Entornos uniformes de desarrollo de software

Norberto Gaspar Cena, Ignacio Daniel Favro, Sebastián Norberto Mussetta
Departamento de Ingeniería en Sistemas de Información
Universidad Tecnológica Nacional, Facultad Regional Villa María
Avenida Universidad 450, Villa María, Córdoba, Argentina
{ngcena, idfavro, smussetta}@frvm.utn.edu.ar

Abstract

El presente documento analiza la complejidad y dificultad que yace en los despliegues individuales de los equipos de trabajo durante la fase de desarrollo por diferentes factores que afectan a la uniformidad de los entornos de prueba, fundamentalmente enfocado a sistemas cliente-servidor y finalmente soluciones existentes.

1. Introducción

Un desafío que suelen enfrentar en la actualidad los equipos de desarrollo sin importar su tamaño, es la dificultad que presenta la diversidad de entornos donde se realizan los despliegues del sistema durante las fases de prueba, dejando como saldo un cúmulo situaciones problemáticas donde el código producido por un integrante del equipo funciona correctamente es su dispositivo de trabajo, no produce los mismos resultados en otros dispositivos, acudiendo a frases tristemente célebres como "en mi dispositivo funciona bien" o "hace instantes funcionaba". Este trabajo pretende aportar un bosquejo de muchas situaciones que se presentan durante el desarrollo de software al momento de realizar despliegues de prueba por parte de los equipos de desarrollo, sus inconvenientes y tecnologías desarrolladas que proveen mitigaciones, optimizan e incluso puedan solucionar estos casos cotidianos. El presente documento se encuentra estructurado de la siguiente manera: la sección II analizará los distintos escenarios que ejemplifican la problemática a tratar. En la sección III se planteará de manera conceptual métodos de confrontar la problemática. La sección IV presenta de forma comparativa implementaciones existentes de los métodos conceptuales descriptos durante la tercera sección. Por último se detallan las conclusiones obtenidas en la sección V.

2. Escenarios y Problemática

Wlodzimierz Gajda en su libro Pro Vagrant[1] describe dos enfoques tradicionales de creación de entornos de desarrollo y advierte sobre los inconvenientes que estos presentan. El primero donde el desarrollador instala todo el software es su estación de trabajo, y tanto el backend como el frontend del software funcionan localmente como se muestra en la Figura 1.

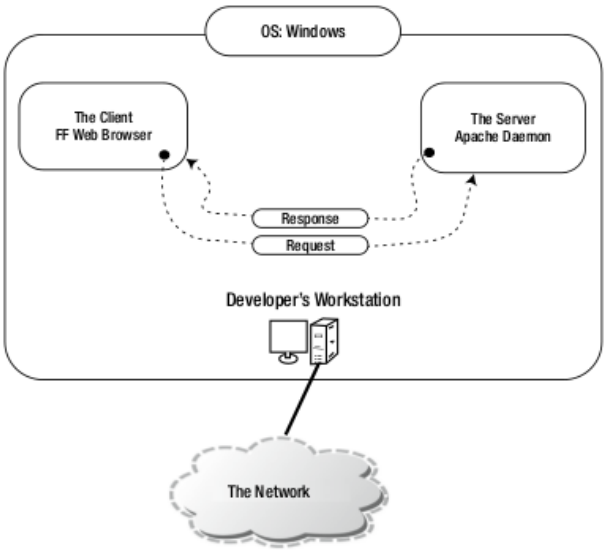

Figura 1. Primer escenario, donde todo el software se instala en la estación de trabajo del desarrollador

El segundo enfoque es donde el cliente y el servidor corren en diferentes máquinas, emulando el modo de despliegue que posteriormente utilizarán los usuarios finales para ejecutaran el software. Para este caso el software cliente funciona en la estación de trabajo del desarrollador y el componente servidor corre en el dispositivo remoto, y dependiendo de las preferencias, el desarrollador puede optar por los siguientes escenarios:

- Sincronizar los archivos mediante el entorno de desarrollo integrado (IDE por sus siglas en inglés) como se muestra en la Figura 2.

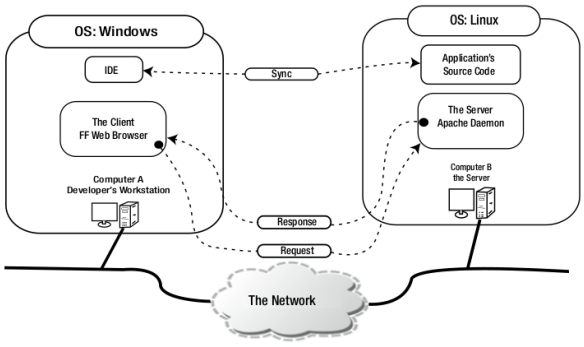

Figura 2. Escenario donde el cliente y el servidor corren en diferentes máquinas y el código fuente es sincronizado con el IDE

- Otra opción es transferir el código por una terminal de comandos remota como por ejemplo puede ser SSH y utilizando un editor de textos como vi/vim esquematizado en la Figura 3.

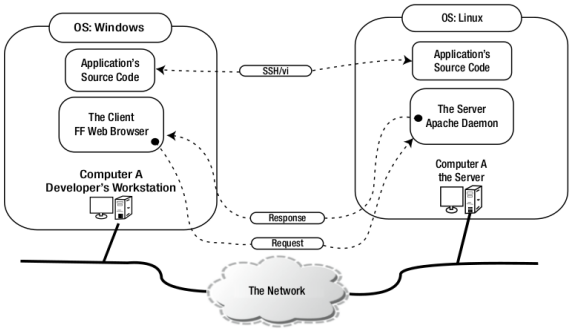

Figura 3. Escenario donde el cliente y servidor corren en distintas máquinas y el código es ingresado por un terminal de comandos.

- Por último, otra opción es emular el acceso local mediante el mapeo de recursos remotos.

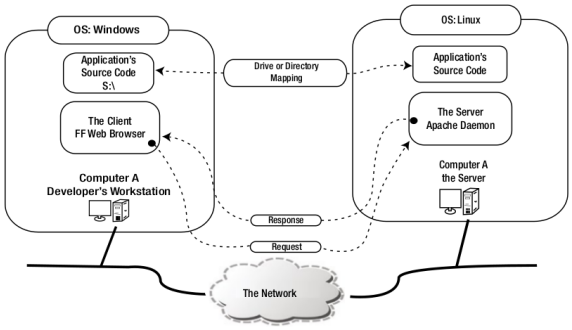

Figura 4. Emulación de acceso local mediante el mapeo de recursos remotos.

Estos escenarios si bien son funcionalmente correctos, presentan inconvenientes, enumerados a continuación. Para el caso del primer escenario donde el tanto el software cliente como servidor funcionan en la estación de trabajo del desarrollador:

- Cada desarrollador debe instalar el software que de servicios manualmente, siendo tiempo de desarrollo desaprovechado.
- Si cada desarrollador puede optar por su propio sistema operativo, se dificulta configurar entornos idénticos, sin mencionar casos donde es imposible debido a restricciones de arquitectura de software.
- Las diferencias de sistema operativo no sólo es entre desarrolladores, sino también entre desarrolladores y el entorno de producción.
- Todos los miembros del equipo deben tener conocimiento del software necesario para que la aplicación corra y como poner en funcionamiento el entorno.
- Desarrolladores que vinculados a distintos proyectos puede tener inconvenientes debido a la incompatibilidad de diferentes versiones de un lenguaje o producto y donde no es posible tener ambas.

Para los demás escenarios se encuentran las siguientes desventajas:

- Todos los desarrolladores requieren de conectividad a la red para trabajar y el rendimiento del servidor también afecta a todo el equipo de trabajo.
- Ante una caída de la red, todos los desarrolladores quedan ociosos.
- Algunas organizaciones con fuertes restricciones de seguridad impiden el acceso a los servidores desde el exterior, prohibiendo a colaboradores acceder a los entornos de desarrollo y trabajar de forma remota.

A su vez Edward H. Bersoff en su trabajo Elements of the Software Configuration[2] resalta la importancia de la gestión de la configuración (GC), donde involucra distintos elementos del software como documentación, código fuente y librerías entre otros, donde podría incluirse el entorno de desarrollo; Bersoff describe a la GC como instantáneas de estos componentes a lo largo del proceso evolutivo del software, que van desde un estado íntegro del producto a otro, a su vez describe los aspectos de la integridad del producto. Estos escenarios,

son controlados por los desarrolladores, y por ende es muy complejo realizar GC de este elemento, cuando no imposible.

Las situaciones descriptas no son ajenas a ningún equipo de desarrollo, y generan no sólo estos inconvenientes, sino otros que quizá escapan a este análisis, pero que a los fines de la descripción de la problemática son suficientes. Finalmente es importante destacar los múltiples efectos negativos como demoras en los proyectos, el desaprovechamiento de recursos humanos en tareas complementarias no productivas y las frustraciones que estas a veces generan.

3. Métodos de confrontación

Un buen método de lograr uniformidad del entorno de desarrollo debe abarcar las siguientes características deseables:

- Independencia de sistema operativo que utilice el desarrollador.
- Configuración automatizada y reproducible. De este modo se evita la necesidad de conocimientos específicos e innecesarios por parte del desarrollador, y siendo reproducibles logramos la uniformidad.
- Emular el entorno de producción.
- Posibilitar el aislamiento del entorno, logrando de este modo evitar dejar expuesto al mundo exterior información sensible.
- Posibilitar la generación de múltiples entornos distintos en una misma estación de trabajo.

Una vez definidas las principales características, una primera aproximación en surgir es la virtualización, es decir simular un nuevo hardware donde sea posible desplegar todos los servicios requeridos y de esta forma atacar la independencia de SO, emulación del sistema en producción, el aislamiento del entorno y la generación de múltiples entornos. ¿Pero qué pasa con la automatización y reproducibilidad? Si bien es posible copiar o exportar las virtualizaciones creadas ¿Qué sucede ante un cambio del software de servicio necesario?, automáticamente quedan obsoletas las máquinas previamente distribuidas, siendo necesario notificar a todos los desarrolladores y redistribuir nuevamente el servidor virtual, algo que al menos se puede adjetivar como incómodo o molesto.

Una buena solución a esto es poder crear una simple receta describiendo que recursos de hardware, sistema operativo, el software necesario y los pasos a seguir para combinar estos y que automáticamente se genere la máquina virtual requerida siguiendo estas instrucciones. La potencia de este método yace en que su simpleza, permitiendo distribuir simplemente la receta y no el producto, pudiendo utilizar medios como correo electrónico, almacenamiento en la nube e incluso sistemas de control de versión (VCS por sus siglas en inglés) que permitan mantener un histórico del servidor asociado a cada versión del sistema entregable, acompañando la iteración del desarrollo. Además cabe resaltar que este método evita la necesidad de los desarrolladores de conocer el cómo de configurar dicho entorno.

Podemos ejemplificar esto de la siguiente manera, el entorno de servidor requiere 512Mb de RAM un sólo CPU, 20GB de disco y se debe poder acceder al puerto 80 del mismo, asimismo es necesario que el sistema operativo sea Debian 8, con una base de datos PostgreSQL 9.4, apache 2.4 con el módulo php 5 y por último al iniciar debe obtener el código a ejecutar de un repositorio. Si bien es breve, abarca todas las necesidades incluso una muy destacable que es tareas que deben ejecutarse al inicio. Esto es lo que se espera de un método de confrontación integral a la problemática.

4. Comparativa, descripción y demostración de implementaciones existente

Existen dos implementaciones que se analizarán durante esta sección, que si bien su fin es prácticamente el mismo, el contexto de utilización difiere considerablemente acorde a la necesidad, estas son Docker y Vagrant. La primera diferencia y quizá la más pronunciada es su técnica de virtualización, Vagrant crea máquinas virtuales completas en base a una serie de instrucciones y definiciones en un archivo de configuración; por otro lado Docker implementa lo que se conoce como contendores, los contenedores simplemente agregan capas sobre el kernel del sistema operativo que están totalmente aislados del sistema operativo base mediante la utilización de lo que se conoce como espacios de nombre; en comparación los contenedores consumen mucho menos recursos y su inicio es más rápido pero no emulan una máquina real y puede que esta sea una característica deseable en ciertos entornos de desarrollo, tampoco pueden generarse contenedores que no sean basados en Linux. Sumado a esto, la esencia de Docker es comunicar contenedores entre sí para generar funcionalidades complejas. Finalmente ambas soluciones poseen un repositorio de imágenes, los cuales son plantillas que satisfacen ciertas necesidades y que además pueden ser modificadas, mejoradas o ampliadas.

Vagrant es un cliente de alto nivel que gestiona una variedad de proveedores de servicio de máquina virtual como VirtualBox, VMWare, XEN, Hyper-V y Amazon Web Services entre los más significativos; aunque es recomendable y principalmente soportado para VirtualBox. Además Vagrant incluye el concepto de aprovisionamiento, que son todas aquellas actividades que deben realizarse sobre la máquina virtual una vez creada a fines de generar y dejar funcional el entorno de desarrollo. Este aprovisionamiento puede ser llevado a cabo tanto mediante un simple script o por herramientas diseñadas específicamente para esta tarea; alguna de estas herramientas son Ansible, Chef, Puppet e incluso el ya nombrado Docker.

Ya presentado Vagrant se mostrará su potencial mediante ejemplos aplicables a una variedad de dificultades de todo desarrollo. Vamos a tomar como caso un equipo de desarrollo donde el software servidor necesita utilizar el framework Django 1.9 con python3 y un motor de base de datos Postgresql 9.4, además para que el sistema emule de mejor manera el entorno de producción es deseable que corra sobre Ubuntu como sistema operativo. Se omiten los pasos para su instalación ya que son muy sencillos y pueden ser consultados por el lector en el sitio web oficial[3]. Vagrant utiliza un archivo llamado Vagrantfile[4] cuyo funcionamiento principal es describir el tipo de maquina requerida para un proyecto, y como esta se debe configurar y aprovisionar. Debe existir necesariamente un archivo Vagrantfile por cada proyecto. Nuestro Vagrantfile para esta situación sería como se muestra en la Figura 5:

```
1  Vagrant.configure(2) do |config|
2    config.vm.box = "ubuntu/trusty64"
3    config.vm.network :forwarded_port, guest: 8000, host: 8080, host_ip: "127.0.0.1"
4
5  $script = <<SCRIPT
6
7    aptitude update -y
8    aptitude install -y python3-pip python3-dev libpq-dev postgresql
9    pip3 install Django==1.9 psycopg2
10   cd /vagrant/mi_proyecto
11   python3 manage.py runserver
12
13 SCRIPT
14
15   config.vm.provision "shell", inline: $script, run: "always"
16
17   config.vm.post_up_message = "Su servidor esta completo, puede
18     acceder al mismo mediante vagrant ssh de ser necesario"
19 end
20
```

Figura 5. Ejemplo de Vagrantfile para Ubuntu/Django/PostgreSQL.

Analizando la configuración se observa lo siguiente: la línea 1 simplemente da comienzo a la configuración para este proyecto de Vagrant e identifica la versión de configuración a utilizar (versión 2 al momento de realización de este documento); en la línea 2 se configura que empaquetado se utilizara, en este caso se obtiene uno del catálogo provisto por el repositorio oficial de Vagrant[5]; la línea 3 por su parte configura una redirección de puerto desde la maquina anfitrión a la máquina virtual; de la línea 5 a 13 son un listado de comandos que se asignan a la variable $script, estos comandos son propios de sistemas basados en Debian y efectuan la instalación de los requisitos del entorno de desarrollo; en la línea 15 se configura el aprovisionamiento de la máquina virtual, que se ejecuta cada vez que inicia la máquina debido al parámetro "run: 'always' ", el cual de omitirse sólo se ejecuta durante el primer inicio, también puede notarse el uso de la palabra reservada "inline" que especifica que el aprovisionamiento se hace en línea y utilizando la variable $script previamente definida; la línea 17 simplemente emite un mensaje cuando se llevaron a cabo todas las tareas correctamente y la máquina se encuentra en funcionamiento; por último se finaliza la configuración mediante la palabra reservada "end".

Otra configuración a resaltar que Vagrant provee a la máquina virtual es una directorio compartido en /vagrant, esto es visible en la línea 10 donde nos posicionamos en /vagrant/mi_proyecto el cual contiene el código a ejecutar. Es posible también configurar directorios personalizados.

Como se describió al inicio de la sección, Vagrant utiliza por defecto Virtualbox como proveedor de máquina virtual, con lo cual luego podemos realizar modificaciones de ser necesario como lo haríamos de forma cotidiana, aunque esto a los fines de la GC se desaconseja totalmente.

Puede ser necesario máquinas virtuales sumamente específicas o de configuraciones complejas, para ello existe la opción de empaquetar máquinas propias creadas y distribuirlas; si bien este tópico escapa a los alcances del documento, de forma simplista se puede decir que para realizar esto se debe modificar la línea 2 por config.vm.box_url la cual apunta al sitio donde se encuentra disponible para descargar (se omite la creación).

En lo que a utilización respecta, el usuario puede acceder vía ssh mediante el comando vagrant ssh, también existe una opción homologa para máquinas windows mediante WinRM.

Docker por su parte utiliza contenedores como se expresa con anterioridad, que si bien no son máquinas virtuales corren distribuciones linux (y solo distribuciones linux, aunque Docker corre en Windows, Mac o Linux) y encapsulan sus procesos aislando y permitiendo la portabilidad de nuestros entornos entre otras ventajas que esto provee como la clusterización y la composición destacado por Sébastien Goasguen en su libro Docker Cookbook[6].

La fortaleza de Docker yace no sólo en las características del párrafo precedente, sino en la enorme comunidad que provee una infinidad de soluciones que se pueden ajustar a nuestras necesidades, y la posibilidad de interconectar contenedores nos lleva a mix de posibilidades totalmente flexibles a un bajo costo.

Obviamente comparte con Vagrant esta característica de fabricar recetas que puedan ser distribuidas en integradas a la GC.

Antes de comenzar es importante establecer dos términos fundamentales de Docker a fines de evitar confusión; que son imagen y contenedor, una imagen es una plantilla solo lectura que luego es instanciado en lectura y escritura en un contenedor que podría decirse que es el cual realmente corre los procesos.

Una buena forma de establecer una comparativa es generar el mismo entorno creado por Vagrant, el archivo de creación de imagen el cual convenientemente se llama Dockerfile sería el que se denota en la Figura 6.

```
1  FROM ubuntu:14.04
2
3  RUN apt-get update && \
4      apt-get install -y python3-pip python3-dev libpq-dev postgresql
5  RUN pip3 install Django==1.9 psycopg2
6
```

Figura 6. Archivo Dockerfile de creación de imagen.

Nuestro Dockerfile generará una imagen con el entorno requerido para correr nuestro software, luego por parte de los desarrolladores resta crear el contenedor y vincular el directorio donde se encuentra la aplicación; para ello se debe ejecutar el comando de la Figura 7.

```
docker run -v /directorio/proyecto:/directorio/en/contenedor \
           -d -p 8000:8000 msi:is
```

Figura 7. Comando de creación del contenedor.

El comando no solo crea un contenedor, sino que también lo inicia por primera vez, luego podemos para e iniciar nuevamente mediante el comando docker start y docker stop. Analizando el comando vemos una serie de parámetros; el primero -v que monta el directorio /directorio/proyecto de la máquina anfitrión en el contenedor, el -d simplemente corre el contenedor en segundo plano, y por último el -p 8000:8000 que su función yace en vincular el puerto 8000 de la máquina anfitrión al 8000 del contenedor. En el contenedor es persistente toda la información que generemos o modifiquemos, y no es necesario los parámetros nuevamente al momento de iniciar el contenedor.

La imagen puede ser distribuida tanto por servicios propios como la utilización gratuita del hub de Docker[7] donde además se encuentran disponibles muchas otras públicas y que pueden ser de nuestra utilidad.

Otra opción es utilizar dos contenedores e interconectarlos, uno con Postgres y otro con Django, pero esto rompe con nuestro objetivo de uniformidad de entorno.

5. Conclusiones

Tanto Vagrant como Docker dan simples muestras de una buena práctica para uniformar entornos de desarrollo, y su potencial es mucho mayor a lo que el presente documento aborda, sumado a la posibilidad integrar estos entre sí o con herramientas como Chef donde se genera un ambiente sumamente automatizado y uniforme en todas las etapas del desarrollo; Oskar Hane[8] deja una buena conclusión sobre Docker y que puede ser extrapolada a Vagrant que cito "ayuda a los desarrolladores a focalizar en el desarrollo y no en configurar servidores y otras operaciones de DevOps".

A su vez, es posible gestionar estas herramientas con ayuda de un VCS como git siendo una solución completa y que amerita el esfuerzo inicial invertido.

Por último no es un detalle menor resaltar, que tanto el archivo de configuración de Vagrant, como una imagen de Docker representan un ítem de la GC, donde se reflejan los cambios realizados y su evolución en el tiempo; también puede ser distribuido como tal a todas las personas vinculadas al proyecto y no solo al equipo de desarrollo a los fines de realizar las tareas necesarias sobre él, como ejecuciones de testing, validación y verificación, y obviamente utilización a los fines de las pruebas del software por parte de los programadores. De este modo simplificamos la uniformidad en el entorno de desarrollo y evitamos que cada persona vinculada al proyecto genere su propia versión del entorno y los problemas que esto pueda generar.

6. References

[1] W. Gajda, "Pro Vagrant" 1ra ed., Apress, junio de 2015.

[2] E. Bersoff, "Elements of Software Configuration Management" Software Engineering IEEE Transactions on, Vol. SE-10, Issue 1, pp79-87, enero 1984.

[3] Installing Vagrant, https://docs.vagrantup.com/v2/installation/index.html (15 de julio de 2016)

[4] Vagrantfile, https://docs.vagrantup.com/v2/vagrantfile/index.html (15 de julio de 2016)

[5] Public Vagrant box catalog, https://atlas.hashicorp.com/boxes/search (17 de julio de 2016)

[6] S. Goasguen, "Docker Cookbook" 1ra ed., O'Reilly Media, noviembre de 2015

[7] Docker Hub, https://hub.docker.com/explore/ (23 de julio de 2016)

[8] O. Hane, "Build Your Own PaaS with Docker", Packt