Centros de Datos y de Provisión de Servicios

Curso 2023-24

PRACTICA CREATIVA 1:

DESARROLLO DE UN SCRIPT PARA LA CREACIÓN AUTOMÁTICA DEL ESCENARIO DEL BALANCEADOR DE LA PRÁCTICA 2

Última actualización: 1 de noviembre de 2023

Objetivos

La práctica final del primer parcial consistirá en el desarrollo de un script en Python que automatice parcialmente la creación del escenario de pruebas del balanceador de tráfico de la segunda parte de la práctica 2 (Figura 1). Opcionalmente se podrá trabajar con otros escenarios virtuales de complejidad similar o que utilicen contenedores LXC o docker en vez máquinas virtuales KVM. Si va a trabajar sobre un escenario alternativo, consulte previamente a los profesores de la asignatura.

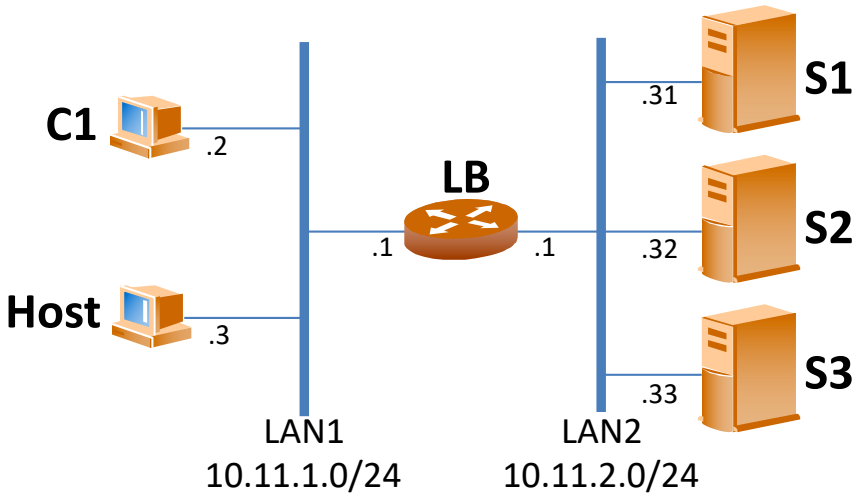

Figura 1: Escenario virtual PC1

Requisitos funcionales mínimos del script

- **RQ1**: El script, de nombre '**auto-p2.py**', deberá ejecutarse en un directorio en el que inicialmente solo esté el fichero con la imagen base que utilizarán las máquinas virtuales (MV) del escenario (*cdps-vm-base-pc1.qcow2*) y la plantilla de MVs (*plantilla-vm-pc1.xml*), disponibles ambos en el directorio */lab/cdps/pc1* del laboratorio y también por web en: https://idefix.dit.upm.es/download/cdps/pc1/. El script no debe copiar los ficheros, éstos deben estar ya copiados al directorio de trabajo.

- **RQ2**: El script *auto-p2.py* se ejecutará pasándole un parámetro obligatorio que definirá la operación a realizar:

```
auto-p2.py <orden>
```

  donde el parámetro *<orden>* puede tomar los valores siguientes:
  - ***crear***, para inicializar las máquinas virtuales, creando las imágenes de diferencias (ficheros *.qcow2) y las especificaciones en XML de cada MV, así como los bridges virtuales que soportan las LAN del escenario.
  - ***arrancar***, para arrancar las máquinas virtuales y mostrar su consola.
  - ***parar***, para parar las máquinas virtuales (sin liberarlas).
  - ***liberar***, para liberar el escenario, borrando todos los ficheros creados.
- **RQ3**: El número de servidores web a arrancar deberá ser configurable (de 1 a 5). Este número se especificará en un fichero de configuración en formato JSON almacenado en el directorio de trabajo (auto-p2.json) con el siguiente formato:

```
{
    "num_serv": 2
}
```

  El script deberá comprobar que el valor del número de servidores es correcto, dando un error en caso contrario.

- **RQ4**: El script deberá realizar la configuración mínima de las MVs antes de arrancarlas, modificando los ficheros */etc/hosts*, */etc/hostname* y */etc/network/interfaces* de cada MV, incluyendo en ellos el nombre de la máquina y la configuración de red respectivamente. También deberá configurar el balanceador (LB) para que funcione como router.

- **RQ5**: Las imágenes de las MVs deben crearse mediante ficheros de diferencias (qcow2) con respecto de la imagen base (*cdps-vm-base-pc1.qcow2*).

- **RQ6**: El script debe ser "no interactivo", esto es, cualquier parámetro necesario se debe pasar por la línea de comandos; el script no debe preguntar nada interactivamente.

- **RQ7**: El script debe utilizar para generar trazas de depuración el módulo "logging" de Python. Por defecto mostrará en pantalla información breves sobre la actividad del script (p.ej.: algo relativo al comando ejecutado). Pero si se añade al fichero de configuración auto-p2.py.json la variable 'debug: true', deberá mostrar mensajes más detallados:

```
{
    "num_serv": 2,
    "debug": true
}
```

Partes opcionales

Como funcionalidades adicionales se propone incluir:

- La monitorización del escenario mediante, por ejemplo, una orden adicional (monitor) que presente el estado de todas las máquinas virtuales del escenario. Esta orden se puede ejecutar con el comando *watch* para monitorizar periódicamente el escenario.

- La configuración y arranque automático del balanceador de tráfico HAproxy en LB y de los servidores apache en S1-S3, de manera que cuando se arranque la MV esté disponible automáticamente el servicio de balanceo de tráfico entre servidores web.

- La configuración de HAproxy para que esté disponible el acceso al interfaz de gestión a través del web o mediante comandos (ver https://www.haproxy.com/blog/dynamic-configuration-haproxy-runtime-api/).
- Funcionalidad para parar y/o arrancar MVs individualmente, pasando al script como parámetro el nombre de la MV.
- Otras funcionalidades a proponer por el alumno. Se recomienda consultar con los profesores antes de realizarlas.

En la evaluación de la práctica se valorará el estilo de programación, la generalidad de los scripts y el grado de automatización alcanzado, así como las partes opcionales implementadas.

Recomendaciones

- Para la gestión de las máquinas y redes virtuales se recomienda crear una librería en Python que exporte dos objetos, **MV** y **Red**, que incluyen métodos para crear, arrancar, mostrar la consola, parar y liberar MVs y para crear y liberar redes. Para facilitar la tarea, se proporciona el esqueleto del módulo y del programa principal (ver apéndice).
- Dado que el script tiene que ejecutar múltiples comandos, se recomienda probarlos manualmente desde la línea de comandos y solo integrarlos y probarlos en el script cuando se esté seguro de que funcionan. Asimismo, se recomienda dividir las funcionalidades del script en partes y probarlas por separado en scripts de prueba (p.ej.: el paso de parámetros por la línea de comandos, la modificación de los ficheros XML, el montaje de las imágenes de las máquinas virtuales y la modificación de ficheros, etc.). Una vez que cada parte esté probada, se puede integrar el código de la prueba como una función en el script principal.
- Para la ejecución de los comandos de Linux desde el script en Python se recomienda utilizar el módulo **subprocess** y, en particular, la función **subprocess.call**. Por ejemplo:

```
subprocess.call(["ls -l"], shell=True)
```

- Los mensajes que se quiera que el programa siempre muestre en pantalla se pueden imprimir utilizando la función **print**. Por el contrario, para la depuración del script es útil que el programa muestre trazas de información en pantalla que se puedan eliminar fácilmente una vez se haya depurado el programa. En el esqueleto del programa principal y del módulo se incluye un ejemplo de cómo inicializar y utilizar el módulo **logging** de Python para generar las trazas de depuración. Más información puede encontrarse aquí: https://goo.gl/sSDT8Y.
- Para la gestión de las máquinas virtuales (arranque, parada, etc.) se recomienda utilizar el comando "**sudo virsh define|start|shutdown|destroy**" ya utilizado en prácticas anteriores. Recuerde que para las máquinas virtuales sean persistentes debe utilizar los comandos "**virsh define**" y "**virsh start**".
  - El comando **crear** debe crear las máquinas virtuales con "**virsh define**" y el comando **arrancar** arrancarlas con "**virsh start**".
  - El comando **parar** debe detener las MVs de forma ordenada usando "**virsh shutdown**" y deberá conservar el contenido de sus imágenes, de forma que toda modificación realizada en las MVs se conserve si se paran y arrancan de nuevo.
  - El comando **liberar** debe liberar las MVs utilizando "**virsh destroy**" y borrar todos los ficheros creados durante el arranque.

- Para crear y modificar los ficheros XML de definición de las MVs, se utilizará alguna de las librerías disponibles en Python para leer y modificar ficheros XML. Se recomienda utilizar la librería **lxml.etree** (http://lxml.de) disponible en el laboratorio y sobre la que se adjunta un ejemplo de uso en el apéndice.
- Para leer y escribir en formato JSON en el fichero de configuración se debe utilizar el módulo **json** de Python.
- El script debe mostrar las consolas de las máquinas virtuales del escenario cuando se arranquen (comando **arrancar**). Para mostrarlas existen dos métodos según se quiera mostrar la consola gráfica o textual:
  - Consola gráfica: ejecutar "virt-viewer <nombre_mv>"
  - Consola textual: arrancar un nuevo terminal "xterm" o "gnome-terminal" con la opción -e para que ejecute el comando "sudo virsh console <nombre_mv>". Por ejemplo:

    ```
    xterm -e "sudo virsh console s1"
    ```

  Se recomienda arrancar la consola textual. Tenga en cuenta que los comandos anteriores deben ser ejecutarlos en segundo plano (*background*) para que el script no se quede detenido cuando se ejecuta el comando (añada un "&" al final del comando para lograrlo).
- Para la parte de configuración de las máquinas virtuales (configuración del *hostname*, de los interfaces de red de las máquinas virtuales y la configuración como router del balanceador) se recomienda el procedimiento siguiente:
  - Crear los ficheros de configuración de cada máquina virtual en el host en un directorio temporal. La configuración de la máquina virtual se debe realizar modificando los ficheros **/etc/hostname** y **/etc/network/interfaces** (consulte en Internet el formato de este último). No hace falta conservar el contenido original de esos ficheros; lo más sencillo es que el script escriba su contenido completo. Por ejemplo, para el servidor s3, el contenido del fichero **/etc/network/interfaces** será:

    ```
    auto lo
    iface lo inet loopback

    auto eth0
    iface eth0 inet static
      address 10.11.2.33
      netmask 255.255.255.0
      gateway 10.11.2.1
    ```

    El contenido del fichero **/etc/hostname** es simplemente el nombre de la máquina virtual.
  - Posteriormente, copiar los ficheros de configuración a las máquinas virtuales utilizando el comando "**virt-copy-in**". Por ejemplo, para copiar el fichero **interfaces** al directorio **/etc/network** de imagen **s1.qcow2** utilizada en la MV s1:

    ```
    sudo virt-copy-in -a s1.qcow2 interfaces /etc/network
    ```
  - Adicionalmente, es necesario modificar el fichero /etc/hosts de las máquinas virtuales, cambiando la línea:

    ```
    127.0.1.1 cdps cdps
    ```

    por:

    ```
    127.0.1.1 <nombre_mv>
    ```

    Se puede realizar directamente mediante el comando **virt-edit**. Por ejemplo:

```
sudo virt-edit -a s1.qcow2 /etc/hosts -e 's/127.0.1.1.*/127.0.1.1 s1/'
```

- IMPORTANTE: para modificar ficheros en la imagen de una máquina virtual es necesario que esta esté parada.
- Para comprobar que un fichero se ha modificado correctamente, se puede utilizar el comando "**virt-cat**", que nos permite ver el contenido del fichero almacenado en la imagen. Por ejemplo:

```
sudo virt-cat -a s1.qcow2 /etc/network/interfaces
```

- También es posible copiar ficheros desde la imagen de la máquina virtual al host mediante el comando "**virt-copy-out**". Por ejemplo, para copiar el fichero **/etc/network/interfaces** de la imagen al directorio actual:

```
sudo virt-copy-out -a s1.qcow2 /etc/network/interfaces .
```

- Finalmente, para ver los ficheros que hay dentro de una imagen se puede usar el comando "**virt-ls**". Por ejemplo, para ver el contenido del directorio **/etc** de una imagen:

```
sudo virt-ls -a s1.qcow2 -l /etc
```

- Para que el balanceador de tráfico funcione como router al arrancar, se recomienda editar el fichero **/etc/sysctl.conf** tal como se describe en http://www.ducea.com/2006/08/01/how-to-enable-ip-forwarding-in-linux/. Para ello puede utilizar el siguiente comando:

```
sudo virt-edit -a lb.qcow2 /etc/sysctl.conf \
   -e 's/#net.ipv4.ip_forward=1/net.ipv4.ip_forward=1/'
```

- Para la parte opcional de monitorización del escenario se pueden utilizar algunos de los comandos que proporciona **virsh** para obtener información sobre las máquinas virtuales, por ejemplo, los comandos **domstate, dominfo, cpu-stats**, etc (consultar el manual mediante "**man virsh**"). También se puede utilizar un ping desde el host para chequear la conectividad con las MVs.
- Para la parte opcional de arranque del balanceador de tráfico y los servidores web durante el arranque de LB, se recomienda utilizar el fichero **/etc/rc.local**. Recuerde que la imagen de las máquinas virtuales utilizada tiene ya instalado el servidor apache2 y el balanceador HAproxy, pero no se arrancan automáticamente.

Apéndices

Fichero lib_mv.py

```
import logging

log = logging.getLogger('auto_p2')

class MV:
  def __init__(self, nombre):
    self.nombre = nombre
    log.debug('init MV ' + self.nombre)

  def crear_mv (self, imagen, interfaces_red, router):
    log.debug("crear_mv " + self.nombre)

  def arrancar_mv (self):
    log.debug("arrancar_mv " + self.nombre)

  def mostrar_consola_mv (self):
    log.debug("mostrar_mv " + self.nombre)

  def parar_mv (self):
    log.debug("parar_mv " + self.nombre)

  def liberar_mv (self):
    log.debug("liberar_mv " + self.nombre)

class Red:
  def __init__(self, nombre):
    self.nombre = nombre
    log.debug('init Red ' + self.nombre)

  def crear_red(self):
      log.debug('crear_red ' + self.nombre)

  def liberar_red(self):
      log.debug('liberar_red ' + self.nombre)
```

Fichero auto-p2.py

```
#!/usr/bin/env python

from lib_mv import MV
import logging, sys

def init_log():
    # Creacion y configuracion del logger
    logging.basicConfig(level=logging.DEBUG)
    log = logging.getLogger('auto_p2')
    ch = logging.StreamHandler(sys.stdout)
    formatter = logging.Formatter('%(asctime)s - %(levelname)s - %(message)s', "%Y-%m-%d
%H:%M:%S")
    ch.setFormatter(formatter)
    log.addHandler(ch)
    log.propagate = False

def pause():
    programPause = raw_input("Press the <ENTER> key to continue...")

# Main
init_log()
print('CDPS - mensaje info1')

# Ejemplo de creacion de una maquina virtual
s1 = MV('s1')
pause()
s1.crear_mv('cdps-vm.qcow2', 'if1', False )
pause()
s1.arrancar_mv()
pause()
s1.parar_mv()
pause()
s1.liberar_mv()
```

Ejemplo de uso de la librería lxml

- Fichero XML de ejemplo:

```
<objeto tipo='A'>
  <nombre>objeto1</nombre>
  <parte1>
    <nombre>p1</nombre>
  </parte1>
  <parte2>
    <nombre>p2</nombre>
  </parte2>
  <parte3>
    <nombre>p2</nombre>
    <cacho nombre='c1'>
      <cachito nombre='c11'/>
    </cacho>
    <cacho nombre='c2'>
    </cacho>
  </parte3>
</objeto>
```

- Programa de ejemplo que procesa y modifica el fichero XML anterior:

```
#!/usr/bin/python3

from lxml import etree

def pause():
    p = input("Press <ENTER> key to continue...")

# Cargamos el fichero xml
tree = etree.parse('test.xml')

# Lo imprimimos en pantalla
print(etree.tounicode(tree, pretty_print=True))
pause()

# Obtenemos el nodo raiz e imprimimos su nombre y el valor del atributo 'tipo'
root = tree.getroot()
print(root.tag)
print(root.get("tipo"))
pause()

# Buscamos la etiqueta 'nombre' imprimimos su valor y luego lo cambiamos
name = root.find("nombre")
print(name.text)
name.text = 'kiko'
print(name.text)
pause()

# Buscamos la etiqueta 'nombre' bajo el nodo 'parte3', imprimimos su valor y lo
cambiamos
nombre_parte3=root.find("./parte3/nombre")
print(nombre_parte3.text)
nombre_parte3.text='veneno'
print(nombre_parte3.text)
pause()

# Buscamos el nodo 'cacho' bajo 'parte3' con nombre 'c1', imprimimos su valor y lo
cambiamos
cachito=root.find("./parte3/cacho[@nombre='c1']/cachito")
print(cachito.get("nombre"))
```

```
cachito.set("nombre", "de hierro y cromo")
print(cachito.get("nombre"))
pause()

# Imprimimos el xml con todos los cambios realizados
print(etree.tounicode(tree, pretty_print=True))
```