UNIVERSITAT POLITÈCNICA DE VALÈNCIA

CAMPUS D'ALCOI

DSIC

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Cloud computing

Laboratorio 3

Contenido

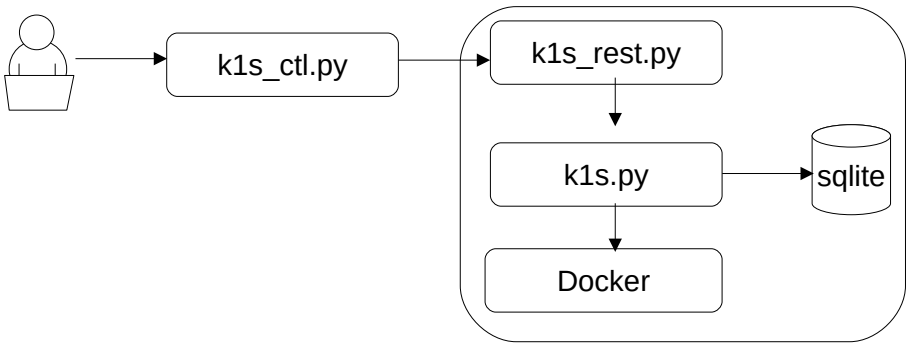
1. Introducción

En este laboratorio se implementará un Kubernetes (K8s) limitado a partir de Docker, lo denominaremos K1s. Este sistema se ejecutará en una única máquina y no en un clúster de máquinas.

Como ya sabemos, Docker es una herramienta que permite ejecutar contenedores a partir de imágenes. Sin embargo, carece de mecanismos de orquestación avanzados como los que proporciona Kubernetes. En este laboratorio, exploraremos cómo podemos implementar algunos de estos mecanismos.

2. Arquitectura del sistema

A continuación se presenta un diagrama general del sistema.



Como se puede observar, K1s recubre una instalación local de Docker y publica una API RESTful, que será consumida por la herramienta CLI k1s_ctl. Además, K1s poseerá una base de datos local (SQLite) en la que almacenará toda la información necesaria para que el sistema funcione correctamente.

3. K1s

Partiendo de Docker, K1s implementará una versión limitada de los siguientes recursos:

- Pods
- ReplicaSets
- HorizontalPodAutoscalers

Para gestionar estos recursos, el módulo k1s.py publicará al menos las siguientes operaciones:

```
def addPod(pod): pass
def removePod(name): pass
def getPod(name): pass
def listPods(labels={}): pass

def addReplicaSet(rs): pass
def removeReplicaSet(name): pass
def getReplicaSet(name): pass
def listReplicaSets(labels={}): pass

def addHorizontalPodAutoscaler(hpa) pass
def removeHorizontalPodAutoscaler(name): pass
def getHorizontalPodAutoscaler(name): pass
def listHorizontalPodAutoscalers(labels={}): pass
```

Además, al arrancar K1s deberá efectuar ciertas tareas de inicialización, como por ejemplo crear la base de datos. Para ello, definiremos la función *init()*, por ejemplo:

```
import sqlite3

FILE_DB = "k1s.db"

def init():
  print("init()")

  # db init
  con = sqlite3.connect(FILE_DB)
  cur = con.cursor()
  try:
    cur.execute("create table rs(name varchar(32), labels text, spec text)")
    cur.execute("create table hpa(name varchar(32), labels text, spec text)")
  except:
    print("Database exists.")
  con.commit()
  con.close()

  # [TODO] Add more init code
  ...

init()
```

K1s debe ser capaz de recuperarse ante caídas. Esto significa que si lo paramos (o falla) y lo volvemos a arrancar, debería de ser capaz de alcanzar el mismo estado que tenía antes de parar (o fallar). Ello implicará añadir más operaciones a la función *init()*, como veremos más adelante.

3.1 Pods

K1s será capaz de gestionar Pods. Para crear un Pod, será necesario suministrar una especificación compatible con K8s. Por ejemplo, K1s deberá ser capaz de aplicar la siguiente especificación:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    app: hello
    version: "1.0"
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
      protocol: TCP
```

Para simplificar, asumiremos que un Pod contiene un único contenedor. Si en la especificación aparecen más de uno, entonces únicamente se considerará el primero.

Para dialogar con Docker, utilizaremos la librería “Docker SDK for Python” que ya hemos utilizado con anterioridad en la asignatura:

https://docker-py.readthedocs.io/en/stable/

```
> python3 -m venv venv
> source venv/bin/activate
(venv) > pip install docker
```

Recordamos rápidamente algunas operaciones con esta librería:

```
import docker
client = docker.from_env()
client.containers.run("nginx", detach=True)
for c in client.containers.list(all=True): print(c)
c = client.containers.get(name)
c.id, c.short_id, c.image, c.name, c.status, c.attrs, c.stats(stream=False)
c.start(), c.stop(), c.remove()
```

Gracias a estas operaciones, cuando creamos Pods en K1s no será necesario registrarlos en nuestra base de datos, ya que podremos obtener esta información en tiempo real directamente desde Docker.

En K1s los Pods serán identificados por su **nombre**. Y podremos agruparlos con **etiquetas**. Afortunadamente, en Docker podemos asignar nombres y etiquetas a los contenedores, de manera que el mapeo Pod/contenedor es directo.

Además de las etiquetas especificadas en “metadata”, recomendamos añadir una etiqueta adicional para distinguir los contenedores creados desde K1s de los contenedores creados desde fuera. Por ejemplo, podemos añadir “k1s=pod”.

```
container = client.containers.run(
  image="nginx",
  name="nginx",
  labels={"k1s":"pod"}
)
```

A continuación se muestra un posible esqueleto de la función *addPod(pod)*:

```
import json
def addPod(pod):
  print(f"addPod({json.dumps(pod)})")

  # 1. Check spec
  if "apiVersion" not in pod: raise Exception("apiVersion field missing")
  if pod["apiVersion"] != "v1": raise Exception("invalid apiVersion")
  if "kind" not in pod: raise Exception("kind field missing")
  if pod["kind"] != "Pod": raise Exception("invalid kind")

  # [TODO] check all the fields are correct!!!
  ...

  # 2. Check the Pod does not exist yet (by name)
  containers = client.containers.list(filters={"name": pod["metadata"]["name"]})
  if len(containers) > 0: raise Exception("Pod already exists")

  # 3. Add dummy label
  labels = {"k1s": "pod"}
  if "labels" in pod["metadata"]:
    for lbl in pod["metadata"]["labels"]:
      labels[lbl] = pod["metadata"]["labels"][lbl]
```

```
  # 4. Create container
  container = client.containers.run(
        image=pod["spec"]["containers"][0]["image"],
        labels=labels,
        name=pod["metadata"]["name"],
        detach=True
  )

  # 5. Return simple data about Pod
  return {
      "name": container.name,
      "id": container.short_id,
      "image": container.image.short_id,
      "status": container.status
   }
```

Para hacer pruebas, se recomienda mantener la especificación del Pod en un fichero YAML, por ejemplo en *pod.yaml.* Después es fácil cargar este fichero utilizando la librería PyYAML:

https://pyyaml.org/

```
(venv) > pip install pyyaml
```

Una vez instalada resulta muy sencillo cargar un fichero YAML con un código similar a éste:

```
import yaml
def load(path):
  f = open(path, "rt")
  text = f.read()
  f.close()
  spec = yaml.load(text, yaml.Loader)
  return spec
```

La función *listPods(labels)* se podría implementar fácilmente así:

```
def listPods(labels={}):
  print(f"listPods({json.dumps(labels)})")

  # 1. Compose query
  filters = {"label": ["k1s=pod"]}
  if len(labels) > 0:
    for lbl in labels:
      filters["label"].append(f"{lbl}={labels[lbl]}")

  # 2. Make query
  containers = client.containers.list(all=True, filters=filters)

  # 3. Prepare results
  pods = []
  for container in containers:
   pods.append({
      "name": container.name,
      "id": container.short_id,
      "image": container.image.short_id,
      "status": container.status
   })

 return pods
```

Para eliminar Pods:

```
def removePod(name):
  print(f"removePod({name})")

  try:
    container = client.containers.get(name)
    container.remove(force=True)
  except:
    pass
```

Por último, podríamos recuperar los detalles de un Pod utilizando la función *getPod(name)*. En este caso, además de los datos básicos, podemos obtener información sus etiquetas, su dirección IP, sus logs y el consumo de recursos (stats).

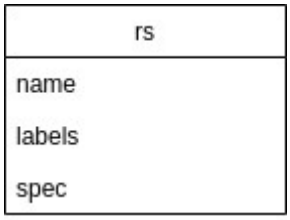
```
def getPod(name):
  print(f"getPod({name})")

  try:
    container = client.containers.get(name)
    return {
       "name": name,
       "id": container.short_id,
       "image": container.image.short_id,
       "labels": container.labels,
                "addr":  container.attrs["NetworkSettings"]["Networks"]["bridge"]
["IPAddress"],
       "status": container.status,
       "logs": container.logs().decode(),
       "stats": container.stats(stream=False)
     }
    except:
        return None
```

3.2 ReplicaSets

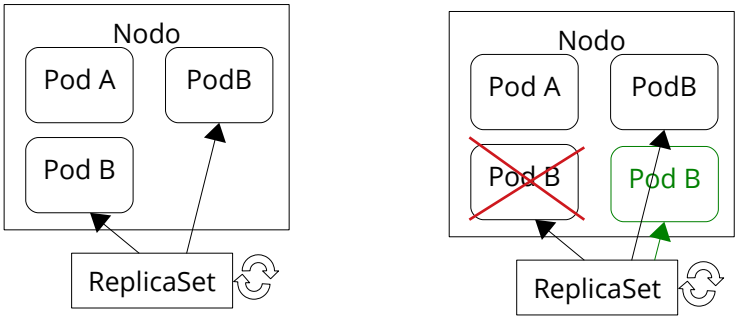
Desafortunadamente, Docker no proporciona soporte directo para este recurso, de manera que tendremos que simularlo nosotros. Para especificar su creación es necesario proporcionar una especificación compatible con K8s, por ejemplo:

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: test-rs
spec:
  replicas: 3
  selector:
    matchLabels:
      app: web-server
  template:
    metadata:
      labels:
        app: web-server
        version: "1.0"
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
          protocol: TCP
```

Como el ReplicaSet es un artefacto que se construye por encima de Docker, necesitaremos almacenarlo en la base de datos, para acordarnos de él. Proponemos el siguiente sencillo modelo de datos. El nombre y las etiquetas serán necesarias para los listados. Toda la información sobre el ReplicaSet residirá en el campo “spec”.



Como ya sabemos, un ReplicaSet es un controlador que mantiene una colección de réplicas funcionando en todo momento, tal y como ilustra la siguiente figura.



Por lo tanto, cuando se añade un ReplicaSet a K1s será necesario crear un Thread que se encargue de monitorizar en todo momento el número de réplicas que se ejecuta de un determinado Pod. Crearemos un diccionario *ReplicaSets* que mantendrá toda la información necesaria sobre los ReplicaSets activos.

```
ReplicaSets = {}
```

Para crear un Thread podemos utilizar la siguiente primitiva, donde “target” apunta a la función que ejecutará el Thread y “args” contiene los parámetros que se le pasarán. Una vez creado el Thread es necesario arrancarlo con *.start()*. Si queremos esperar a que un Thread finalice su ejecución, lo haremos con el método *.join()*.

```
import threading
thread = threading.Thread(target=Func, args=(arg,))
thread.start()
thread.join()
```

La función que ejecuta el Thread podría denominarse *ReplicaSet()*. A continuación presentamos el esqueleto que podría presentar la función *addReplicaSet(rs)*:

```
def addReplicaSet(rs):
  print(f"addReplicaSet({json.dumps(rs)})")

  # 1. Check specck
  if "apiVersion" not in rs: raise Exception("apiVersion field missing")
  if rs["apiVersion"] != "apps/v1": raise Exception("invalid apiVersion")
  if "kind" not in rs: raise Exception("kind field missing")
  if rs["kind"] != "ReplicaSet": raise Exception("invalid kind")

  # [TODO] check all the fields are correct!!!
  ...
```

```
  # 2. Check the ReplicaSet does not exist yet (by name)
  if rs["metadata"]["name"] in ReplicaSets:
    raise Exception("ReplicaSet already exists")

  # 3. Insert ReplicaSet in database
  ...

  # 4. Add entry to ReplicaSets table
  ReplicaSets[rs["metadata"]["name"]] = {
    "spec": rs,
    "status": "starting",
    "thread": threading.Thread(target=ReplicaSet,args=(rs["metadata"]["name"],))
  }

  # 5. Start Thread
  ReplicaSets[rs["metadata"]["name"]]["thread"].start()

  # 6. Return basic info of ReplicaSet
  return {
      "name": rs["metadata"]["name"],
      "replicas": rs["spec"]["replicas"]
   }
```

La función *ReplicaSet(name)* es clave. Deberá de comprobar en bucle que se ejecuta el número correcto de réplicas especificado. Si hay menos de las que debería, se deberán crear nuevas. Si hay más de las que debería se deberán eliminar las que sobran. Para el listado, creación y eliminación de réplicas podemos utilizar las operaciones sobre Pods que ya hemos implementado *listPods(), addPod()* y *removePod()*. A continuación se presenta un posible esqueleto de esta función:

```
def ReplicaSet(name):

  # 1. Get info about the ReplicaSet in the table
  rs = ReplicaSets[name]
  rs["status"] = "running"

  while rs["status"] == "running":
    # 2. Get number of replicas to control
    replicas = int(rs["spec"]["spec"]["replicas"])

    # 3. Get Pods
    pods = listPods(rs["spec"]["spec"]["selector"]["matchLabels"])

    # [TODO] filter only running Pods

    # 4. Increase/decrease Pods
    if len(pods) < replicas:
      # Increase pods
      ...
    elif len(pods) > replicas:
      # Decrease pods
      ...

    # 5. Sleep for the next tick
    time.sleep(REPLICASET_UPDATE_TIME)

  # 6. The ReplicaSet has been stopped; delete Pods
  ...
```

Faltaría implementar las operaciones:

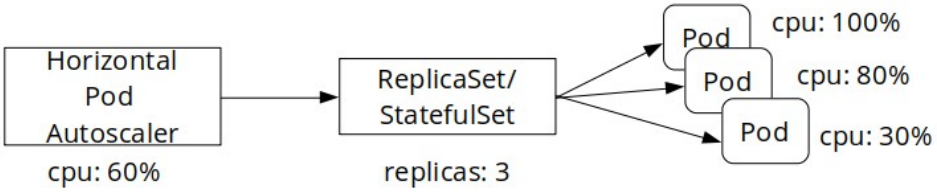
- *removeReplicaSet(name)*: elimina un Replicaset por nombre
- *getReplicaSet(name)*: recupera información detallada de un ReplicaSet por nombre
- *listReplicaSets(labels):* recupera información breve sobre los ReplicaSets que poseen las etiquetas especificadas

Antes de finalizar esta sección es importante resaltar que el ReplicaSet es un recurso mantenido íntegramente por K1s. Esto significa que cuando K1s finaliza su ejecución, los ReplicaSets dejan de funcionar. Esto no sucede por ejemplo con los Pods. Como los Pods son directamente mantenidos por Docker, aunque K1s deje de funcionar, los Pods seguirán en ejecución.

Para asegurarnos de que K1s puede ser parado y arrancado, y todo sigue funcionando correctamente, será necesario introducir cierto código relacionado con los Replicasets en *init()*. En concreto, se debería de recuperar toda la información sobre los ReplicaSets añadidos al sistema, y arrancarlos adecuadamente, añadiéndolos a la tabla *ReplicaSets*, iniciando los Threads y demás.

3.3 HorizontalPodAutoscalers

Este recurso tampoco está soportado por Docker; habrá que simularlo. Como ya sabemos, se construye por encima de un ReplicaSet y ajusta el número de réplicas que debería mantener en función del consumo medio de recursos deseado.



Para especificar su creación es necesario proporcionar una especificación compatible con K8s, por ejemplo:

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: test-hpa
spec:
  maxReplicas: 5
  minReplicas: 1
  metrics:
  - type: Resource
    resource:
      name: cpu
      target: {type: Utilization, averageUtilization: 60}
  scaleTargetRef:
    apiVersion: apps/v1
    kind: ReplicaSet
    name: test-rs
```

Para implementar este recurso se recomienda seguir un esquema muy similar al del Replicaset. De este modo, habría que crear una tabla *HorizontalPodAutoscaler*, en la que se registraría una entrada por cada nuevo HorizontalPodAutoscaler que se añada al sistema.

```
HorizontalPodAutoscalers = {}
```

Además, por cada HorizontalPodAutoscaler deberá existir un Thread que se ejecute el código de este controlador. Lo implementaremos en la función *HorizontalPodAutoscaler()*. Esta función poseerá un esqueleto similar al siguiente:

```
def HorizontalPodAutoscaler(name):
  # 1. Get info about the HorizontalPodAutoscaler in the table
  hpa = HorizontalPodAutoscalers[name]
  hpa["status"] = "running"

  while hpa["status"] == "running":
    # 2. Calculate metrics of the Pods controlled by the HorizontalPodAutoscaler
    ...

    # 3. Calculate desired Pods
    ...

    # 4. Update replicas in ReplicaSet
    ...

    time.sleep(HORIZONTALPODAUTOSCALER_UPDATE_TIME)
```

Para calcular las métricas de un contenedor se puede utilizar el método *client.container.get(name).stats(stream=False)*, de hecho ya lo hemos usado en la función *getPod(name)*. Sería similar a efectuar este comando por consola:

```
> docker stats [<pod-name>]
```

Por ejemplo, para obtener el consumo medio de CPU en un contenedor se puede usar el siguiente código:

```
stats = client.containers.get(name).stats(stream=False)
cpuDelta = stats["cpu_stats"]["cpu_usage"]["total_usage"] -
stats["precpu_stats"]["cpu_usage"]["total_usage"]
sysDelta = stats["cpu_stats"]["system_cpu_usage"] - stats["precpu_stats"]
["system_cpu_usage"]
cpuPercent = (cpuDelta/sysDelta) * stats["cpu_stats"]["online_cpus"] * 100
```

Para obtener el consumo de memoria en un contenedor se puede usar el siguiente código:

```
mem = stats["memory_stats"]["usage"]
mem = int(mem/1000000)
```

Habrá que implementar las funciones:

- *addHorizontalPodAutoscaler(hpa)*: añade un nuevo HorizontalPodAutoscaler
- *removeHorizontalPodAutoscaler(name)*: elimina un HorizontalPodAutoscaler por nombre
- *getHorizontalPodAutoscaler(name)*: recupera información detallada de un HorizontalPodAutoscaler por nombre
- *listHorizontalPodAustoscalers(labels)*: recupera información breve sobre los HorizontalPodAutoscalers que poseen las etiquetas especificadas

Por último, para hacer pruebas realistas y comprobar la corrección del HorizontalPodAutoscaler se recomienda trabajar con contenedores que hagan un uso real de la CPU/memoria. Para ello podemos utilizar el binario Linux *stress-ng*.

```
> sudo apt install stress-ng
> man stress-ng
```

Por ejemplo, con el siguiente comando se ejecuta un proceso que mantiene un consumo medio de una CPU del 50%:

```
> stress-ng -c 1 -l 50
```

4. La API RESTful

La API RESTful será implementada en el fichero *k1s_rest.py.* Utilizaremos Flask para ello.

A continuación se resume la API mínima que este servicio RESTful deberá implementar:

Método	URL	Comentarios
Recurso Pods		
`GET`	`/k1s/pod`	Recupera información breve sobre los Pods con las etiquetas especificadas Parámetros: labels Resultado (JSON): [pod]
`GET`	`/k1s/pod/XXX`	Recupera información detallada sobre el Pod con nombre XXX Resultado (JSON): podExtra
`POST`	`/k1s/pod`	Crea un nuevo Pod Contenido (JSON): pod Resultado (JSON): pod
`DELETE`	`/k1s/pod/XXX`	Elimina el Pod con nombre XXX
Recurso ReplicaSets		
`GET`	`/k1s/rs`	Recupera información breve sobre los ReplicaSets que poseen las etiquetas especificadas Parámetros: labels Resultado (JSON): [rs]
`GET`	`/k1s/rs/XXX`	Recupera información detallada sobre el ReplicaSet con nombre XXX Resultado (JSON): rsExtra
`POST`	`/k1s/rs`	Crea un nuevo ReplicaSet Contenido (JSON): rs Resultado (JSON): rs
`DELETE`	`/k1s/rs/XXX`	Elimina el ReplicaSet con nombre XXX
Recurso HorizontalPodAutoscaler		
`GET`	`/k1s/hpa`	Recupera información breve sobre los HorizontalPodAutoscalers que poseen las etiquetas especificadas Parámetros: labels Resultado (JSON): [hpa]
`GET`	`/k1s/hpa/XXX`	Recupera información detallada sobre el HorizontalPodAutoscaler con nombre XXX Resultado (JSON): hpaExtra
`POST`	`/k1s/hpa`	Crea un nuevo HorizontalPodAutoscaler Contenido (JSON): hpa Resultado (JSON): hpa
`DELETE`	`/k1s/hpa/XXX`	Elimina el HorizontalPodAutoscaler con nombre XXX

5. El CLI

A continuación se muestra los comandos que soportará la herramienta CLI k1s_ctl.py:

```
k1s_ctl help: print this message
k1s_ctl create [pod | rs | hpa] <file>: create the resource specified in file
k1s_ctl get [pod | rs | hpa] [-l lbl=val ]: list all resources, or list by label
k1s_ctl describe [pod | rs | hpa] <name>: describe the specified resource
k1s_ctl delete [pod | rs | hpa] <name>: delete the specified resource
```