TEMA 4. Actividades de aula

Actividad 1

OBJETIVO: Entender qué debe realizarse en cada etapa del despliegue.

ENUNCIADO: Tal como se ha mencionado en la explicación del concepto de despliegue, el despliegue consta de estas etapas:

1. Instalar los componentes *software* en sus respectivos ordenadores.
2. Iniciar los componentes.
3. Mantenerlos en servicio.
4. Pararlos y eliminarlos cuando ya no se necesiten.

Uno de los problemas principales a resolver en estas tareas de despliegue es la resolución de dependencias. Esa resolución se realiza parcialmente en dos de esas etapas: instalación e inicio.

En la etapa de instalación (tanto en aplicaciones distribuidas como en las que no lo son), los programas a instalar pueden depender de otras herramientas y bibliotecas que deberán estar ya instaladas en esos ordenadores.

En la etapa de inicio de una aplicación distribuida, cada componente puede depender de otros componentes que habrán sido iniciados previamente para que puedan interactuar sin problemas y así procesar cada petición de los clientes. Se debe definir un orden de inicio de los componentes y algún procedimiento deberá existir para resolver esas dependencias invocador-invocado entre los componentes.

Analicemos en los próximos apartados cómo se realizan las etapas de instalación en algunos sistemas actuales. Para ello, consideremos que hemos desarrollado dos programas Java que definen una aplicación distribuida y que necesitamos instalarlos en un ordenador con un sistema Ubuntu que utiliza la misma configuración que la máquina virtual utilizada en las prácticas de TSR. En esa máquina virtual no está instalada la MV Java pues estaba orientada al desarrollo de componentes JavaScript.

Se plantean las siguientes cuestiones:

1. Instale el paquete **default-jre** en su MV Ubuntu (o en cualquier otra distribución Linux sin JRE ni JDK). Describa cuántos paquetes se han tenido que instalar como requisitos de **default-jre**. Indique también cuál es el tamaño global de los ficheros descargados y el espacio en disco que ocuparán una vez instalados. Esa información se ofrece antes de confirmar la instalación. La orden necesaria para realizar esta tarea es: **sudo apt install default-jre**. Obsérvese que todos esos elementos son las dependencias de los programas que pretendíamos utilizar en nuestro servicio, puesto que ha sido desarrollado en Java.

2. La orden **apt install** busca los paquetes necesarios en los almacenes de *software* de Ubuntu, los descarga y, posteriormente, los instala y los configura, teniendo en cuenta los recursos disponibles en nuestro sistema. Explique por qué esos pasos de instalación

y configuración (es decir, las tareas a llevar a cabo tras la descarga) necesitan un tiempo prolongado para finalizar.

3. Algunas veces intentaremos instalar una aplicación que necesitará una biblioteca o algún paquete auxiliar en una versión determinada y esa versión será incompatible con la utilizada por otras aplicaciones ya instaladas en nuestro ordenador. Esto se conoce como "dependency hell" y se llega a dar en empresas que utilicen múltiples servicios desplegados en sus propios clústeres o centros de datos. ¿Hay alguna estrategia para instalar aplicaciones que resuelva este problema? ¿Puede automatizarse o requiere que los administradores la gestionen manualmente?

   No. Requiere que los administradores la gestionen manualmente

4. Algunos sistemas actuales evitan el "dependency hell" con sus gestores de paquetes de *software*. Por ejemplo, en los sistemas Ubuntu (y la máquina virtual **tsr-vbox-***curso* es un ejemplo de este tipo) la herramienta por omisión para gestionar la instalación de paquetes es **apt**. Esa herramienta no permite que instalemos paquetes que necesiten versiones incompatibles de bibliotecas u otros paquetes ya instalados. Para corroborar esto, intentemos instalar la versión 3.4.0 (que fue publicada en los primeros meses de 2014) del intérprete de **python** en nuestra máquina virtual. Esa acción puede realizarse mediante la orden:
   sudo apt install python3=3.4.0-0ubuntu2
   ¿Acepta **apt** esa versión? (Ten en cuenta que **python3** ya está instalado por omisión en esa distribución Linux. Además, la versión instalada es más reciente que la solicitada en este ejemplo) ¿Qué resultado proporciona ese intento de instalación? ¿Podremos tener en un mismo sistema la versión 3.4.0 del intérprete de Python y su última versión?

   No se podrá, ya que habrá un problema de *dependency hell.*

5. Podríamos forzar la instalación de la versión 3.4 de Python con otras herramientas. Para ello, debemos encontrar el archivo utilizado por ese paquete (p.ej., python3_3.4.0-0ubuntu2_amd64.deb) en algún depósito Ubuntu (p.ej., https://launchpad.net), descargarlo, e instalarlo con **dpkg -i** junto al nombre del archivo como argumento. La diferencia principal entre **apt** y **dpkg** es que el primero solo necesita el nombre del paquete para automatizar su descarga, la resolución de dependencias (que incluye la descarga, instalación y configuración de todos los paquetes auxiliares que sean necesarios) y la configuración del paquete solicitado. El segundo, por otra parte, necesita tener una copia local del archivo asociado a ese paquete y utiliza ese archivo para intentar su instalación; si encuentra algún error relacionado con las dependencias (p.ej., que falte alguno de los paquetes necesarios), reporta el error y aborta la instalación. ¿Llega a tener éxito este segundo intento? En caso negativo, ¿por qué no? (NOTA: Debemos ser cuidadosos. Si la instalación abortara, tendremos que restaurar aquellos paquetes cuya configuración haya sido modificada parcialmente, utilizando para ello la orden **sudo apt install -f**, sin ningún argumento adicional.)

   No llega a tener éxito debido a que ya hay instalada una versión más reciente de *Python.*

6. ¿Sabes si hay alguna herramienta que facilite la colección de recursos generados una vez se haya instalado un paquete y todas sus dependencias (algo similar a una "foto" de las modificaciones aplicadas globalmente en ese sistema como resultado de esa instalación)? ¿Cómo podría una herramienta de este tipo simplificar las tareas de despliegue de un mismo componente *software* sobre múltiples ordenadores?

   Los contenedores.

En los servicios distribuidos, el despliegue también incluye la etapa de "mantener los componentes en servicio" que ya hemos mencionado previamente como tercera etapa del despliegue. Implícitamente, esa etapa requiere que el servicio permanezca activo en todos los escenarios posibles; es decir, el servicio debe ser altamente disponible. Además, los servicios deben adaptarse dinámicamente para proporcionar siempre una calidad de servicio adecuada, tal como se habrá establecido en el SLA. Describa de manera breve (en una o dos líneas) cómo podrían gestionarse los siguientes objetivos en esa etapa del despliegue:

a) Reaccionar a un incremento de carga.

b) Reaccionar a un decremento de carga.

c) Actualizar un componente para corregir errores o vulnerabilidades.

Actividad 2

OBJETIVO: Identificar qué información debe incluirse en un plan de despliegue.

ENUNCIADO:

Todo plan de despliegue debe especificar qué componentes definen una aplicación distribuida y cuáles son las dependencias de invocación entre esos elementos. Con esa información, alguna herramienta de automatización del despliegue decidirá una secuencia válida de inicio de esos componentes a la hora de proporcionar un determinado servicio. Además, una vez finalice la primera instalación y el correspondiente inicio ordenado, esa herramienta monitorizará la carga y adaptará el número actual de instancias de cada componente para respetar los requisitos establecidos en el SLA.

Intentemos generar un diseño preliminar de la información que necesitaría una hipotética herramienta de automatización del despliegue. Para ello, se asumirá que cada componente solo puede recibir conexiones entrantes en un único puerto en aquella máquina (posiblemente, virtual) en la que haya sido desplegado. Con ese objetivo, seleccione qué elementos de la lista siguiente serían necesarios para esa herramienta (es decir, esos elementos deben configurarse en el plan de despliegue), y explique por qué se necesitan:

a) Número de puerto o *endpoint* (por componente).

b) Dirección IP (por componente).

c) Programa a ejecutar, así como sus biblotecas y otros paquetes auxiliares (por componente).

d) Nombre o identificador del componente (por componente).

e) Lista de componentes a los que este componente utilizará / invocará / enviará mensajes (por componente).

f) Número mínimo de instancias (por componente).

g) Número máximo de instancias (por componente).

h) Requisitos de CPU (por componente).

i) Requisitos de memoria (por componente).

Actividad 3

OBJETIVO: Evaluar si algunas tareas de automatización del despliegue están siendo gestionadas actualmente por los principales proveedores de sistemas de computación en la nube.

ENUNCIADO: Muchos proveedores de sistemas de computación en la nube gestionan recursos de la infraestructura. Así, podrían considerarse proveedores IaaS. Algunos también automatizan el escalado de los componentes y, por ello, también gestionan (al menos, parcialmente) las responsabilidades de un proveedor PaaS.

Seleccione alguno de los siguientes proveedores (o, si hubiera suficiente tiempo, repita la actividad para cada uno de ellos) y responda las siguientes cuestiones sobre él: (a) Amazon EC2, (b) Microsoft Azure, (c) Google Cloud.

1. Encuentre al menos una página de documentación (oficial, de la propia empresa) que explique cómo gestiona ese proveedor el autoescalado de componentes. Anote su URL.

2. ¿Puede elegir el cliente qué métricas deben considerarse para automatizar las decisiones de escalado? De ser así, ¿entre qué métricas puede elegirse?

3. ¿Menciona esa guía el SLA en su descripción de la automatización del autoescalado de servicios? Si no lo hace, ¿quién tendrá que considerar el SLA en sus decisiones de autoescalado? ¿Quiénes serán el proveedor y el consumidor relacionados por ese SLA?

4. Encuentre al menos una página oficial de documentación que explique cómo automatiza ese proveedor las actualizaciones de *software* de los propios servicios desplegados en ese sistema. ¿Ha sido posible encontrar alguna?

5. A partir de la información recogida en los apartados anteriores, ¿cree que los proveedores actuales cumplen con los requisitos mencionados en este Tema 4 para los proveedores PaaS?

Actividad 4

OBJETIVO: Comparar contenedores y máquinas virtuales.

ENUNCIADO: Supongamos que se ha instalado VirtualBox (o cualquier otro gestor de máquinas virtuales) en nuestro ordenador. En ese caso, se puede utilizar la máquina virtual **tsr-vbox-***curso* para resolver esta actividad. Si no, se puede consultar esta página para obtener información sobre imágenes Ubuntu para máquinas virtuales: https://www.osboxes.org/ubuntu/ . Con ello se podrá responder a las siguientes cuestiones:

1. Compare el tamaño de la imagen Ubuntu 20.04 para VirtualBox y la de ubuntu:20.04 para Docker. Para ello, una imagen VirtualBox mínima para ese sistema puede encontrarse en https://www.osboxes.org/ubuntu/, y el tamaño de la imagen Docker puede averiguarse utilizando **docker images** en nuestra **tsr-vbox-***curso* o en cualquier otro ordenador con alguna versión de Docker instalada. Si no existe ninguna **ubuntu:focal** o **ubuntu:20.04** en nuestro sistema Docker, se puede descargar con la orden **docker pull ubuntu:20.04**. Explica por qué se da esa diferencia de tamaño.

2. Mide de manera aproximada el tiempo necesario en cada uno de esos sistemas virtualizados (contenedor o máquina virtual) para completar su inicio. Explica por qué hay, en caso de que las haya, diferencias en este apartado. La máquina virtual puede ser iniciada desde la ventana principal del gestor correspondiente, mientras que el contenedor se iniciará con esta orden: **docker run -i -t ubuntu:***tag*

3. Explica al menos dos ventajas de una máquina virtual cuando se compara con un contenedor.

4. ¿En qué casos debemos utilizar una máquina virtual en lugar de un contenedor para ejecutar algún componente virtualizado?

5. ¿En qué casos debemos utilizar un contenedor en lugar de una máquina virtual para ejecutar algún componente virtualizado?

Actividad 5

OBJETIVO: Utilizar contenedores o máquinas virtuales para resolver el *dependency hell*.

ENUNCIADO: La Actividad 1 en este boletín presentó el concepto de "*dependency hell*" en su tercer apartado. Tanto las máquinas virtuales como los contenedores pueden utilizarse para resolver ese problema. Obsérvese que tanto las imágenes de máquinas virtuales como las de contenedores pueden incluir todas las bibliotecas y paquetes relacionados que pueda necesitar un componente determinado. Así, la descarga de paquetes, su descompresión y su configuración se realizan una sola vez durante la generación de la imagen. Posteriormente, basta con ejecutar los elementos incluidos en la imagen en una máquina virtual o en un contenedor y así, de manera inmediata, todos los problemas relacionados con las dependencias entre paquetes locales desaparecen, pues todos esos elementos ya están en el contenedor o máquina virtual, en el lugar donde se espera que estén.

Lee cuidadosamente el apartado 4 de la Actividad 1 (no se necesita repetir ese apartado 4, pero proporciona información relevante para resolver aquello que proponemos seguidamente).

Las imágenes Docker **ubuntu** incluyen por omisión el intérprete **python3** y otras herramientas auxiliares. Trata de resolver los apartados 5 y 6 de la Actividad 1, pero utilizando ahora un contenedor Docker en lugar de una instalación en el ordenador local mediante la orden **dpkg**. Para ello, considera que el número de versión asociado a una distribución Ubuntu consta de dos primeros dígitos que mencionan el año de edición y dos dígitos finales que representan el mes. Además, solo hay dos ediciones por año, en abril y octubre. Por tanto, **Ubuntu 20.04** sería el sistema Ubuntu editado en abril de 2020. Las imágenes Docker **ubuntu** siguen un etiquetado similar. Así, la imagen **ubuntu:20.04** es la que se corresponde con la edición **Ubuntu 20.04** de ese sistema.

Así, los apartados a resolver tendrían ahora estos enunciados:

5. Podríamos forzar la instalación de la versión 3.4 de Python con otras herramientas. Para ello, averigua qué imagen **ubuntu:aa.mm** del Docker Hub puede proporcionar esa versión del intérprete de Python. Descarga esa imagen y crea un contenedor con esta orden (en la que tendrás que utilizar la etiqueta apropiada):
   docker run -i -t ubuntu:*etiqueta*
   Una vez en este contenedor Docker, ejecuta el intérprete Python con la orden **python3**. El intérprete visualizará en su primera línea el número de versión. Abre otra consola en el sistema Linux anfitrión. Ejecuta allí **python3**, también. ¿Se da algún problema al intentar ejecutar ambos intérpretes en el mismo ordenador? ¿Hemos resuelto el *dependency hell* en este ejemplo concreto? ¿Se puede utilizar este tipo de solución en otros escenarios donde se dé el *dependency hell*?

6. ¿Sabes si hay alguna herramienta que facilite la colección de recursos generados una vez se haya instalado un paquete y todas sus dependencias (algo similar a una "foto" de las modificaciones aplicadas globalmente en ese sistema como resultado de esa instalación)? ¿Cómo podría una herramienta de este tipo simplificar las tareas de despliegue de un mismo componente *software* sobre múltiples ordenadores?

Actividad 6

OBJETIVO: Crear dos imágenes Docker y ejecutar múltiples contenedores con ellas.

ENUNCIADO:

Considérense estos dos programas Node.js:

```
// File: server6.js
const zmq=require('zeromq')
const rp=zmq.socket('router')
// Pseudoarray of clients.
let clients={}
// Number of clients.
let counter=0

try {
    // Bind the server socket to port number 9000.
    rp.bindSync("tcp://*:9000")
} catch(e){
    // If any error is generated, show it.
    console.log(JSON.stringify(e))
    // End the process here.
    process.exit(1)
}

// Manage message reception.
rp.on('message', (sender, msg) => {
    let cli=sender+''
    // Check whether this client has sent any message.
    if (!clients[cli]) {
        // If not, set its identifier as a new property in
        // the clients object, increase the number of
        // known clients and assign its current value to
        // this client property.
        clients[cli] = ++counter
    }
    // Show message contents and sender ID.
    console.log("Message '%s' from client '%s'.", msg+'', cli)
    // Reply to client.
    rp.send([sender,parseInt(msg+'')%clients[cli]])
})
```

```
// File: client6.js
const zmq=require('zeromq')
const rq=zmq.socket('dealer')
// Connect to the server in the local computer.
rq.connect('tcp://127.0.0.1:9000',(o) =>
    {console.log(JSON.stringify(o))})

// Manage message reception...
rq.on('message', (msg)=>{
    // Show the message contents.
    console.log('Server response: %s', msg+'')
})

// Keep a private counter and increase it each time
```

```
// a message is sent. Include its value as the
// message contents.
function generator(){
    let counter=0
    return () => {rq.send(++counter + '')}
}

// Send one message per second.
setInterval(generator(),1000)
```

Resuelva los apartados siguientes sobre esos dos programas:

1. Ejecute dos procesos clientes y un proceso servidor, utilizando esos programas y el intérprete **node**, en su ordenador. Describa qué hacen y explique en qué casos podría haber problemas cuando los clientes intenten interactuar con el servidor.

2. Utilizando como base la imagen Docker **tsr*curso*/ubuntu-zmq** (que está disponible en la máquina virtual **tsr-vbox-*curso***), construya dos nuevas imágenes Docker: una para client6.js y otra para server6.js. Para ello, escriba un Dockerfile para cada uno (y proporciónelo como respuesta en este aparado). Describa también qué orden deberá utilizarse, y dónde, para generar cada imagen. Los nombres para las imágenes resultantes deben ser: **client6** y **server6**.

3. Inicie un contenedor **client6** y otro **server6**. Determine si pueden interactuar sin problemas o no. Explique por qué se comportan de esa manera.

4. Modifique el programa client6.js para que recibe el URL del servidor como un argumento desde la línea de órdenes. Proporcione el programa resultante. Compruebe su funcionamiento correcto utilizando la orden **node**, para que interactúe con un servidor, sin que deba intervenir Docker para ello.

5. Modifique el Dockerfile original para la imagen **client6**. Use ENTRYPOINT para garantizar que la imagen siempre ejecute el programa client6.js y utilice CMD para especificar cuál será el argumento por omisión para ese programa cliente. Proporcione el fichero resultante.
   NOTA: ENTRYPOINT y CMD tienen dos sintaxis alternativas para especificar sus argumentos:
   a. Con ENTRYPOINT arg1 arg2... (o CMD arg1 arg2...) esos argumentos se pasan a un intérprete de órdenes. Por tanto, habrá un arg0 implícito que corresponde al nombre de ruta del propio intérprete de órdenes. Eso será un inconveniente para pasar los valores de los argumentos a un proceso.
   b. Con ENTRYPOINT ["arg1", "arg2"...] (o CMD ["arg1", "arg2"...]) esos argumentos se pasan directamente a la llamada al sistema que crea el proceso (en caso de ENTRYPOINT) o al proceso que gestiona esos argumentos (en caso de CMD). ¡Esta es la sintaxis recomendada para resolver esta actividad!

6. Compruebe el funcionamiento de la nueva imagen **client6**. Siga estos pasos y anote las órdenes que haya necesitado para completarlos:
    a. Pare y elimine todos los contenedores que utilizaban la imagen **client6** anterior. Utilice **docker ps** para encontrar sus IDs, **docker stop** para interrumpir su ejecución y, finalmente, **docker rm** para eliminarlos.
    b. Elimine la imagen **client6** anterior. Utilice **docker rmi** para ello.
    c. Construya la nueva imagen **client6**.
    d. Averigüe cuál es la dirección IP del contenedor **server6**. Utilice **docker ps** y **docker inspect** *identificador-de-contenedor* para ello.
    e. Inicie varios contenedores **client6**, proporcionando el URL del servidor como argumento. Compruebe si pueden interactuar correctamente con el servidor.

Actividad 7

OBJETIVO: Utilizar variables de entorno en los ficheros Dockerfile y en la orden **docker run**.

ENUNCIADO: Resuelva de nuevo los apartados 5 y 6 de la Actividad 6, pero utilice variables de entorno en el Dockerfile para para pasar el URL del servidor como argumento de la línea de órdenes para client6.js, en lugar de la instrucción CMD. La nueva imagen resultante debe llamarse **client7** y debe interactuar sin problemas con contenedores **server6**.

En este caso, no se necesitará la sintaxis ENTRYPOINT ["arg1", "arg2"...].

Considere también que se necesitará la opción **-e** (véase una descripción con **man docker run**) para especificar la variable de entorno y su valor a la hora de iniciar el contenedor **client7** con la orden **docker run**.

Como solución para esta actividad, se debe presentar el nuevo Dockerfile para la imagen **client7** y la línea de órdenes necesaria para iniciar el contenedor **client7**. Suponga que el contenedor **server6** está utilizando la dirección IP 172.17.0.2

Actividad 8

OBJETIVO: Realizar el despliegue manual de múltiples componentes sin que haya que reconstruir las imágenes Docker a utilizar.

ENUNCIADO:

La Actividad 7 y los dos últimos apartados de la Actividad 6 han mostrado que se pueden construir imágenes Docker que admitan cierta reconfiguración dinámica gracias al uso de variables de entorno (Actividad 7) o instrucciones ENTRYPOINT y CMD en el Dockerfile, combinadas con argumentos de línea de órdenes en **docker run** (Actividad 6).

a) Utilice ENTRYPOINT / CMD para generar imágenes **client** y **worker** que no necesiten ser regeneradas en el despliegue manual descrito en la Sección 9.1 de la presentación del Tema 4 (o en este vídeo: "Docker: Despliegue manual de múltiples componentes"). Proporcione los ficheros Dockerfile resultantes para ambos componentes (cliente y worker) y explique cómo deben generarse e iniciarse los tres componentes, facilitando las órdenes **docker build** y **docker run** necesarias para ello.

b) Utilice ahora variables de entorno para resolver ese mismo problema. Observe que los ficheros Dockerfile necesarios para ello ya han sido presentados en la Sección 9.2 del Tema 4. Explique cómo se iniciarán manualmente los tres componentes utilizando las nuevas imágenes. Proporcione también las órdenes **docker run** a utilizar.

Actividad 9

OBJETIVO: Modificar y utilizar adecuadamente las cláusulas del fichero **docker-compose.yml** para configurar un despliegue automatizado de servicios.

ENUNCIADO:

Considérese el despliegue automatizado descrito en la Sección 9.2 de la presentación del Tema 4 (o en este vídeo: "Docker-compose: Despliegue de múltiples componentes"). Este es su fichero **docker-compose.yml**:

```
version: '2'
services:
    cli:
        image: client
        build: ./client/
        links:
            - bro
        environment:
            - BROKER_URL=tcp://bro:9998
    bro:
        image: broker
        build: ./broker/
        expose:
            - "9998"
            - "9999"
    wor:
        image: worker
        build: ./worker/
        links:
            - bro
        environment:
            - BROKER_URL=tcp://bro:9999
```

Copie todos los programas y ficheros Dockerfile necesarios en las carpetas citadas en esa presentación. Resuelva los apartados siguientes:

a) Las cláusulas **image:** y **build:** tienen objetivos similares: asegurar que las imágenes a utilizar existen. Explique qué ocurre si eliminamos alguna de esas dos cláusulas en algún componente.

b) Supongamos que en nuestro despliegue basta con una instancia **worker**. Debido a eso, ese programa worker.js hará un bind() de su socket ZeroMQ sobre el puerto correspondiente y el programa broker.js se conectará al URL resultante.
Explique cómo debe modificarse este fichero **docker-compose.yml** para que el orden de inicio de los componentes pase a ser: worker (primero), broker, cliente (último). Proporcione el fichero resultante.
Explique también qué modificaciones habrá que realizar en los ficheros Dockerfile de las imágenes **worker** y **broker**.