UNIVERSITAT POLITÈCNICA DE VALÈNCIA

CAMPUS D'ALCOI

DSIC

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Cloud computing

Laboratorio 1

Contenido

1. Introducción

En este laboratorio se persiguen los siguientes objetivos:

- Utilizar la API de virtualización *libvirt*.
- Utilizar una API para gestión de contenedores *docker*.
- Diseñar un servicio RESTful utilizando el framework Python *Flask*.

Para ello, se propone el diseño de un servicio RESTful que publique una API capaz de gestionar máquinas virtuales de distintos tipos. Internamente, el servicio hará uso de las primitivas proporcionadas por distintos drivers.

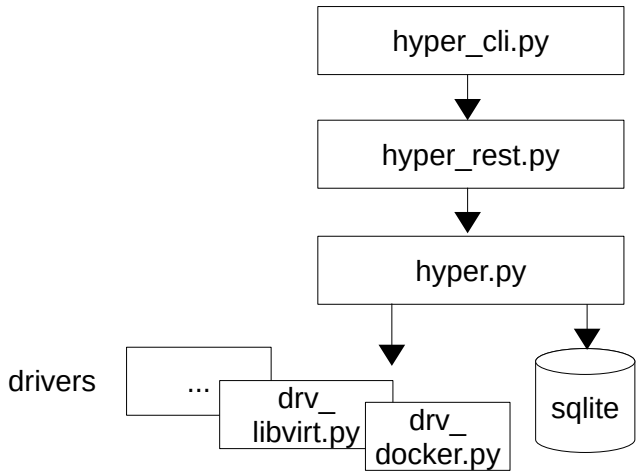
Este componente, que denominaremos *hyper* (de hipervisor/hypervisor), constituye la pieza base sobre la que se irá construyendo un entorno de cloud computing en los siguientes laboratorios.

En el apartado 2 se presenta la arquitectura del sistema a implementar, compuesto de tres capas. En el resto de apartados se discute acerca de la implementación de estas capas.

2. Arquitectura del sistema

Se propone una arquitectura compuesta de los siguientes elementos básicos:

- *hyper_cli.py*: implementa un sencillo cliente CLI (*Command-Line Interface*) que se comunica con el servicio RESTful y que permite ejecutar las principales operaciones publicadas por el sistema.
- *hyper_rest.py*: implementa la API RESTful utilizando el framework *Flask*. Redirige las operaciones solicitadas a *hyper.py*.
- *hyper.py*: contiene el modelo (la lógica) de la aplicación. Permite gestionar distintos tipos de máquinas virtuales. Registra toda la información en una base de datos SQLite. Las operaciones sobre cada tipo de máquina virtual son implementadas por un driver diferente. Todos los drivers implementan una API homogénea.
- *drv_libvirt.py*: driver que implementa todas las operaciones sobre máquinas virtuales de tipo *Libvirt*.
- *drv_docker.py*: driver que implementa todas las operaciones sobre máquinas virtuales de tipo *Docker*.



Para comenzar se creará el directorio raíz del proyecto, por ejemplo */lab1*. En su interior se creará un entorno virtual, y se activará:

```
> mkdir lab1
> cd lab1
> python3 -m venv venv
> source venv/bin/activate
```

Para este proyecto se propone la siguiente estructura de ficheros y directorios.

```
lab1/
  |- hyper.py
  |- hyper_rest.py
  |- hyper_cli.py
  |- drivers/
      |- drv_libvirt.py
      |- drv_docker.py
```

En las siguientes secciones se proporcionan más detalles acerca de todos estos módulos.

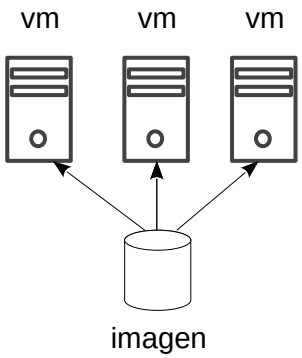
3. El modelo (*hyper.py*)

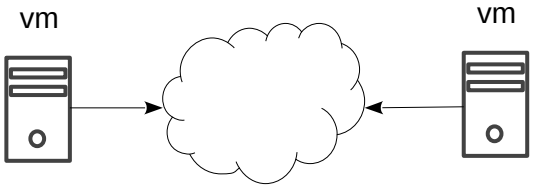
A continuación se presenta el modelo que implementará el gestor de máquinas virtuales *hyper*.

El sistema permitirá la gestión de múltiples máquinas virtuales. Ello incluye su creación, listado, arranque, parada y destrucción.

Cada máquina virtual posee un *tipo*, y es controlada por un driver diferente (e.g. libvirt, docker, etc.). Además, cada máquina virtual se basa en una *imagen*, que posee un identificador o nombre. Se pueden crear distintas máquinas virtuales a partir de la misma imagen. De momento, para simplificar, asumiremos que estas imágenes están previamente registradas en el sistema. Veremos en futuros laboratorios cómo gestionar las imágenes.

Todas las máquinas virtuales poseen conectividad con el exterior. Ya veremos en futuros laboratorios cómo conectar distintas máquinas virtuales entre sí.





3.1 La API

A continuación se lista la API que el módulo *hyper.py* implementará, y que será accedida por el módulo *hyper_rest.py*.

`hyper.listDrivers()`
Lista los drivers disponibles en el sistema.
Devuelve una lista de diccionarios.

`hyper.addMachine(mach)`
Crea una nueva máquina virtual, utilizando para ello el driver apropiado
Devuelve un diccionario con la información de la máquina creada.

`hyper.startMachine(machId)`
Arranca la máquina virtual especificada, utilizando para ello el driver apropiado

`hyper.stopMachine(machId)`
Para la máquina virtual especificada, utilizando para ello el driver apropiado

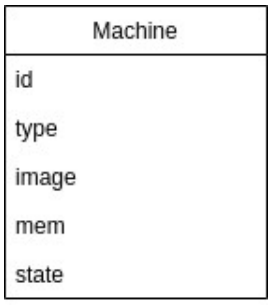
`hyper.removeMachine(machId)`
Elimina la máquina virtual especificada, utilizando para ello el driver apropiado

`hyper.listMachines(query)`
Lista las máquinas virtuales que verifican las condiciones de la consulta.
Devuelve una lista de diccionarios.

3.2 SQLite

La información será almacenada en una base de datos SQLite. Para ello, será necesario importar el módulo *sqlite3* y construir un esquema de base de datos capaz de almacenar toda la información requerida.

Para almacenar la información del sistema, en este primer laboratorio sería suficiente con implementar un modelo de datos similar al siguiente:



Esto se podría conseguir, por ejemplo, con el siguiente código:

```
import slite3
...
con = sqlite3.connect(FILE_DB)
cur = con.cursor()
cur.execute("create table machines(id varchar(32), type varchar(32), image
varchar(32), mem int, state varchar(32))")
con.commit()
con.close()
```

En esta base de datos se almacenará toda la información sobre la aplicación. Obviamente el esquema no está cerrado, y será ampliado en futuros laboratorios. Se anima al alumno que añada toda la información que estime necesaria.

4. Los drivers

Cada tipo de máquina virtual soportada por el sistema está implementado en su propio driver, un módulo almacenado en el directorio /drivers. Comenzamos con dos: *drv_libvirt.py* y *drv_docker.py*.

El sistema de drivers funciona del siguiente modo. Al arrancar *hyper.py* lista el contenido del directorio /drivers e importa todos los drivers. Cada driver es un módulo que ofrece la siguiente interfaz:

`def info()`
Devuelve información sobre el driver.
Devuelve un diccionario, que *hyper.py* deberá de registrar internamente. El diccionario debería de mantener al menos la entrada “name”, que determina el tipo de máquinas virtuales que el driver puede gestionar.

`def addMachine(mach)`
Crea una nueva máquina virtual.
Devuelve un diccionario con la información de la máquina creada.

`def startMachine(machId)`
Arranca la máquina virtual especificada.

`def stopMachine(machId)`
Para la máquina virtual especificada.

`def removeMachine(machId)`
Elimina la máquina virtual especificada.

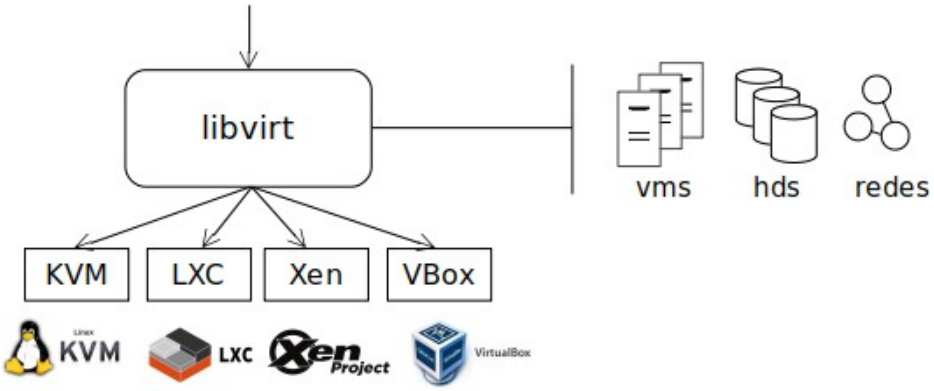
Podemos importar los drivers de manera dinámica utilizando el siguiente código. En el diccionario drivers mantenemos todos los drivers registrados en el sistema.

```
DIR_DRIVERS = "drivers"
drivers = {}
paths = os.listdir(DIR_DRIVERS)
   for path in paths:
      if path.endswith(".py"):
         drv = importlib.import_module(DIR_DRIVERS + "." + path[0:-3])
         drivers[drv.info()['name']] = drv
```

5. Libvirt

5.1 Introducción a Libvirt

Para implementar este driver será necesario utilizar la librería *Libvirt*. *Libvirt* implementa una API que permite acceder a múltiples hipervisores, por medio de un sistema de drivers.



Para instalarlo será necesario instalar los siguientes paquetes:

```
> sudo apt install libvirt-daemon-system libvirt-clients virt-manager python3-libvirt
```

A continuación se añadirá al usuario al grupo *Libvirt* y será necesario reiniciar la máquina:

```
> sudo usermod -aG libvirt alumno
> sudo shutdown -r now
```

Se puede trabajar con *Libvirt* de los siguientes modos:

- Gráficamente utilizando el cliente *virt-manager*
- En línea de comandos utilizando el cliente *virsh*.
- Desde Python utilizando la librería *libvirt-python*.

Para crear una máquina virtual se partirá de una imagen que posee un sistema operativo ya instalado. La mayoría de distribuciones Linux proporcionan estas imágenes. Por ejemplo, en la siguiente URL es posible acceder a las de la distribución Debian:

https://cloud.debian.org/images/cloud/

Para este laboratorio, se escogerá por ejemplo la máquina *bullseye/xxx/debian-11-nocloud-amd64-xxx.qcow2*

Esta imagen dispone de un usuario root sin password.

A continuación se abrirá *virt-manager* (el cliente gráfico) y se creará una nueva máquina virtual partiendo de dicha imagen. Desde el cliente gráfico es fácil operar con la máquina virtual.

Si se inicia sesión en esa imagen se puede observar que el código de teclado no se corresponde con el teclado español. Para modificarlo basta con ejecutar los comandos:

```
> apt-update && apt install console-data
> loadkeys es
```

También es posible trabajar con *Libvirt* a través del cliente *virsh*[1]:

```
> virsh -c qemu:///system list –all
> virsh -c qemu:///system start <mach>
> virsh -c qemu:///system shutdown <mach>
> virsh -c qemu:///system destroy <mach>
```

Para conectarse a su consola podemos usar el comandos:

```
> virsh -c  qemu:///system console <mach>
```

Se puede salir de la consola utilizando Ctrl+5.

Para crear una máquina virtual es necesario especificar un fichero XML[2]. La aproximación más sencilla es partir de una plantilla XML de una máquina ya creada y modificarla a nuestra conveniencia. Podemos hacerlo con el siguiente comando.

```
> virsh -c qemu:///system dumpxml <mach> > mach.xml
> virsh -c qemu:///system create mach.xml
```

También se puede trabajar con *Libvirt* utilizando el binding de Python[3], que es un mapeo directo de la API[4] C a Python. Es posible consultar un manual sobre la API de Python en la dirección:

https://libvirt.org/docs/libvirt-appdev-guide-python/en-US/html/

La API está disponible en esta dirección:

https://libvirt-python.readthedocs.io/

El módulo Python de *Libvirt* está instalado globalmente. Para poder acceder a éste desde un entorno virtual Python es necesario modificar la siguiente propiedad en el fichero *venv/pyenv.cfg:*

```
include-system-site-packages = true
```

En Python, primero es necesario crear una conexión contra *Libvirt*. Después se puede operar a través de dicha conexión.

```
con = libvirt.open("qemu:///system")
doms = con.listAllDomains()
for dom in doms: print(dom.ID())
dom = con.lookupByName(name)
dom = con.createXML(template, 0)  # crea y arranca una máquina virtual
dom.shutdown()
dom.destroy()
```

1 https://libvirt.org/manpages/virsh.html
2 https://libvirt.org/formatdomain.html
3 https://libvirt.org/python.html
4 https://libvirt.org/html/index.html

5.2 Implementación del driver de Libvirt

En el proyecto Python se crearán los siguientes ficheros y directorios adicionales:

lab1/
  |- hyper.py
  |- hyper_rest.py
  |- hyper_cli.py
  |- drivers/
    |- drv_libvirt.py
    |- drv_docker.py
    |- libvirt_template.xml
    |- libvirt_images/
      |- debian11.qcow2
      |- debian12.qcow2
    |- livirt-machines/
      |- mach1.qcow2
      |- mach2.qcow2

- */drivers/libvirt_images*: contendrá las imágenes a partir de las cuales se pueden crear nuevas máquinas virtuales. Añadir una nueva imagen es tan sencillo como copiar un nuevo archivo .qcow2 en este directorio. Para simplificar, es recomendable que los nombres de los archivos sean cortos (ej. *debian.qcow2*, *ubuntu.qcow2*, etc.)
- */drivers/libvirt_machines*: contendrá los discos virtuales con los que se crean las máquinas virtuales. Cuando se crea una máquina virtual a partir de una imagen, se crea una copia de dicha imagen y se deposita en este directorio.
- */drivers/libvirt_template.xml*: contendrá la plantilla XML que permite crear nuevas máquinas virtuales. Cuando se crea una nueva máquina se carga la plantilla, se reemplazan ciertas constantes en su interior y el resultado se utilizará para crear el dominio en *libvirt*.

A continuación se dan recomendaciones básicas para implementar cada operación.

`drv.addMachine(mach)`

Se copia la imagen definida en mach["image"] del directorio /drivers/libvirt_images al directorio /drivers/libvirt_machines con un nombre de fichero igual a mach["id"].
Se lee la plantilla /drivers/libvirt_template.xml y se modifican las configuraciones específicas de la máquina (el id, la mem, el disk, etc.)
Se crea el dominio con libvirt.defineXML().

`drv.startMachine(machId)`

Se busca el dominio con libvirt.lookupByName().
Se arranca el dominio con libvirt.create().

`drv.stopMachine(machId)`

Se busca el dominio con libvirt.lookupByName().
Se para el dominio con libvirt.shutdown().

`drv.removeMachine(machId)`

Se busca el dominio con libvirt.lookupByName().
Se elimina el dominio con libvirt.undefine().
Se elimina de /drivers/libvirt_machines el fichero que contiene la máquina virtual.

6. Docker

6.1 Introducción a Docker

Docker es un sistema de virtualización a nivel del sistema operativo. Las máquinas virtuales que gestiona reciben el nombre de contenedores. En Python podemos comunicarnos con el demonio de Docker utilizando el Docker SDK for Python:

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

Para instalarlo:

```
> pip install docker
```

Primero creamos un cliente:

```
import docker
client = docker.from_env()
```

Una vez disponemos del cliente, podemos mapear fácilmente las operaciones disponibles en el cliente por línea de comandos a la API. Por ejemplo, todas las operaciones relacionadas con contenedores están disponibles bajo el prefijo client.containers.XXX. Todas las operaciones relacionadas con imágenes bajo el prefijo client.images.XXX, etc.

Por ejemplo, podemos ejecutar contenedores con el comando:

```
client.containers.run("ubuntu", "echo hello world")
```

En background:

```
client.containers.run("nginx", detach=True)
```

Para listarlos:

for c in client.containers.list(all=True): print(c)

Cada contenedor se puede gestionar fácilmente:

```
c = client.containers.get(id)
c.id, c.image, c.name, c.status
c.start()
c.stop()
c.remove()
```

También resulta sencillo trabajar con las imágenes de Docker usando la API disponible bajo el prefijo client.images.XXX. Por ejemplo:

```
client.images.pull(“debian”)
client.images.list()
client.images.remove(id)
img = client.images.get(id)
img.id
```

6.2 Implementación del driver de Docker

Este driver será muy sencillo de implementar.

`drv.addMachine(mach)`
Se crea el contenedor en Docker

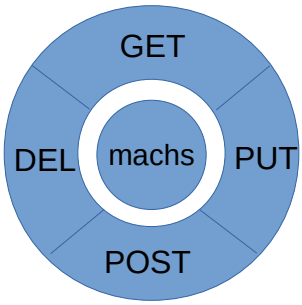
`drv.startMachine(machId)`
Se arranca el contenedor en Docker

`drv.stopMachine(machId)`
Se para el contenedor en Docker

`drv.removeMachine(machId)`
Se elimina el contenedor de Docker

7. El servicio RESTful (*hyper_rest.py*)

El primer paso para construir un servicio consiste en diseñar su API (contrato). En este caso se va a diseñar un servicio RESTful, y por tanto todas las operaciones deben girar alrededor de una colección de recursos. De momento, únicamente se publicarán máquinas virtuales.



Por tanto, únicamente se publicará un punto de entrada principal:

- */hyper/machines*

A continuación se resume la API del servicio RESTful en la siguiente tabla:

Método	**URL**	**Comentarios**
Recurso machines		
`GET`	`/hyper/machines`	Recupera las máquinas virtuales Parámetros: query Resultado (JSON): [machine]
`GET`	`/hyper/machines/XXX`	Recupera la máquina con id XXX Resultado (JSON): machine
`POST`	`/hyper/machines`	Crea una nueva máquina Contenido (JSON): machine Resultado (JSON): machine
`PUT`	`/hyper/machines/XXX`	Actualiza el estado de la máquina con id XXX Contenido (JSON): {“state”:<estado>”} Resultado (JSON): machine
`DELETE`	`/hyper/machines/XXX`	Elimina la máquina con id XXX

A continuación se instalará el framework *Flask* en el entorno virtual Python.

```
(venv) > pip install flask
```

El siguiente paso consiste en implementar el servicio definiendo una colección de rutas, tal y como se ha visto en clase:

```
from flask import Flask, request, jsonify
import json
import hyper

app = Flask(__name__)

@app.route("/hyper/machines", methods=["POST"])
def addMachine():
...
@app.route("/hyper/machines/<machId>", methods=["PUT"])
def updateMachine(machId):
...
@app.route("/hyper/machines/<machId>", methods=["DELETE"])
def removeMachine(machId):
...
@app.route("/hyper/machines")
def listMachines():
...
```

8. El CLI (*hyper_cli.py*)

Por su parte, el CLI suministrará una interfaz basada en consola y hará uso de la librería *requests* (http para humanos).

```
(venv) > pip install requests
```

A continuación se muestra un posible listado de opciones disponibles a través de esta herramienta:

```
Usage: python hyper_cli.py <command> [options]
Commands:
      add <image> [-i id] [-t <type>] [-m <mem>]
      start <id>
      stop <id>
      rm <id>
      list {field=value}
      exit
```

Si el proxy de la UPV da problemas en la comunicación con el servicio, es necesario eliminar las siguientes variables de entorno:

```
> unset HTTP_PROXY
> unset HTTPS_PROXY
> unset http_proxy
> unset https_proxy
```