos están Cloudynizados.

- Instalamos versión 1.14 de k8s en la cual tendremos N clusters formados por:
    - N Nodos Master
    - N Nodos Workers
- Nuestras aplicaciones no tienen estado, es decir, son *stateless.*

En la siguiente imagen se observa un overview del tercer escenario.

Figura 14: Escenario multi clúster




***Nota:*** *No expongo los diagramas de flujo ya que son iguales que en el primer escenario.*

#### 4.1.3.2 Cómo conseguiremos el objetivo del escenario

La arquitectura y elementos son los mismos que en el segundo escenario pero con los siguientes matices:

- Cada aplicación/dominio será propietario de su clúster.
- Cada aplicación/dominio deberá de asegurarse de hablar con la api de su clúster.
- La administración deja de ser distribuida para ser localizada en cada clúster.
- Podríamos omitir el balanceador de carga y el DNS (no recomendado).
- Mediante el controller creado serf-publisher, expondremos las aplicaciones al resto de nodos Cloudy.

#### 4.1.3.3 Viabilidad del escenario

Escenario totalmente viable, se solventan los problemas encontrados con la seguridad en el apartado anterior. Además, la administración pasa a ser centralizada (cada clúster tendrá su control).

#### 4.1.3.4 Conclusiones

A simple vista no vemos ningún problema añadido para tener N clusters de Kubernetes.
El “problema” es que no podemos usar todos los nodos Cloudy para escalar nuestras aplicaciones, es decir, el despliegue de nuestras aplicaciones están limitadas a los recursos que ofrezca nuestro clúster.

### 4.2 Selección de escenario

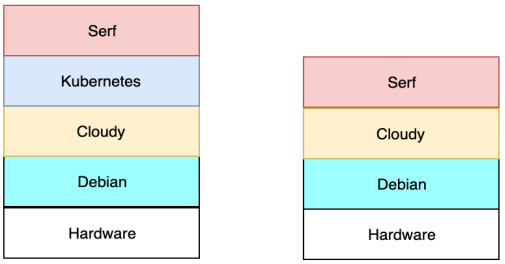
En este proyecto cubriremos el desarrollo del primer escenario (Red Local) e intentaremos cubrir parcialmente el último (multiclúster [30]), descartando totalmente el segundo por los problemas encontrados.

Dada la infraestructura que tenemos, podremos sin problemas desplegar un clúster en local.
Para multiclúster, además de la infraestructura física dispuesta, tenemos a nuestra disposición máquinas Virtuales y raspberries de GUIFI.

Usando el diagrama del tercer escenario, crearemos 2 clúster uno con las máquinas que nos han ofrecido y otro con las locales.

En ambos casos usaremos el operator serf publisher  para exponer nuestras aplicaciones mediante SERF.

Figura 15: Arquitectura de un nodo usando Cloudy y Kubernetes.



# 5. Implementación

En la fase de diseño hemos visto diferentes escenarios y diseños que encajan con la arquitectura de Cloudy y Kubernetes.
En este apartado veremos como se ha realizado la implementación de ambos y qué elementos hemos requerido para ello.
Dividiremos la implementación en 2 partes:

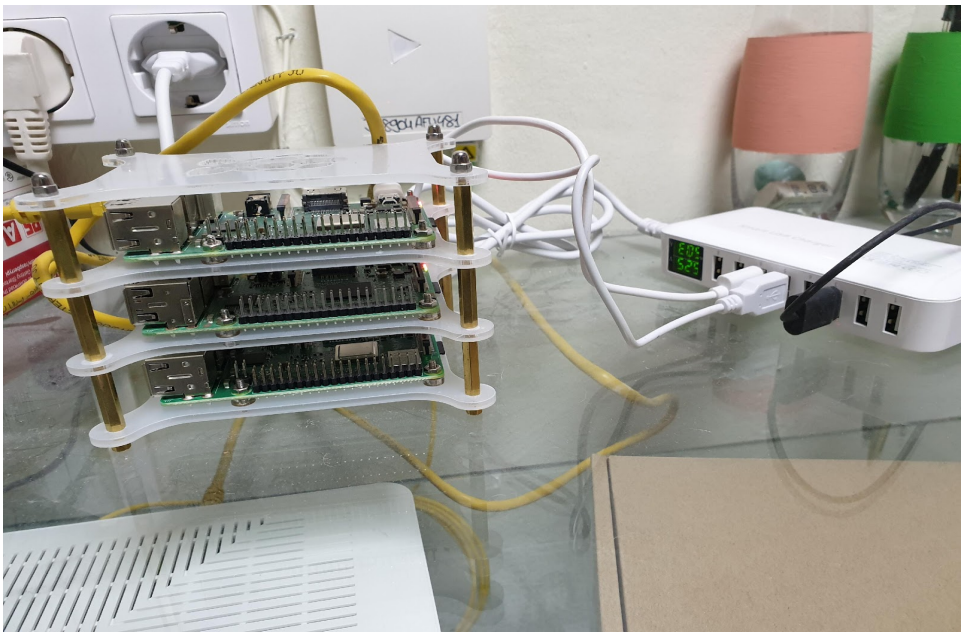
- Implementación en una red local (un único clúster).
- Implementación en una red distribuida y heterogénea (multiclúster).

## 5.1 Infraestructura de la microCloud (red local)

Usamos 3 raspberries pi 3+ con las siguientes características:

- Procesador quad-core de 64 bits con 1,4 Ghz
- Banda dual de 2,4 Ghz.
- LAN inalámbrica de 5 Ghz.
- Bluetooth 4.2 / BLE.
- Ethernet.
- 1GB de RAM.
- Tarjeta microSDHC 32GB.
- Hub USB de 8 puertos para alimentar las raspberries.

Figura 16: Cluster de raspberries y cargador USB



## 5.2 Entorno de desarrollo

Para el desarrollo y mantenimiento de los scripts que creados, se usará el control de versiones GIT [31] y en concreto 2 repositorios:

- Scripts/helpers para instalar y hacer el setup (tfg [32]) de los nodos.
- Código de el operator (serf-publisher).

Usamos GoLand como IDE para el desarrollo del controller.
Usamos VIM para los shell scripts.
Hemos incluido algunos *Playbooks* de Ansible [33] para automatizar el el aprovisionamiento de los nodos.

## 5.3 Instalando Kubernetes

Al igual que en apartado anterior haremos uso de shell scripts y Ansible para automatizar el proceso de instalación de Kubernetes, en el cual debemos de instalar:

- Docker
- kubeadm

Además el proceso realizará una serie de cambios necesarios para arrancar Kubernetes sin problemas:

- Deshabilitar la memoria swap, ya que Kubernetes hará uso de toda la RAM disponible en la toma de decisiones para el *scheduler* de nuestras aplicaciones.
- En la arquitectura de nuestras Raspberries el kernel no viene cgroups de memoria habilitados.
  Los habilitamos añadiendo al fichero "/boot/cmdline.txt" la siguiente línia "cgroup_enable=cpuset cgroup_enable=memory"

```
pi@worker-2:~ $ cat /proc/cgroups
#subsys_name    hierarchy       num_cgroups     enabled
cpuset  4       1       1
cpu     3       1       1
cpuacct 3       1       1
blkio   6       1       1
memory  0       1       0
devices 2       50      1
freezer 7       1       1
net_cls 5       1       1
```


El script que automatiza este proceso se puede encontrar en el siguiente repositorio installer [34].

Para ejecutar lo lanzamos el siguiente comando:

```
± |master U:3 ?:3 ✗| → ansible-playbook -i host installer.yaml -u pi --ask-pass
```

Hacemos los mismo para el resto de nodos.

## 5.4 Arrancando el clúster de Kubernetes

Entramos por ssh al que será nuestro nodo master y lanzamos el siguiente comando:

```
root@raspberry-1:~# kubeadm init --kubernetes-version 1.14.1 --pod-network-cidr=10.244.0.0/16 | tee kubeadm-init.out
```

Se debe de lanzar como root y además debemos pasar el flag `--pod-network-cidr=10.244.0.0/16` para asegurarnos que la variable podCIDR será *"seteada".*

**Nota:** Aclarar que éste parámetro lo seteamos por el uso de Flannel como CNI [35] y arrancamos el clúster con kubeadm.

Al acabar, el *output* nos indica cómo podemos hablar con la API.

- Hemos de copiar el fichero que se ha generado a la home del usuario.
- Cambiar el propietario del fichero.

```
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

El fichero que hemos copiado no es más que el manifest del objeto config el cual nos da permisos de administración sobre todo el clúster.

Comprobamos que tenemos acceso a la API.

```
pi@raspberry-1:~ $ kubectl get nodes
NAME          STATUS   ROLES    AGE   VERSION
raspberry-1   Ready    master   13d   v1.14.1
```

**Nota:** Como mostramos en el anexo V no es buena práctica compartir ésta configuración con el resto de clientes.

Para añadir nodos al clúster lanzamos el siguiente comando desde el nodo master:

```
pi@raspberry-1:~ $ kubeadm token create --print-join-command
```

El cual nos devuelve el comando a ejecutar en el resto de nodos que queremos adjuntar.
Lanzamos el siguiente comando en los nodos:

```
pi@worker-2:~ $ sudo kubeadm join 192.168.1.40:6443 --token km90l7.eezaswpygljmcnxb     --discovery-token-ca-cert-hash sha256:a11f6625cdb99a230c2235f07ea7f8a2402d913a76389a34f3bbbb3af029adb9
[preflight] Running pre-flight checks
[preflight] Reading configuration from the cluster...
[preflight] FYI: You can look at this config file with 'kubectl -n kube-system get cm kubeadm-config -oyaml'
[kubelet-start] Downloading configuration for the kubelet from the "kubelet-config-1.14" ConfigMap in the kube-system namespace
[kubelet-start] Writing kubelet configuration to file "/var/lib/kubelet/config.yaml"
[kubelet-start] Writing kubelet environment file with flags to file "/var/lib/kubelet/kubeadm-flags.env"
[kubelet-start] Activating the kubelet service
[kubelet-start] Waiting for the kubelet to perform the TLS Bootstrap...

This node has joined the cluster:
* Certificate signing request was sent to apiserver and a response was received.
* The Kubelet was informed of the new secure connection details.

Run 'kubectl get nodes' on the control-plane to see this node join the cluster.
```

Comprobamos que los workers se han añadido correctamente:

```
pi@raspberry-1:~ $ kubectl get nodes
NAME          STATUS   ROLES    AGE     VERSION
raspberry-1   Ready    master   13d     v1.14.1
raspberrypi   Ready    <none>   12d     v1.14.1
worker-2      Ready    <none>   7m52s   v1.14.2
```

### 5.5 Instalando Cloudy en los nodos

Seguimos los pasos para cloudinizar nuestros nodos.
**Nota:** En el Anexo I hay toda la información de la instalación.

Comprobamos que los 3 nodos tienen Cloudy instalado comprobando la conexión al socket {IP}:7000 con NCAT [36].

```
○ → ips=(192.168.1.40 192.168.1.41 192.168.1.42) &&  for ip in ${ips[*]}; do $(ncat -zv $ip 7000); done
Ncat: Version 7.70 ( https://nmap.org/ncat )
Ncat: Connected to 192.168.1.40:7000.
Ncat: 0 bytes sent, 0 bytes received in 0.10 seconds.
Ncat: Version 7.70 ( https://nmap.org/ncat )
Ncat: Connected to 192.168.1.41:7000.
Ncat: 0 bytes sent, 0 bytes received in 0.09 seconds.
Ncat: Version 7.70 ( https://nmap.org/ncat )
Ncat: Connected to 192.168.1.42:7000.
Ncat: 0 bytes sent, 0 bytes received in 0.11 seconds.
```

### 5.6 Corriendo nuestro operator serf-publisher

Ya tenemos el clúster de Kubernetes arrancado y los nodos Cloudinizados.

Lo que queremos ahora es capturar los eventos de creación de un *Service NodePort.* Para ello hemos creado el operator, su función es escuchar y exponer mediante un llamada a “/usr/sbin/avahi-ps publish”.

Para comprobar este funcionamiento hacemos la prueba de lanzar el operator en el nodo master, crear un service, ver el resultado en Cloudy y realizar una petición desde fuera del clúster.

Lanzamos serf-publisher

```
pi@raspberry-1:~ $ ./serf-publisher-arm --namespace cloudy --kubeconfig /home/pi/.kube/config
{"level":"warn","ts":1559660386.8211348,"caller":"cmd/root.go:101","msg":"Falling back to using kubeconfig file: unable to load in-cluster configuration, KUBERNETES_SERVICE_HOST and KUBERNETES_SERVICE_PORT mu
st be defined"}
{"level":"warn","ts":1559660386.8464415,"caller":"controller/generic.go:55","msg":"no metrics recorder specified, disabling metrics"}
{"level":"info","ts":1559660386.84675,"caller":"controller/generic.go:147","msg":"starting controller"}
{"level":"info","ts":1559660387.9581785,"caller":"pkg/service.go:38","msg":"command \nSuccessfully updated agent tags\n\n"}
```

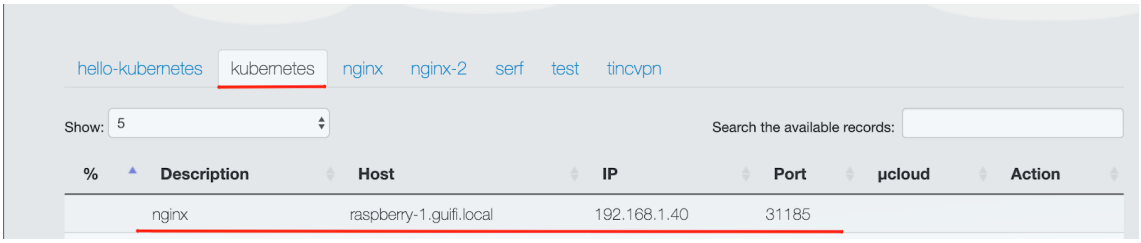
Exponemos nuestro nginx (creamos un service):

```
○ → kubectl1.6 -v=9 expose deployment/nginx --insecure-skip-tls-verify=true -n cloudy --type=NodePort
```

```
○ → kubectl1.6 get service -n cloudy --insecure-skip-tls-verify=true
NAME      CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
nginx     10.96.68.175    <nodes>       80:31185/TCP   4m
```

El operator captura y expone en Cloudy:

```
{"level":"info","ts":1559660881.158088,"caller":"pkg/service.go:38","msg":"command \nSuccessfully updated agent tags\n\n"}
```




```
± |master S:3 ?:2 x| → curl -i 192.168.1.40:31185
HTTP/1.1 200 OK
Server: nginx/1.15.12
Date: Tue, 04 Jun 2019 15:15:40 GMT
Content-Type: text/html
Content-Length: 612
Last-Modified: Tue, 16 Apr 2019 13:08:19 GMT
Connection: keep-alive
ETag: "5cb5d3c3-264"
Accept-Ranges: bytes

<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

En el **Anexo VII: Código** se añade todo el código del operator serf-publisher, además se ha subido al siguiente repositorio de github: serf-publisher.

### 5.6.1 Análisis de la solución

En este apartado veremos las capacidades y limitaciones que tiene actualmente nuestra solución.

#### 5.6.1.1 Capacidades

Como hemos visto en el apartado anterior somos capaces de exponer servicios desde Kubernetes y que llegue al Backoffice de Cloudy.
Lo conseguimos mediante un *operator* de Kubernetes, lo único que estamos haciendo es suscribirnos a los eventos de la API de Kubernetes, concretamente los eventos sobre los objetos *Service*, y llamar a Avahi para hacer la publicación mediante Serf.

Esta solución nos permite de forma sencilla poder tener la integración de ambos servicios, sin hacer modificaciones en la arquitectura de Cloudy y con una instalación sencilla de Kubernetes.

#### 5.6.1.2 Limitaciones

Con nuestra solución hemos cumplido con el objetivo marcado, pero hay que subrayar y aclarar las limitaciones en ellas.

1.Despliegue:

Una vez creado serf-publisher y probado en Minikube, queríamos desplegarlo en nuestro clúster de Kubernetes y siguiendo los siguiente pasos:

- Compilar para ARM con GO y generar el binario.
  https://github.com/ismferd/serf-publisher#generation-of-binary
- Crear un imágen docker para desplegar en Kubernetes y subirla a un repositorio de DockerHub.
  https://github.com/ismferd/serf-publisher#docker-release
  https://cloud.docker.com/u/ismaelfm/repository/docker/ismaelfm/serf-publisher
- Creamos el ServiceAccount para que nuestra app pueda actuar sobre los objetos *Service*.
- Crear un *manifest* con la información necesaria para el despliegue:

```
pi@raspberrypi:~ $ cat serf-publisher-deployment.yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: serf-publisher
  labels:
    app: serf-publisher
spec:
  replicas: 1
  selector:
    matchLabels:
      app: serf-publisher
  template:
    metadata:
      labels:
        app: serf-publisher
    spec:
      containers:
        - name: serf-publisher
          image: ismaelfm/serf-publisher:0.1.0
          env:
          - name: NAMESPACE
            value: default
```

Una vez desplegada, nos damos cuenta de que el resultado no es el esperado, ya que no estaba exponiendo los servicios. El problema realmente son las llamadas al sistema que hacemos para publicar mediante Avahi. Avahi está instalado pero en los hosts de los nodos, así que Kubernetes no puede ver éste proceso.

El *workaround*, fue crear un *daemon* en systemd y que sólo se ejecute en el nodo máster de los N clusters.

```
pi@raspberry-1:~/kubernetes $ cat /etc/systemd/system/serf-publisher.service
[Unit]
Description=Serf Publisher Service
After=kubelet.service

[Service]
Type=simple
ExecStart=/home/pi/serf-publisher-arm --namespace cloudy --kubeconfig /home/pi/.kube/config
```

Pasándole como parámetros el namespace y la configuración del usuario.

2. Posible Inconsistencia de datos Cloudy-Kubernetes

Nuestro operator, mantiene en memoria los objetos *Service* para poder hacer la llamada *unpublish* de Avahi, es decir:

- Creamos un nuevo service mediante el comando `kubectl expose deployment/nginx -n Cloudy --type=NodePort`
- Serf-publisher expone este servicio mediante Avahi-Serf y lo guarda en memoria mediante un *map*.
- Borramos éste *Service* `kubectl delete service nginx -n Cloudy`
- Obtenemos el Service de memoria y hacemos *unpublish* en Avahi-Serf.

El problema existe  si ha nuestro proceso serf-publisher le llega un SIGTERM o SIGKILL, perderemos los objetos en memoria y no podremos eliminar los de Cloudy, con lo que no tendremos consistencia de datos entre Cloudy y Kubernetes

3. Tipo de exposición

Como sabemos existen 3 tipos de *Service* en Kubernetes. Nosotros estamos limitados a usar el tipo *NodePort* si queremos que el resto del clúster de Cloudy pueda hacer uso de nuestras aplicaciones.

4. Multi-Cluster

Aunque por un lado es beneficioso, digamos que cada dominio tiene control sobre sus aplicaciones, por otro lado, perdemos la oportunidad de que cada servicio pueda ser asignado a cualquier nodo de Cloudy.
Nos referimos a que nuestras aplicaciones están limitadas a nuestro clúster de Kubernetes.

5. Añadir funcionalidades desde Cloudy

Actualmente el backoffice o frontend que proporciona Cloudy añade lógica a los recursos que proporciona como es el simple hecho de poder hacer un `docker start o docker stop.` Con nuestra solución no podemos añadir este tipo de acciones ya que hacemos el *publish* directamente  sin pasar por el php.

#### 5.6.1.3 Conclusión de la solución

Podríamos decir que el objetivo se ha logrado, aunque con algunos *cons* importantes como es la posible inconsistencia entre ambas plataformas.
De forma fácil, sencilla y sin modificar la arquitectura de Cloudy podemos trabajar con Kubernetes haciendo uso de su potencial.
La limitación por parte de Cloudy es que no podemos hacer uso de su interfaz, sólo veremos los servicios expuestos desde Kubernetes.

# 6. Conclusiones finales

**Conclusiones del trabajo:**

- Se ha investigado diferentes situaciones en la que los productos analizados se pueden ser complementarios.
- Se ha implementado e aprendido GO. El código del *operator* que hace la integración de ambos sistemas es en GO.
- Se crean varios scripts y playbooks para facilitar el despliegue de Kubernetes y Cloudy en varios nodos.
- Se ha aprendido Kubernetes la cual no es una plataforma sencilla y trivial.

**Consecución de los objetivos:**
Los objetivos principales que si recordamos era implementar Kubernetes en nodos Cloudy, se han conseguido de forma satisfactoria.
Cuando hemos estudiado las 2 plataformas nos hemos dado cuentas de que el objetivo de las 2 es muy similar y se solapan, es decir, desde las 2 se pueden exponer servicios y aplicaciones, aún así, hemos conseguido que colaboren entre si.

**Seguimiento de la planificación y metodología:**
Sobre la planificación, hemos intentado trabajar sobre una metodología agile [37], la cual sólo nos ha servido en la parte de implementación.

Hemos necesitado mucho más tiempo para aprender sobre Kubernetes de lo esperado, al querer ampliar la funcionalidad de éste con el controller, hemos dedicado mucho tiempo al aprendizaje de GO [38] no previsto.

**Trabajo futuro:**
Tareas y pensamientos a tratar en un futuro:

- Alguna partes críticas de Cloudy se pueden extraer y Kubenizar, es decir, partes como Serf, podrían ser un DaemonSet en cada nodo de Kubernetes.
- Investigar en profundidad el nuevo paradigma de networking que ofrecen los *service mesh*.
- Automatizar en el arranque de Kubernetes desde Cloudy.
- Definir las políticas de usuario y aplicación. Ésta parte es realmente importante, tanto si queremos un único clúster como multi-clúster.
- Definición de la estrategia para un posible *disaster recovery* del nodo máster.
- Añadir lógica a las aplicaciones de Kubernetes desde Cloudy

**Proyectos de interés:**

- KubeEdge [39]: *Is an open source system for extending native containerized application orchestration capabilities to hosts at Edge.It is built upon kubernetes and provides fundamental infrastructure support for network, app. deployment and metadata synchronization between cloud and edge. Kubeedge is licensed under Apache 2.0. and free for personal or commercial use absolutely.*
  Our goal is to make an open platform to enable Edge computing, extending native containerized application orchestration capabilities to hosts at Edge, which built upon kubernetes an provides fundamental infrastructure support for network, app deployment and metadata synchronization between cloud and edge.
- K3s [40]: *K3s is a Certified Kubernetes distribution [41] designed for production workloads in unattended, resource-constrained, remote locations or inside IoT appliances.*
- Kind [42]: *Is a tool for running local Kubernetes clusters using Docker container "nodes".*

**Vídeos de interés:**

Este año he tenido la oportunidad de asistir a la Kubecon [43] [44] en Barcelona, dejo algunos vídeos realmente interesantes y pueden ayudar a un trabajo futuro.

- Kubeedge [1]
- SPIFFE [2]
- OPA [3] [4]
- Permission [5]
- Operators [6]
- Buenas prácticas [7]
- HELM [8]
- Secrets [9]
- Namespaces [10]

# 7. Glosario

| | |
|---|---|
| Alta Disponibilidad | Parte del diseño de la arquitectura de sistemas que nos asegura un cierto grado de protección y continuidad operacional. |
| Ansible | Plataforma para automatizar, provisionar y administrar computadoras. |
| 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. |
| APP | Abreviación de application, se trata de cualquier aplicación informática. |
| APT | Sistema de gestión de paquetes creado por Debian. |
| Autenticación | Sistema que acredita que un usuario o aplicación es quien dice ser. |
| Autodiscovery | Sistema o aplicación que mediante un protocolo puede ver el resto de sistemas de su misma red. |
| Automatización | Técnica para resolver problemas rutinarios, mejorandolos y evitando el fallo humano. |
| Autorización | Sistema que limita/permite a los usuarios y aplicaciones realizar acciones de forma controlada. |
| Autoscaling | Patrón de arquitectura de sistemas, en el cual se incrementa/decrementa máquinas dependiendo de una serie de métricas. |
| Avahi | Avahi es un protocolo libre que permite a los programas publicar y descubrir servicios y hosts que |

| | |
|---|---|
| | se ejecutan en una red local mediante el protocolo mDNS y DNS-SD. |
| AWS | Acrónimo de Amazon Web Services. Es un proveedor cloud. Actualmente el número Gartner (Enlace al cuadrante 2017). |
| Backoffice | Aplicación que nos permite gestionar y administrar procesos desde una interfaz. |
| Balanceador de carga | Se trata de un dispositivo que se pone delante de un conjunto de servidores, los cuales dan el mismo tipo de servicio, y hace llegar las peticiones de los usuarios a estos. |
| Cgroup | Abreviación de Control group, es la característica del Kernel de GNU/Linux que permite aislar los recursos(RAM, CPU, I/O, red) de nuestro servidor destinados a una aplicación. |
| Cloud Computing | Paradigma de computación el cual permite obtener recursos y servicios en cualquier momento a través de internet y normalmente un Cloud Provider. |
| Cloud Provider | Proveedor de servicios a través de internet. |
| Cloudy | Es una distribución basada en Debian GNU/Linux, destinada a los usuarios, para fomentar la transición y adopción del entorno cloud en las redes comunitarias. |
| Cluster | Conjunto de ordenadores conectados entre sí que dan servicio como si fuesen uno. |
| Community Network Cloud | Sistema de servicios que los usuarios comparten sin ánimo de lucro y dentro de una misma red que normalmente es internet. |
| Consistència | Es la capacidad en los sistemas distribuidos de que cada nodo tenga la misma información. |

| | |
|---|---|
| Container | Actualmente, es un estándar como unidad de software, la cual es autocontenida y puedo ser lanzada en cualquier entorno. |
| CRUD | Acrónimo de Create, Read, Update and Delete, se usa para referirse a las acciones básicas de un sistema de persistencia. |
| C-RIO | Nueva tecnología de containers dedicada a Kubernetes. |
| Custom metric | Métrica personalizada que crearemos desde nuestra aplicación para monitorizar normalmente una parte de negocio. |
| Daemon | Servicio del sistema que normalmente arranca y mantiene el estado de una aplicación. |
| DDOS | Del inglés Denial of Service, es un tipo de ataque informático que causa la inaccesibilidad del servicio a usuarios legítimos. |
| Debian | Es una distribución del sistema operativo GNU/Linux. |
| Digital Ocean | Al igual que AWS, es un proveedor de servicios Cloud. |
| DNS | Se refiere a Domain Name Server, servicio de nombres el cual dado un nombre de dominio puede resuelve a una IP y viceversa. |
| Docker | Tecnología para la creación y administración de containers. |
| Downtime | Tiempo en el cual una aplicación no está disponible para los usuarios legítimos. |
| GIT | Sistema de control de versiones creado por Linus Torvalds. |

| | |
|---|---|
| GO | Lenguaje de programación inventado por Google. |
| Gossip | Protocolo peer-to-peer de comunicación en sistemas distribuídos. |
| GUIFI | Es un proyecto tecnológico, social y económico impulsado desde la ciudadanía que tiene por objetivo la creación de una red de telecomunicacione abierta, libre y neutral basada en un modelo de procomún. |
| Healthcheck | Recurso de una aplicación que nos indica su estado, normalmente para comprobar que está corriendo sin problemas. |
| HELM | Gestor de paquetes para Kubernetes. |
| 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. |
| HPA | Del inglés Horizontal Pod Autoscaling, son las siglas que hacen referencia al auto escalado horizontal en Kubernetes. |
| IDE | Del inglés Integrated Development Environment. es una aplicación informática que proporciona servicios integrales para facilitarle al desarrollador o programador el desarrollo de software. |
| IP | Protocolo de la capa de red según el modelo OSI, también se refiere a la identificación de un dispositivo en la red. |
| Kubernetes/K8s | Plataforma de código abierto para automatizar la implementación, el escalado y la administración de aplicaciones en contenedores. |
| 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. |
| Map | Es un tipo abstracto de dato formado por una colección de claves únicas y una colección de valores, con una asociación uno a uno. |
| Minikube | Tecnología que permite emular un clúster de Kubernetes, se usa normalmente en entornos de desarrollo. |
| Multiclúster | Arquitectura en la que la infraestructura está replicada en varios clusters. |
| Namespace | Tipo de recurso de Kubernetes que permite aislar de forma lógica las aplicaciones. |
| Ncat | Herramienta command-line open-source que permite abrir puertos TCP/UDP en los dispositivos de la red y comunicar a través de ellos. |
| Nginx | Servidor web y proxy inverso open-source. |
| Nodo | Cada uno de los dispositivos de una red. |
| On-Premise | Servidores físicos alojados normalmente en centros de datos. |
| Open source | Modelo colaborativo en el desarrollo de software, en el cual se fomenta el uso, la modificación y distribución sin ánimo de lucro. |
| Operator | Un operador permite a los usuarios crear, configurar y administrar aplicaciones al extender la API de Kubernetes. |
| Orquestador de containers | Aplicación usada para administrar unidades software containers. |
| PaaS | Acrónimo Platform as a Service, viene a referirse a proveedor de una plataforma en la nube. |

| | |
|---|---|
| Playbook | Los tareas que se escriben de forma descriptiva y agrupadas en ficheros yaml. Este tipo de ficheros y tareas es usado por la aplicación Ansible. |
| Pod | Unidad mínima del clúster de Kubernetes formado por 1 o más containers. |
| Puerto | Interfaz por donde el resto de dispositivos se van a poder comunicar con la aplicación. |
| Pull Request | Una Pull Request es la acción de validar un código que se va a mergear de una rama a otra. En este proceso de validación pueden entrar los factores que queramos: Builds (validaciones automáticas), asignación de código a tareas, validaciones manuales por parte del equipo, despliegues, etc. |
| Rack | Estante o armario metálico en el cual se almacenan equipo informático o electrónico. |
| Raspberry | Microordenador basado en arquitectura ARM y bajo coste desarrollado por la Fundación Raspberry Pi, usado normalmente con fines educativos o particulares. |
| Resiliencia | Es la capacidad de los sistemas informáticos de mantener un estado estable a pesar de fallos y/o contratiempos operacionales. |
| REST | Es un estilo de arquitectura software para sistemas hipermedia distribuidos como la World Wide Web. |
| Restful | Hace referencia a un servicio web que implementa la arquitectura REST. |
| Rkt | Es el runtime de pods nativo en Kubernetes. |
| Rolling-update | Uno de los patrones usados para desplegar aplicaciones sin tener pérdida de servicio. |
| Runtime | Es el intervalo de tiempo en el que un programa de computadora se ejecuta en un sistema operativo. Este tiempo se inicia con la puesta en memoria |

| | |
|---|---|
| | principal del programa, por lo que el sistema operativo comienza a ejecutar sus instrucciones |
| Script | Documento que ejecuta instrucciones, escritas en un lenguaje de programación. |
| Scheduler | Proceso o procesos que se encargan de priorizar y manejar al resto de aplicaciones. |
| Serf | Herramienta de creación de Clúster, descentralizada, con detección y tolerancia a fallos y alta disponibilidad. |
| Service | Aplicación expuesta al resto de servicios en Kubernetes. Puede ser de 3 tipos: LoadBalancer, NodePort y ClusterIP |
| Service Mesh | Es una capa de control, idealmente usada en entornos arquitectura de microservicios, que ayuda y libera a las aplicaciones de la comunicación con otros servicios. |
| Shellscript | Documento que ejecuta instrucciones, escritas en un lenguaje de programación que la shell de linux puede entender y ejecuta. La shell depende de la definición del usuario. |
| Sistemas Distribuídos | Colección de computadoras separadas físicamente y conectadas entre sí por una red de comunicaciones; cada máquina posee sus componentes de hardware y software que el programador percibe como un solo sistema (no necesita saber qué cosas están en qué máquinas). |
| 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. |
| Stack | Del inglés pila, denominado también al conjunto de aplicaciones o recursos que tiene un único cometido |
| Stateless | Aplicación sin estado, es decir, no guarda los datos generados ni hace falta que el servidor retenga |

| | |
|---|---|
| | dicha información. Las peticiones a este tipo de aplicaciones deben de ser idempotentes. |
| 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. |
| Tolerancia a fallos | Es la capacidad de un sistema informático y distribuído a mantener el estado aún si alguno de los nodos fallan. |
| 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. |
| VIM | Editor de texto ligero usado también como IDE. |
| VPA | Del inglés Vertical Pod Autoscaling, son las siglas que hacen referencia al auto escalado vertical en Kubernetes. |
| Wget | Aplicación Open Source que nos permite descargarnos ficheros usando los protocolos HTTP, HTTPS, FTP y FTPS. |
| Worker | Nodo de un clúster el cual se dedica normalmente a las operaciones de lectura, este tipo de nodos tiene una consistencia eventual. |
| YUM | Es una herramienta libre de gestión de paquetes para sistemas Linux basados en RPM. |

# 8. Bibliografía


[1] FLOSS <<Software Open Source >> [Documentación en línia].
[Fecha de la consulta: 06 de Marzo de 2019]
<https://es.wikipedia.org/wiki/Software_libre_y_de_código_abierto>

[2] Documentación Amazon Web Services<< s3 >> [Documentación en línia].
[Fecha de la consulta: 06 de Marzo de 2019]
<https://aws.amazon.com/es/s3/storage-classes/>

[3] Cloudy <<What is Cloudy?>> [Documentación en línia].
[Fecha de la consulta: 9 de Marzo de 2019]
<http://Cloudy.community/what-is-Cloudy/>

[4] GPLV2 [Documentación en línia].
[Fecha de la consulta: 9 de Marzo de 2019]
<https://www.gnu.org/licenses/old-licenses/gpl-2.0.html>

[5] Serf <<Introduction to Serf>> [Documentación en línia].
[Fecha de la consulta: 10 de Marzo de 2019]
<https://www.serf.io/intro/index.html>

[6] Gossip <<Serf internals>>  [Documentación en línia].
[Fecha de la consulta: 10 de Marzo de 2019]
<https://www.serf.io/docs/internals/gossip.html>

[7] Documentación Kubernetes [Documentación en línia].
[Fecha de la consulta: 11 de Marzo de 2019]
<https://kubernetes.io/>

[8] Curso “Kubernetes para desarrolladores” impartido por José Armesto
[Codelytv]
[Fecha de inicio: 11 de Marzo de 2019]
<https://pro.codely.tv/library/kubernetes-para-desarrolladores/about/>

[9] Documentación Kubernetes “Overview of Kubectl” [Documentación en línia].
[Fecha de la consulta: 12 de Marzo de 2019]
<https://kubernetes.io/docs/reference/kubectl/overview/>

[10] Documentación Kubernetes “Running Kubernete locally”  [Documentación en línia].
[Fecha de la consulta: 14 de Marzo de 2019]
<https://kubernetes.io/docs/setup/minikube/>

[11] Documentación Kubernetes “Control Plane”  [Documentación en línia].
[Fecha de la consulta: 13 de Abril de 2019]
<https://kubernetes.io/docs/concepts/#kubernetes-control-plane>

[12] Documentación Kubernetes “Master Node” [Documentación en línia].
[Fecha de la consulta: 13 de Abril  de 2019]
<https://kubernetes.io/es/docs/concepts/#el-master-de-kubernetes>

[13] Documentación Kubernetes “Node” [Documentación en línia].
[Fecha de la consulta: 13 de Abril  de 2019]
<https://kubernetes.io/es/docs/concepts/#kubernetes-nodes>

[14] Documentación Kubernetes “Api Server” [Documentación en línia].
[Fecha de la consulta: 13 de Abril  de 2019]
<https://kubernetes.io/docs/reference/command-line-tools-reference/kube-apiserver/>

[15] Documentación Kubernetes “Controller Manager” [Documentación en línia].
[Fecha de la consulta: 13 de Abril  de 2019]
<https://kubernetes.io/docs/reference/command-line-tools-reference/kube-controller-manager/>

[16] Documentación Kubernetes “Scheduler” [Documentación en línia].
[Fecha de la consulta: 13 de Abril  de 2019]
<https://kubernetes.io/docs/reference/command-line-tools-reference/kube-scheduler/>

[17] Documentación Kubernetes “Kubeadm” [Documentación en línia].
[Fecha de la consulta:  30 de Abril  de 2019]
<https://kubernetes.io/docs/setup/independent/create-clúster-kubeadm/>

[18] Documentación Kubernetes “Kubelet” [Documentación en línia].
[Fecha de la consulta: 5 de mayo  de 2019]
<https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/>

[19] Documentación Kubernetes “Kube-proxy” [Documentación en línia].
[Fecha de la consulta: 5 de mayo de 2019]
<https://kubernetes.io/docs/reference/command-line-tools-reference/kube-proxy/>

[20]Documentación Kubernetes “Horizontal pod autoscale” [Documentación en línia].
[Fecha de la consulta: 5 de mayo de 2019]
<https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/>

[21] Documentación Kubernetes “Metrics” [Documentación en línia].
[Fecha de la consulta: 7 de mayo de 2019]
<https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/#support-for-multiple-metrics>

[22] Documentación Kubernetes “Deployment” [Documentación en línia].
[Fecha de la consulta: 7 de mayo de 2019]
<https://kubernetes.io/docs/tutorials/kubernetes-basics/update/update-intro/>

[23] Helm “What is Helm?” [Documentación en línea].
[Fecha de la consulta: 22 de mayo de 2019]
<https://helm.sh/>

[24] Cloudinitzar “How to cloudinitzar your Debian” [Documentación en línia].
[Fecha de la consulta: 5 de Abril de 2019]
<https://github.com/Clommunity/Cloudynitzar>

[25] Documentación Kubernetes “Auth systems” [Documentación en línia].
[Fecha de la consulta: 13 de mayo de 2019]
<https://kubernetes.io/docs/reference/access-authn-authz/authentication/#static-token-file>

[26] Documentación Kubernetes “RBAC” [Documentación en línia].
[Fecha de la consulta: 13 de mayo de 2019]
<https://kubernetes.io/docs/reference/access-authn-authz/rbac/>

[27] Código del operator “Serf-publisher” [Documentación en línea].
[Fecha de la consulta: 19 de Marzo de 2019]
<https://github.com/ismferd/serf-publisher>

[28] Wikipedia “CRUD” [Documentación en línea].
[Fecha de la consulta: 29 de mayo de 2019]
<https://es.wikipedia.org/wiki/CRUD>

[29] Documentación Kubernetes “How tag a manifest” [Documentación en línia].
[Fecha de la consulta: 29 de mayo de 2019]
<https://kubernetes.io/docs/concepts/configuration/assign-pod-node/#more-practical-use-cases>

[20] Kubernetes multiclúster “one or many clusters” [Documentación en línia].
[Fecha de la consulta: 21 de mayo de 2019]
<https://content.pivotal.io/blog/kubernetes-one-clúster-or-many>

[31] GIT “Control de versiones de código” [Documentación en línia].
[Fecha de la consulta: 28 de Febrero de 2019]
<https://github.com/>

[32] Repositorio de código [Documentación en línia].
[Fecha de la consulta: 28 de Febrerode 2019]
<https://github.com/ismferd/tfg-things>

[33] Documentación Ansible [Documentación en línia].
[Fecha de la consulta: 21 de Mayo de 2019]
<https://www.ansible.com/>

[34] Repositorio de código “Script de ayuda para la configuración de las raspberries” [Documentación en línia].
[Fecha de la consulta: 21 de Mayo de 2019]
<https://github.com/ismferd/tfg-things/blob/master/scripts/installer.yaml>

[35] Documentación Kubernetes “CNI” [Documentación en línia].
[Fecha de la consulta: 21 de Mayo de 2019]
<https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/network-plugins/#cni>

[36] Ncat [Documentación en línia].
[Fecha de la consulta: 1 de Junio de 2019]
<https://nmap.org/ncat/>

[37] Wikipedia “Desarrollo ágil de software” [Documentación en línia].
[Fecha de la consulta: 5 de Junio de 2019]
<https://es.wikipedia.org/wiki/Desarrollo_%C3%A1gil_de_software>

[38] Wikipedia “Golang” [Documentación en línia].
[Fecha de la consulta: 5 de Junio de 2019]
<https://es.wikipedia.org/wiki/Go_(lenguaje_de_programaci%C3%B3n)>

[39] KubeEdge “About KubeEdge” [Documentación en línia].
[Fecha de la consulta: 7 de Junio de 2019]
<https://kubeedge.io/>

[40] K3s “About k3s” [Documentación en línia].
[Fecha de la consulta: 7 de Junio de 2019]
<https://k3s.io/ >

[41] Cloud Native “k3s certification” [Documentación en línia].
[Fecha de la consulta: 7 de Junio de 2019]
<https://www.cncf.io/certification/software-conformance/>

[42] kind “Kubernetes in docker” [Documentación en línia]
[Fecha de la consulta: 7 de Junio de 2019]
<https://kind.sigs.k8s.io/>

[43] Kubecon “Kubernetes conference” [Documentación en línia].
[Fecha de la consulta: 8 de Junio de 2019]
<https://events.linuxfoundation.org/events/kubecon-cloudnativecon-europe-2019/>

[44] Youtube “Lista de reproducción Kubecon” [Documentación en línia].
[Fecha de la consulta: 8 de Junio de 2019]
<https://www.youtube.com/watch?v=4jEASYCaVDo&list=PLj6h78yzYM2PpmMAnvpvsnR4c27wJePh3>

[45] Documentación de Kubernetes “Instalación de Minikube”
[Documentación en línia].
[Fecha de la consulta: 8 de Junio de 2019]
<https://kubernetes.io/docs/tasks/tools/install-minikube/>

[46] Documentación de Kubernetes “Pod” [Documentación en línia].
[Fecha de la consulta: 8 de Junio de 2019]
<https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/>

[47] Documentación de Kubernetes “Service” [Documentación en línia].
[Fecha de la consulta: 8 de Junio de 2019]
<https://kubernetes.io/docs/concepts/services-networking/service/>

[48] Documentación de Kubernetes “Volumen” [Documentación en línia].
[Fecha de la consulta: 8 de Junio de 2019]
<https://kubernetes.io/docs/concepts/storage/volumes/>

[49] Documentación de Kubernetes “Namespace” [Documentación en línia].
[Fecha de la consulta: 9 de Junio de 2019]
<https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/>

[50] Documentación de Kubernetes “Replicaset” [Documentación en línia].
[Fecha de la consulta: 9 de Junio de 2019]
<https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/>

[51] Documentación de Kubernetes “Deployment” [Documentación en línia].
[Fecha de la consulta: 6 de Junio de 2019]
<https://kubernetes.io/docs/concepts/workloads/controllers/deployment/>

[52] Documentación de Docker “Registry” [Documentación en línia].
[Fecha de la consulta: 19 de Marzo de 2019]
<https://docs.docker.com/registry/>

[53] Documentación de Kubernetes “Taint nodes” [Documentación en línia].
[Fecha de la consulta: 1 de Junio de 2019]
<https://kubernetes.io/docs/concepts/configuration/taint-and-toleration/>

[54] Documentación de Kubernetes “Label selector” [Documentación en línia].
[Fecha de la consulta: 9 de Junio de 2019]
<https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/#label-selectors>

[55] Documentación de Kubernetes “Resource Quotas” [Documentación en línia].
[Fecha de la consulta: 6 de Junio de 2019]
<https://kubernetes.io/docs/concepts/policy/resource-quotas/>

[56] Repositorio de código “Stack heapster-influxdb-grafana” [Documentación en línia].
[Fecha de la consulta: 9 de Junio de 2019]
<https://github.com/kubernetes-retired/heapster>

# 9. Anexos

# Anexo I: Instalación de Cloudy en sistemas Debian

Dada las 3 Rasbpberries seguimos los pasos indicados en el repositorio de github:
https://github.com/Clommunity/Cloudynitzar

Rápidamente tenemos en nuestros nodos Cloudy corriendo, comprobamos que todas las raspberries tienen el puerto 7000 abierto y en estado LISTEN.

Créo una *pull request* para automatizar el proceso de instalación de Cloudy con Ansible:
https://github.com/Clommunity/Cloudynitzar/pull/17

Además, hemos realizado un pequeño *lab* donde vemos cómo funciona Serf "*under the hood*" en Cloudy y como se usa para hacer el discovery de servicios.
https://github.com/ismferd/Cloudy-serf-lab

# Anexo II: Empezando con Kubernetes (Minikube)

Kubernetes mantiene un proyecto llamado Minikube, el cual nos permite tener en nuestro entorno de desarrollo un clúster de Kubernetes.

Se trata de una máquina virtual que corre Kubernetes como *single-node* (master-worker en el mismo nodo).

Es un proyecto FOSS y podemos hacer contribuciones en el siguiente repositorio de GitHub.

### 1. Instalando Minikube

Siguiendo los pasos de la documentación oficial de Kubernetes, podemos hacer la instalación [45] sin problemas en nuestro *laptop.*

### 2. Conociendo los elementos básicos de Kubernetes

Antes de empezar, tenemos que claro el comportamiento de los siguientes elementos básicos del clúster de Kubernetes:

- Pod [46]: Es la unidad mínima que nosotros podemos desplegar, ya sea con la estrategia “1 pod → 1” contenedor o “1 pod → n contenedores”.
- Service [47]: Es la definición de cómo acceder a los pods, es decir cómo se expone un servicio.
- Volume [48]: Es el disco, donde escribirá nuestra aplicación.
- Namespace [49]: Separación lógica dentro de los nodos, la cual permite compartir recursos entre los recursos del mismo namespace.
- ReplicaSet [50]: Es el encargado de mantener estable el número de copias de nuestra aplicación.
- Deployment [51]: Este objeto se encarga de mantener el estado de los pods y los objetos replicaset.

### 3. Conociendo a Minikube

Para poder empezar a usar Minikube debemos de arrancar la máquina virtual. Como observamos en la siguiente imagen es tan fácil como lanzar el comando "`#minikube start`".


Comprobamos que tenemos el clúster levantado haciendo una simple llamada a la API:

```
o → kubectl1.6 get pod
No resources found.
```

**Nota:** Minikube por defecto nos crea un contexto en nuestro fichero de configuración situado por defecto en "`#~/.kube/config`". Además, nos cambia al contexto de Minikube para poder usarlo sin tener que hacer cambios de configuración.

Para lanzar nuestro primer pod, hemos creado una imagen docker "hello-world"y subido tanto a nuestro repositorio de GIT como a nuestro *registry* [52] de docker.

En este enlace se encuentra el código de la imagen: hello-k8s
En este enlace se encuentra la imágen: hello-k8s

En las siguientes dos imágenes vemos cómo el *generator* ha creado el objeto deployment y la comprobación de que el pod está *up&running*.

```
○ → kubectl1.6 run hello-k8s --image=docker.io/ismaelfm/hello-k8s --port=8080
deployment "hello-k8s" created
```

```
○ → kubectl1.6 get pods
NAME                         READY     STATUS    RESTARTS   AGE
hello-k8s-3112637362-54w5d   1/1       Running   0          1m
```

Pero que ha creado realmente el generator?

Con el generator hemos creado los siguientes *resources*:

- Objeto deployment (el cual crea un objeto replicaset y el pod visto previamente).

```
○ → kubectl1.6 get deployment
NAME        DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
hello-k8s   1         1         1            1           1d
```

- Objeto ReplicaSet, como sabemos es el encargado de mantener nuestro estado deseado, es decir, si hiciesemos *delete* del pod el objeto ReplicaSet levantaría automáticamente una réplica de nuestra aplicación.

```
○ → kubectl1.6 get rs hello-k8s-3112637362
NAME                   DESIRED   CURRENT   READY   AGE
hello-k8s-3112637362   1         1         1       4d
```

Ahora que tenemos nuestra aplicación levantada y corriendo, necesitamos acceder a ella.

Para ello debemos de crear el objeto *Service* del cual existen 3 tipos:

- LoadBalancer: Nos creará un *LoadBalancer* y añadirá nuestro servicio como *backend*. Este servicio será accesible desde fuera del clúster.
- NodePort: Nos creará un *Service* que al igual que el loadbalancer será accesible desde fuera del clúster mediante un *socket* {“IP”:”PORT”}.
- ClusterIP: Es el *Service* que crea por defecto y sólo es accesible desde dentro del clúster.

En la imagen vemos la creación de un *Service* (ClusterIP) y la comprobación de que se ha creado correctamente.

```
o → kubectl1.6 expose pod/hello-k8s-3112637362-54w5d --port=8080
service "hello-k8s-3112637362-54w5d" exposed
```

```
o → kubectl1.6 get service
NAME                          CLUSTER-IP    EXTERNAL-IP   PORT(S)    AGE
hello-k8s-3112637362-54w5d    10.0.0.58     <none>        8080/TCP   4m
```

Con ésta configuración las peticiones que hacemos nos dan 404.

Para cambiar este comportamiento se le puede pasar el flag –type=”el tipo de *Service* que queremos” al generator:

```
o → kubectl1.6 expose pod/hello-k8s-3112637362-54w5d --port=8080 --type=NodePort
service "hello-k8s-3112637362-54w5d" exposed
```

Como observamos en la imágen siguiente y a diferencia de la anterior, NodePort hace bind del puerto donde corre nuestra aplicación al un puerto del rang {30000-32767}.

```
o → kubectl1.6 get service
NAME                          CLUSTER-IP    EXTERNAL-IP   PORT(S)          AGE
hello-k8s-3112637362-54w5d    10.0.0.144    <nodes>       8080:31813/TCP   4m
```

Se averigua la IP del clúster, todas las peticiones que hagamos a CLUSTERIP:31813 llegarán a nuestra aplicación.

```
o → minikube ip
There is a newer version of minikube available (v1.0.0).  Download it here:
https://github.com/kubernetes/minikube/releases/tag/v1.0.0

To disable this notification, run the following:
minikube config set WantUpdateNotification false
192.168.99.100
```

```
o → curl 192.168.99.100:31813
Hello World !
```

**NOTA:** El tipo de *Service* llamado LoadBalancer no lo vamos a usar en ésta práctica ya que levanta y expone un balanceador en un cloud provider, infraestructura que no disponemos.

# ANEXO III: Config File

El fichero que nos permite acceder al clúster mediante nuestras credenciales se aloja por defecto en “`~/.kube/config`”

Si tenemos diferentes clusters y usuarios, podemos hacer uso de los contextos. Con el comando “`kubectl config view`” veremos la descripción de este fichero:

```
○ → kubectl1.6 config view
apiVersion: v1
clusters:
- cluster:
    server: https://192.168.1.40:6443
  name: cluster-token
- cluster:
    certificate-authority: /Users/ismaelfernandez/.minikube/ca.crt
    server: https://192.168.99.100:8443
  name: minikube
contexts:
- context:
    cluster: cluster-token
    user: ismael
  name: context-token
- context:
    cluster: minikube
    user: minikube
  name: minikube
current-context: minikube
kind: Config
preferences: {}
users:
- name: ismael
  user:
    token: TokenOfTheJ0n
- name: minikube
  user:
    client-certificate: /Users/ismaelfernandez/.minikube/apiserver.crt
    client-key: /Users/ismaelfernandez/.minikube/apiserver.key
- name: user-token
  user:
    token: TokenOfTheJ0n
```

En nuestro caso tenemos dos clusters, dos usuarios uno por clúster y la relación entre el clúster y el usuario se realiza mediante el contexto.
Si nos fijamos, este comando nos dice cual es el contexto actual activo.

# ANEXO IV: Setup de nuestros nodos en la red local (microCloud)

Hacemos el setup tanto de Cloudy como de Kubernetes usando los scripts y playbooks montados con Ansible para la automatización.

El objetivo del setup es el siguiente:

- Cada nodo debe de tener ip fija.
- No deben de haber hosts con el mismo *hostname.*

Antes de empezar a montar el clúster, debemos de hacer un pequeño setup en las raspberries.

Se inicializa sshd por defecto y se habilita para que esté disponible al *restart*.
Los comandos son:

- `#systemctl start ssh.service`
- `#systemctl enable ssh.service` (para tenerlo habilitado en el restart del nodo)

```
pi@raspberrypi:~ $ sudo systemctl enable ssh.service
Synchronizing state of ssh.service with SysV service script with /lib/systemd/systemd-sysv-install.
Executing: /lib/systemd/systemd-sysv-install enable ssh
Created symlink /etc/systemd/system/sshd.service → /lib/systemd/system/ssh.service.
pi@raspberrypi:~ $ systemctl status ssh
● ssh.service - OpenBSD Secure Shell server
   Loaded: loaded (/lib/systemd/system/ssh.service; enabled; vendor preset: enabled)
   Active: active (running) since Mon 2019-06-03 10:25:42 BST; 2min 33s ago
 Main PID: 816 (sshd)
   CGroup: /system.slice/ssh.service
           └─816 /usr/sbin/sshd -D

Jun 03 10:25:42 raspberrypi systemd[1]: Starting OpenBSD Secure Shell server...
Jun 03 10:25:42 raspberrypi sshd[816]: Server listening on 0.0.0.0 port 22.
Jun 03 10:25:42 raspberrypi sshd[816]: Server listening on :: port 22.
Jun 03 10:25:42 raspberrypi systemd[1]: Started OpenBSD Secure Shell server.
Jun 03 10:26:18 raspberrypi sshd[830]: Connection closed by 192.168.1.130 port 55033 [preauth]
Jun 03 10:26:41 raspberrypi sshd[832]: Connection closed by 192.168.1.130 port 55034 [preauth]
Jun 03 10:26:56 raspberrypi sshd[834]: Accepted password for pi from 192.168.1.130 port 55035 ssh2
Jun 03 10:26:56 raspberrypi sshd[834]: pam_unix(sshd:session): session opened for user pi by (uid=0)
```

Lanzamos el playbook [hosting](#) pasandole 3 parámetros:

- hostname: Nombre de nuestro nodo.
- IP: IP fija de nuestro nodo.
- DNS: Es la puerta de enlace

```
2019-06-03 11:50:03  MacBook-Pro-de-Ismael-2 in ~/ismferdev/tfg-things/scripts
± |master U:3 ?:2 ✗| → ansible-playbook -i host hosting.yaml -u pi --ask-pass --extra-vars "hostname=worker-2 ip=192.168.1.42 dns=192.168.1.1"
```

Éste script llamará al shell script que se encargará de modificar los ficheros:

- `/etc/dhcpcd.conf`
- `/etc/hosts`

```
± |master U:3 ?:2 ✗| → cat hosting.yaml
- name: Transfer and execute a script.
  hosts: raspberry
  vars:
    hostname: "{{ hostname }}"
    ip: "{{ ip }}"
    dns: "{{ dns }}"
  remote_user: pi
  sudo: yes
  tasks:
     - name: Transfer the script
       copy: src=hosting.sh dest=/tmp/ mode=0777

     - name: Execute the script
       command: sh /tmp/hosting.sh {{hostname}} {{ip}} {{dns}}
```

```
± |master U:3 ?:3 ✗| → cat hosting.sh
#!/bin/sh

hostname=$1
ip=$2 # should be of format: 192.168.1.100
dns=$3 # should be of format: 192.168.1.1

# Change the hostname
sudo hostnamectl --transient set-hostname $hostname
sudo hostnamectl --static set-hostname $hostname
sudo hostnamectl --pretty set-hostname $hostname
sudo sed -i s/raspberrypi/$hostname/g /etc/hosts

# Set the static ip

sudo cat <<EOT >> /etc/dhcpcd.conf
interface eth0
static ip_address=$ip/24
static routers=$dns
static domain_name_servers=$dns
EOT
```

Revisamos que los cambios son los deseados; tanto de nuestra dirección de loopback, como nuestra IP fija.

```
pi@worker-2:~ $ sudo cat /etc/hosts
127.0.0.1       localhost
::1             localhost ip6-localhost ip6-loopback
ff02::1         ip6-allnodes
ff02::2         ip6-allrouters

127.0.1.1       worker-2
```

```
pi@worker-2:~ $ sudo grep "interface eth0" /etc/dhcpcd.conf  -A 10
#interface eth0
#static ip_address=192.168.0.10/24
#static ip6_address=fd51:42f8:caae:d92e::ff/64
#static routers=192.168.0.1
#static domain_name_servers=192.168.0.1 8.8.8.8 fd51:42f8:caae:d92e::1

# It is possible to fall back to a static IP if DHCP fails:
# define static profile
#profile static_eth0
#static ip_address=192.168.1.23/24
#static routers=192.168.1.1
--
#interface eth0
#fallback static_eth0

interface eth0
static ip_address=192.168.1.42/24
static routers=192.168.1.1
static domain_name_servers=192.168.1.1 8.8.8.8
```

Repetimos el proceso para el resto de Raspberries.

# ANEXO V: Buenas prácticas en Kubernetes

1. No compartir el fichero *manifest* generado por el Nodo Master. Este fichero nos permite acceder como administrador al clúster.
2. Crear namespaces. Debemos de dividir, separar y aislar nuestras aplicaciones y/o entornos de forma lógica.
3. *Untaint* [53] master node. En los entornos *edge* en los que nos movemos es bueno marcar los master cómo *Untaint*, de esta forma podremos usar los recurso disponibles en éste nodo.
4. Labels [54]: Los labels son atributos de los metadatos que nos permiten identificar y agrupar a las aplicaciones.
5. Usar e instalar kubens y kubectx. Con estas dos herramientas podemos hacer cambios de contexto y namespaces de una forma muy sencilla. En su repositorio de git tenemos las instrucciones para su uso e instalación.
6. Habilitar el auto-completado de kubectl.
7. Tener diferentes contextos en el fichero "`~/.kube/config`"
8. En el caso de los sistemas *edge* es importante limitar los de recursos de nuestras aplicaciones haciendo uso de las quotas, además, se disponemos del objeto ResourceQuota [55]. Este recurso creará límites para todos los pods corriendo en un mismo namespaces.

# ANEXO VI: Troubleshooting

**1.Preflight da los siguientes errores:**

```
error execution phase preflight: [preflight] Some fatal errors occurred:
  [ERROR IsDockerSystemdCheck]: cgroup driver is not defined in 'docker info'
  [ERROR Swap]: running with swap on is not supported. Please disable swap
  [ERROR SystemVerification]: unsupported docker version:
  [ERROR SystemVerification]: missing cgroups: memory
```

**Solución:**

- Añadimos a la la configuración que arranca kubeadm los siguientes parámetros:

```
KUBELET_ARGS="--kubeconfig=/etc/kubernetes/kubeadminconfig\
--require-kubeconfig=true \
--pod-manifest-path=/etc/kubernetes/manifests \
--cgroup-driver=systemd"
```

- Después lanzar el comando swapoff -a (el que deshabilita la memoria swap).
- Verificar tu versión Docker, si instalamos la versión desde Cloudy nos sacará el error. Si instalamos con la última versión disponible de docker-ce para ARM no tendremos problemas.
- Añadir al fichero `/boot/cmdline.txt`:
  `cgroup_enable=cpuset`
  `cgroup_enable=memory`
- Puedes ser que recibamos un error más relacionado con el cgroups pids, para solucionarlo hay que montarlo con el comando:
  `# groups-mount pids`

### 2. El plugin CNI no arranca y los pods COREDNS tampoco.

El problema reside en la red que el CNI montará.

**Solución:**

Para solucionar el problema se le debe indicar el CIDR que el CNI debe de usar como argumento a kubeadm:

```
#sudo kubeadm init \
--kubernetes-version 1.14.1 \
--pod-network-cidr 192.168.1.0/24 | \
tee kubeadm-init.out
```

### 3. Los pods no arrancan y docker da el siguiente STERR standard_init_linux.go:207: exec user process caused "exec format error"

El problema es que o bien el container usado en el FROM de la imagen docker o bien la aplicación no está compilada para ARM.

```
pi@raspberrypi:~ $ docker run ismaelfm/serf-publisher-arm:0.1.0
standard_init_linux.go:207: exec user process caused "exec format error"
```

**Solución:**
Compilar las aplicaciones de forma cross arquitectura o para arquitecturas ARM, usar las imágenes base (FROM) armv* donde el asterisco es la versión de tu arquitectura. En mi caso armv7.

### 4. Al arrancar Kubernetes da error de driver en los cgroups.

```
pi@worker-2:~ $ sudo kubeadm join 192.168.1.40:6443 --token km90l7.eezaswpygljmcnxb     --discovery-token-ca-cert-hash sha256:a11f6625cdb99a230c2235f07ea7f8a2402d913a76389a34f3bbbb3af029adb9
[preflight] Running pre-flight checks
        [WARNING IsDockerSystemdCheck]: detected "cgroupfs" as the Docker cgroup driver. The recommended driver is "systemd". Please follow the guide at https://kubernetes.io/docs/setup/cri/
```

**Solución:**
Indicar al daemon de docker que arranque como parte de systemd.
Siguiendo ésta guía se solventa el warning.

# ANEXO VII: CÓDIGO

**main.go**

```
package main
import "github.com/ismferd/serf-publisher/cmd"
func main() {
  cmd.Execute()
}
```

**root.go**

```
package cmd
import (
  "fmt"
  "os"
  "path/filepath"
  "os/signal"
  "syscall"
  "time"
  "k8s.io/client-go/util/homedir"
  "github.com/ismferd/serf-publisher/pkg"
  "github.com/spf13/cobra"
  "github.com/spf13/viper"
  "github.com/spotahome/kooper/operator/controller"
  "go.uber.org/zap"
  "k8s.io/client-go/kubernetes"
  "k8s.io/client-go/rest"
  "k8s.io/client-go/tools/clientcmd"
)
```

```
var (
  zapLogger, _ = zap.NewProduction(zap.AddCallerSkip(1))
  logger       = pkg.NewLogger(*zapLogger.Sugar())
  stopC        = make(chan struct{})
  finishC      = make(chan error)
  signalC      = make(chan os.Signal, 1)
)

// rootCmd represents the base command when called without any subcommands
var rootCmd = &cobra.Command{
  Use:   "serf-publisher",
  Short: "Kubernetes controller that automatically adds annotations in Pods to they can assume AWS Roles.",
  Long:  `Kubernetes controller that automatically adds annotations in Pods to they can assume AWS Roles.`,
  Run: func(cmd *cobra.Command, args []string) {
    defer zapLogger.Sync()
    signal.Notify(signalC, syscall.SIGTERM, syscall.SIGINT)

    k8sCli, err := getKubernetesClient(viper.GetString("kubeconfig"), logger)
    if err != nil {
      logger.Errorf("Can't create k8s client: %s", err)
      os.Exit(1)
    }
```

```
    go func() {
      finishC <- getController(k8sCli, logger).Run(stopC)
    }()
    select {
    case err := <-finishC:
      if err != nil {
        logger.Errorf("error running controller: %s", err)
        os.Exit(1)
      }
    case <-signalC:
      logger.Info("Signal captured, exiting...")
    }
  },
}
// Execute adds all child commands to the root command and sets flags appropriately.
// This is called by main.main(). It only needs to happen once to the rootCmd.
func Execute() {
  if err := rootCmd.Execute(); err != nil {
    fmt.Println(err)
    os.Exit(1)
  }
}
func init() {
  cobra.OnInitialize(func() {
    namespace := viper.GetString("namespace")
    if len(namespace) == 0 {
      logger.Error("Error: required flag \"namespace\" or environment variable
\"NAMESPACE\" not set")
      os.Exit(1)
    }
```

```
  })
  rootCmd.Flags().String("kubeconfig", filepath.Join(homedir.HomeDir(), ".kube", "config"),
"Path to the kubeconfig file")
  _ = viper.BindPFlag("kubeconfig", rootCmd.Flags().Lookup("kubeconfig"))

  rootCmd.Flags().String("namespace", "", "kubernetes namespace where this app is
running")
  _ = viper.BindPFlag("namespace", rootCmd.Flags().Lookup("namespace"))

  rootCmd.Flags().Int("resync-seconds", 30, "The number of seconds the controller will
resync the resources")
  _ = viper.BindPFlag("resync-seconds", rootCmd.Flags().Lookup("resync-seconds"))

  viper.AutomaticEnv()
}

func getKubernetesClient(kubeconfig string, logger pkg.Logger) (kubernetes.Interface, error)
{
  var err error
  var cfg *rest.Config

  cfg, err = rest.InClusterConfig()
  if err != nil {
    logger.Warningf("Falling back to using kubeconfig file: %s", err)
    cfg, err = clientcmd.BuildConfigFromFlags("", kubeconfig)
    if err != nil {
      return nil, err
    }
  }

  return kubernetes.NewForConfig(cfg)
```

```
}

func getController(k8sCli kubernetes.Interface, logger pkg.Logger) controller.Controller {
  return controller.NewSequential(
    time.Duration(viper.GetInt("resync-seconds"))*time.Second,
    pkg.NewHandler(pkg.NewSerfPublisher(k8sCli, logger)),
    pkg.NewDeploymentRetrieve(viper.GetString("namespace"), k8sCli),
    nil,
    logger,
  )
}
```

**retrieve.go**

```
package pkg
import (
  v1 "k8s.io/api/core/v1"
  metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
  "k8s.io/apimachinery/pkg/runtime"
  "k8s.io/apimachinery/pkg/watch"
  "k8s.io/client-go/kubernetes"
  "k8s.io/client-go/tools/cache"
)

// DeploymentRetrieve knows how to retrieve Deployments.
type DeploymentRetrieve struct {
  namespace string
  client    kubernetes.Interface
}

// NewDeploymentRetrieve returns a new Deployment retriever.
```

```
func NewDeploymentRetrieve(namespace string, client kubernetes.Interface)
*DeploymentRetrieve {
  return &DeploymentRetrieve{
    namespace: namespace,
    client:   client,
  }
}

// GetListerWatcher knows how to return a listerWatcher of a Deployment.
func (p *DeploymentRetrieve) GetListerWatcher() cache.ListerWatcher {

  return &cache.ListWatch{
    ListFunc: func(options metav1.ListOptions) (runtime.Object, error) {
      return p.client.CoreV1().Services(p.namespace).List(options)
    },
    WatchFunc: func(options metav1.ListOptions) (watch.Interface, error) {
      return p.client.CoreV1().Services(p.namespace).Watch(options)
    },
  }
}

// GetObject returns an empty Deployment.
func (p *DeploymentRetrieve) GetObject() runtime.Object {
  return &v1.Service{}
}
```

logger.go

```
package pkg
import (
  "go.uber.org/zap"
)

// Logger is the interface that the loggers used by the library will use.
type Logger interface {
  Info(format string)
  Warning(format string)
  Error(format string)
  Infof(format string, args ...interface{})
  Warningf(format string, args ...interface{})
  Errorf(format string, args ...interface{})
}

// ZapLogger is a logger using the zap logger underneath.
type ZapLogger struct {
  logger zap.SugaredLogger
}

// Info logging with INFO level.
func (l *ZapLogger) Info(format string) {
  l.logger.Info(format)
}

// Debug logging with DEBUG level.
func (l *ZapLogger) Debug(format string) {
  l.logger.Debug(format)
```

```
}

// Warning logging with WARNING level.
func (l *ZapLogger) Warning(format string) {
  l.logger.Warn(format)
}

// Error logging with ERROR level.
func (l *ZapLogger) Error(format string) {
  l.logger.Error(format)
}

// Infof logging with INFO level.
func (l *ZapLogger) Infof(format string, args ...interface{}) {
  l.logger.Infof(format, args...)
}

// Debugf logging with DEBUG level.
func (l *ZapLogger) Debugf(format string, args ...interface{}) {
  l.logger.Debugf(format, args...)
}

// Warningf logging with WARNING level.
func (l *ZapLogger) Warningf(format string, args ...interface{}) {
  l.logger.Warnf(format, args...)
}

// Errorf logging with ERROR level.
func (l *ZapLogger) Errorf(format string, args ...interface{}) {
  l.logger.Errorf(format, args...)
}
```

```
// NewLogger returns a new Logger using the zap logger underneath
func NewLogger(logger zap.SugaredLogger) *ZapLogger {
  return &ZapLogger{
     logger: logger,
  }
}

// NewDummyLogger returns a new Logger that doesn't log anything
func NewDummyLogger() *DummyLogger {
  return &DummyLogger{}
}

// DummyLogger is a logger that logs nothing
type DummyLogger struct {
}

// Info logging with INFO level.
func (d *DummyLogger) Info(format string) {}

// Debug logging with DEBUG level.
func (d *DummyLogger) Debug(format string) {}

// Warning logging with WARNING level.
func (d *DummyLogger) Warning(format string) {}

// Error logging with ERROR level.
func (d *DummyLogger) Error(format string) {}

// Infof logging with INFO level.
func (d *DummyLogger) Infof(format string, args ...interface{}) {}
```

```
// Debugf logging with DEBUG level.
func (d *DummyLogger) Debugf(format string, args ...interface{}) {}

// Warningf logging with WARNING level.
func (d *DummyLogger) Warningf(format string, args ...interface{}) {}

// Errorf logging with ERROR level.
func (d *DummyLogger) Errorf(format string, args ...interface{}) {}
```

### handler.go

```
package pkg
import (
  v1 "k8s.io/api/core/v1"
  "k8s.io/apimachinery/pkg/runtime"
)

// Handler will receive objects whenever they get added or deleted from the k8s API.
type Handler struct {
  serfPublisher SerfPublisherInterface
}

// Add is called when a k8s object is created.
func (h *Handler) Add(obj runtime.Object) error {
  _, err := h.serfPublisher.Publish(*obj.(*v1.Service))
  return err
}

// Delete is called when a k8s object is deleted.
```

```
func (h *Handler) Delete(s string) error {
  return nil
}

// NewHandler returns a new Handler to handle Deployments created/updated/deleted.
func NewHandler(serfPublisher SerfPublisherInterface) *Handler {
  return &Handler{
    serfPublisher: serfPublisher,
  }
}
```

**service.go**

```
package pkg

import (
  v1 "k8s.io/api/core/v1"
  "k8s.io/client-go/kubernetes"
  "os/exec"
 "strconv"
)

// SerfPublisherInterface for the SerfPublisher
type SerfPublisherInterface interface {
  Publish(service v1.Service) (v1.Service, error)
}

// SerfPublisher is simple annotator service.
type SerfPublisher struct {
  client kubernetes.Interface
```

```
  logger Logger
}

// NewSerfPublisher returns a new SerfPublisher.
func NewSerfPublisher(k8sCli kubernetes.Interface, logger Logger) *SerfPublisher {
  return &SerfPublisher{
    client: k8sCli,
    logger: logger,
  }
}

// Publish will add a new service trhought serf
func (s *SerfPublisher) Publish(service v1.Service) (v1.Service, error) {
  newService := service.DeepCopy()
  cmd := exec.Command("/usr/sbin/avahi-ps", "publish", newService.ObjectMeta.Name,
"kubernetes", strconv.Itoa(int(newService.Spec.Ports[0].NodePort)),
string(newService.Spec.Type))
  out, err := cmd.CombinedOutput()
  if err != nil {
    s.logger.Infof("cmd.Run() failed with %s\n", err)
  }
  s.logger.Infof("command \n%s\n", out)
  return *newService, nil
}
```

# ANEXO VIII: Contribuciones Open Source

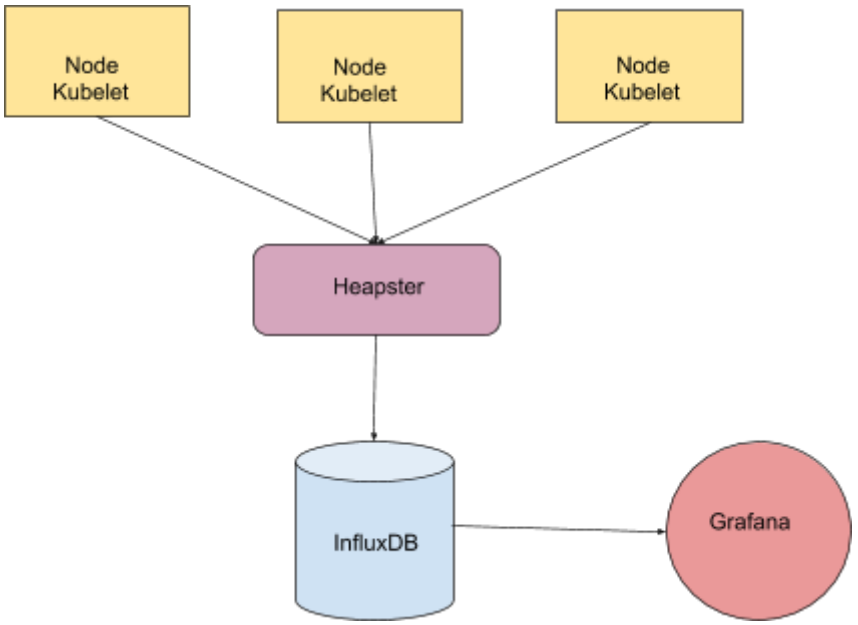
Para llevar a cabo el proyecto hemos contribuido al proyecto Cloudy y generado código open source.

A continuación mostramos las contribuciones:
Lab Cloudy-Serf:
https://github.com/ismferd/cloudy-serf-lab
Queremos que Cloudy se instale en ‘n’ hosts y con un solo comando:
https://github.com/Clommunity/cloudynitzar/pull/17
La versión de Docker nos da problemas con Kubernetes:
https://github.com/Clommunity/cloudynitzar/issues/18
Instalación de paquetes para `Ubuntu - Bionic:`
https://github.com/Clommunity/cloudynitzar/pull/15
Scripts para instalar y configurar nuestros nodos:
https://github.com/ismferd/tfg-things
Operator de Kubernetes:
https://github.com/ismferd/serf-publisher
Dockerizando Serf:
https://github.com/hashicorp/serf/pull/558

# ANEXO XI: Métricas en Kubernetes

Para monitorizar y el envío de métricas hemos usado el *stack* Heapster-InfluxDB-Grafana [56].

Imagen 17: Stack Heapster-InfluxDB-Grafana




Heapster es el recolector de métricas (se debe de desplegar como *daemonset* ya que recogerá datos de los nodos de Kubernetes)
InfluxDB es el *Backend*, la Base de datos donde persistiremos nuestras métricas.
Grafana, hará la explotación de los datos mostrando las métricas mediante *dashboards*.

El stack es muy sencillo de desplegar, nos bajamos los *manifests* oficiales mediante wget:

```
wget https://raw.githubusercontent.com/kubernetes/heapster/master/deploy/kube-config/influxdb/grafana.yaml
wget https://raw.githubusercontent.com/kubernetes/heapster/master/deploy/kube-config/influxdb/heapster.yaml
wget https://raw.githubusercontent.com/kubernetes/heapster/master/deploy/kube-config/influxdb/influxdb.yaml
wget https://raw.githubusercontent.com/kubernetes/heapster/master/deploy/kube-config/rbac/heapster-rbac.yaml
```

Realizamos las modificaciones para que no nos de problemas con la arquitectura ARM:

```
containers:
- name: heapster
  image: k8s.gcr.io/heapster-arm:v1.4.2
  imagePullPolicy: IfNotPresent
  command:
  - /heapster
  - --source=kubernetes:https://kubernetes.default?kubeletHttps=true&kubeletPort=10250&insecure=true
  - --sink=influxdb:http://monitoring-influxdb:8086
```

```
containers:
- name: influxdb
  image: gcr.io/google_containers/heapster-influxdb-arm:v1.1.1
  volumeMounts:
  - mountPath: /data
    name: influxdb-storage
volumes:
- name: influxdb-storage
  emptyDir: {}
```

```
containers:
- name: grafana
  image: k8s.gcr.io/heapster-grafana-arm:v5.0.4
  ports:
  - containerPort: 3000
    protocol: TCP
  volumeMounts:
  - mountPath: /etc/ssl/certs
    name: ca-certificates
    readOnly: true
  - mountPath: /var
    name: grafana-storage
```

Guardamos los cambios y antes de crear aplicar los manifest creamos un ClusterRole y un ServiceAccount para que Heapster pueda recoger las métricas de los recursos.

Aplicamos todos los ficheros manifests y enseguida vemos métricas.

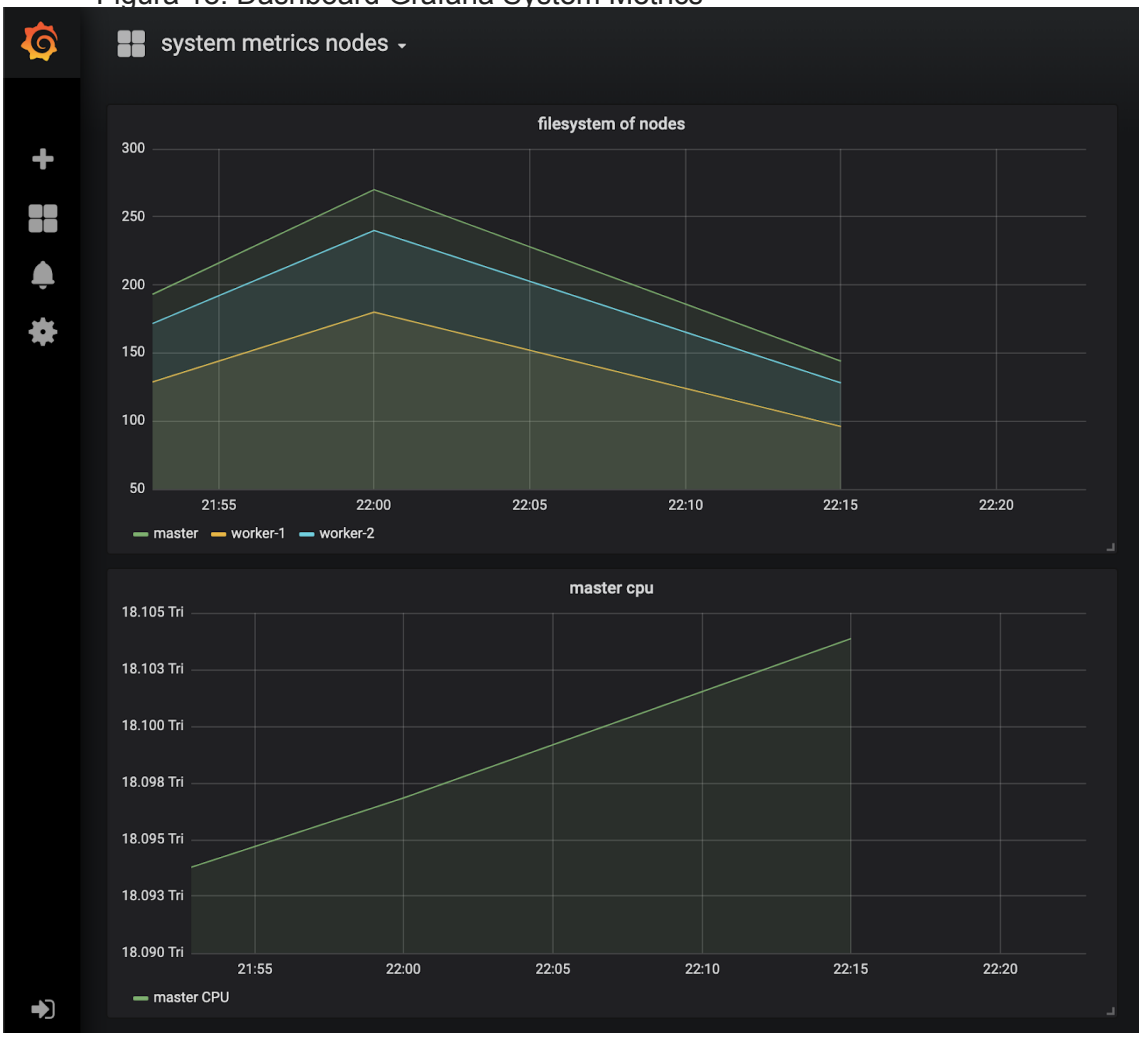
```
pi@raspberry-1:~/kubernetes/monitoring $ kubectl top node
NAME          CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
raspberry-1   567m         14%    651Mi           78%
raspberrypi   224m         5%     495Mi           59%
worker-2      297m         7%     561Mi           67%
```

Miramos los *Services* y filtramos por NodePort para ver en qué puerto está expuesto Grafana.

```
pi@raspberry-1:~/kubernetes/monitoring $ kubectl get services -n kube-system|grep -i NodePort
monitoring-grafana      NodePort    10.105.186.74    <none>        80:30629/TCP            21h
```

Nos creamos los dashboards deseados, en nuestro caso tenemos uno con la CPU y el filesystem.

Figura 18: Dashboard Grafana System Metrics




En éste repositorio de git encontramos todas la información al respecto: https://github.com/kubernetes-retired/heapster