METEOPLAY, S.L.

Servidor WS

Documentación

Julio B.M.

27/08/2014

WS (Web Services) es un servidor de producción destinado a ejecutar aplicaciones web elaboradas en Python, junto con el framework web Flask. Este documento es una guía rápida de configuración. El contenido está orientado a administración de sistemas, no a las aplicaciones en sí mismas.

Índice

Introducción

WS (*Web Services*) es un servidor de producción destinado a ejecutar aplicaciones web elaboradas en Python, junto con el framework web Flask.

El servidor de desarrollo de Flask (Werkzeug) no es suficiente, por las siguientes razones:

- El rendimiento es mucho menor que lo utilizado en WS: Nginx y uWSGI. Además, se podría colgar con tráfico intenso, y habría que reiniciarlo manualmente.
- No se permiten varias aplicaciones a la vez en el mismo puerto. Habría que cambiar el tradicional puerto 80 de HTTP por otro, y para cada aplicación. Esto es un poco cutre.
- Su prioridad no es la seguridad, sino el desarrollo.

Este documento es una guía rápida de configuración. No es un manual de instalación[1]. He intentado explicar todo con el suficiente detalle como para que alguien sin ninguna experiencia en desarrollo web con Python pueda hacerse una idea inicial. Si el lector desea profundizar, le invito a navegar por los enlaces (en azul y subrayados). El contenido está orientado a administración de sistemas, no a las aplicaciones en sí mismas. Se comentará algún detalle de la aplicación si fuese necesario. De momento sólo hay una instalada, la de los *widgets* de Castilla-La Mancha, y otras dos de prueba.

Servidor

Características

- **Sistema operativo**: CentOS 6.5
- **Hostname**: ws (dominio local amd.local).
- **Dominio de Internet**: `www.meteoapps.com`.
- **IP asignada (reserva DHCP, ver tempesta1 [2])**: 212.81.192.246 (subred 212.81.192.240/28). Ésta es la subred pública que está *detrás* del Juniper ("DMZ"), no la que hay delante (subred 212.81.192.224/28), parte de la zona *untrust*, y que está *directamente conectada* al Mikrotik de nuestro ISP, Sarenet. Sarenet tiene una ruta hacia 212.81.192.240/28.
- **Network Label (ver vSphere)**: "Network_WAN" (VLAN 70).

El sistema operativo es CentOS 6.5. Era la versión más actual en el momento que lo instalé. No tiene instalados los VMware Tools porque para ESXi el soporte empieza en la versión 5.0 Update 2, y la nuestra es la 4.0.0.

Hay una partición primaria que ocupa todo el disco. Dentro hay varios volúmenes lógicos (LVM). Son: `/`, `swap`, `/var`, `/home`, `/tmp`, `/var/log`. Los volúmenes lógicos son más ventajosos que las particiones tradicionales. Entre otras cosas, permiten cambiar su tamaño sin necesidad de herramientas externas. `/var` está en su propio volumen por si se monta una

[1] Tanto uWSGI como Nginx se pueden instalar desde los repositorios. Lo único que hay que compilar es Python 2.7.6, porque CentOS 6.5 sólo viene con la versión 2.6.6. En la sección Compilación de Python 2.7.6 menciono la guía que he seguido.

[2] Tempesta1 está en VLAN30, y WS en VLAN70. DHCP puede pasar de una VLAN a otra gracias a DHCP Relay (IP Helper) en el switch HP 1910.

caché de Nginx (`/var/nginx/cache`) y queremos aumentar la capacidad exclusivamente para este cometido. En `/var` se encuentran también las imágenes de la aplicación de (`/var/opt/apps/`).

`/var/log` también está aparte para contener algún posible fichero de log que crezca de manera descontrolada. De esta manera, sólo fallaría un volumen y los demás no se verían afectados. Lo mismo podemos decir de `/tmp`.

Otro beneficio de separar volúmenes es la posibilidad de formatear cada uno con sistemas de ficheros distintos, por si pudiese mejorar el rendimiento, aunque yo no he hecho esto (salvo para swap, claro).

El tipo de adaptador para la tarjeta de red es VMXNET3, por su rendimiento superior al hardware emulado Intel E1000. En principio haría falta el driver incluido en VMware tools, pero afortunadamente ya viene en CentOS. Había pensado añadir otros dos adaptadores para administración y base de datos, pero todavía no lo he conseguido. Ver "múltiples interfaces" en la sección "Problemas encontrados".

Aplicaciones instaladas

Azucarera

- `http://www.meteoapps.com/azucarera/azucarera.html`

Widgets

- `http://www.meteoapps.com/rtvcm/widgets/minimodulo-cabecera.html`
- `http://www.meteoapps.com/rtvcm/widgets/modulo-mapa.html`
- `http://www.meteoapps.com/rtvcm/widgets/modulo-mapas-grandes.html`

Aplicación de prueba 1

- `http://www.meteoapps.com/prueba/`

Aplicación de prueba 2

- `http://www.meteoapps.com/prueba2/`

Procedimiento de arranque

Simplemente arrancar la máquina virtual. El software necesario (Nginx y uWSGI, y Samba) inicia automáticamente.

Dependencias aplicación

Equipo	Producto	Programa	Origen	Destino
Eumetcast-RX1	Meteosat - IR	2met! User Station	Tc-recv (Eumetcast TELLICAST Multicast Distribution System Client 2.5.17)	\\eumetcast-1\d\2met\incoming
Eu	Me	2met! User Station	D:\2met\imagenes\IR_087	D:\2met\Processing\Product

meteocast-1	teosat-IR	(*Product: IR_87_EspanyaAZUCARERA*)	(IR_087 significa *InfraRed canal 087*)	s \\sirius\Recursos\mapes\<Territorio>\Meteosat\TipusAEASA
Grads-1	Radar	`/home/sam/radar/scripts/radaresregionales2014.sh`		\\sirius\Recursos\Mapes\<Territorio>\Radar\
	Meteosat-IR	`/home/sam/satelite/a.sh`	\\sirius\Recursos\Mapes\<Territorio>\Meteosat\TipusEspanaAEASA4x3\jpg	\\sirius\Recursos\Mapes\<Territorio>\Meteosat\TipusEspanaAEASA4x3\jpg_2_grads-1
Ent-11	Meteosat-IR	Exportador_Meteo	\\sirius\Recursos\Mapes\<Territorio>\Meteosat\TipusEspanaAEASA4x3_Reduit	\\ws\a\meteosat-ir\<Territorio>
	Radar		\\sirius.amd.local\Recursos\Mapes\Espanya\Radar\aa\	\\ws\a\radar\
		SamDelFilesE3	\\sirius\Recursos\Mapes\Espanya\Meteosat\TipusEspanaAEASA4x3\jpg \\sirius\Recursos\Mapes\Europa\Meteosat\TipusEuropaAEASA4x3\jpg \\ws\a\ (*SubDirectoris* a True). \\sirius\Recursos\Mapes\Espanya\Meteosat\TipusEspanaAEASA4x3\jpg_2_grads-1 \\sirius\Recursos\Mapes\Europa\Meteosat\TipusEuropaAEASA4x3\jpg_2_grads-1	
Ent-6	Radar	SamDelFiles (*por Oriol*)	\\Sirius\Recursos\Mapes\Espanya\Radar\ (*SubDirectoris* a True).	

Aplicaciones web

Breve introducción

Una página web dinámica es aquella generada mediante programas o scripts ejecutados en el servidor, en contraposición contenido estático tradicional. Con programas me refiero a código binario compilado (C, C++), y con scripts a lenguajes interpretados (ASP, Perl, PHP, Python, Ruby), aunque a partir de ahora usaré uno u otro término indistintamente. Han experimentado un gran desarrollo desde su inicio hace veinte años. Son esenciales para prestar servicios al público en general. Sin embargo, también se utilizan internamente en empresas, porque evitan la instalación de programas de escritorio en cada equipo. Sólo se necesita un explorador web.

En principio, un servidor web sólo se encarga de procesar y transmitir páginas web HTML a través del protocolo HTTP. Por sí mismo no puede ejecutar nada. Necesita algún tipo de interfaz, como por ejemplo CGI, FastCGI, SCGI, o AJP. Puede que un mismo script pueda utilizar dos interfaces distintas. Por ejemplo, PHP puede usar CGI, y, en el caso del servidor web Apache, el módulo mod_php.

En la siguiente sección veremos qué lenguaje e interfaz he elegido para las aplicaciones.

Nuestro caso

Por el momento, las aplicaciones instaladas en el servidor WS están programadas en Python, utilizando un framework web llamado Flask. Cuando empecé a trabajar, también conocía otro, Django. Sin embargo, Flask parece más apropiado para proyectos pequeños, como los que tengo asignados. Hace poco me he enterado de la existencia de Bottle, aunque todavía no lo he utilizado.

El diseño de la aplicación se basa en un servicio web de tipo REST (REpresentational State Transfer). En este enlace se explica REST y se describe su implementación en Flask. Básicamente, un cliente accede a una URL y se le envía una respuesta. Las URLs forman parte de la especificación del "API" que estemos desarrollando. Por ejemplo, en el "widget" de Castilla-La Mancha, al abrir la URL http://www.meteoapps.com/azucarera/todo/api/v1.0/relacion-simbolos en el explorador, aparece esto:


```
{
  "relacionSimbolos": {
    "granizo": "Z+",
    "llovizna": "M M+ M- N N+ N- O O+ O-",
    "lluvia": "P P+ P- Q Q+ Q-",
    "neblina": "F F+ F-",
    "niebla": "G G+ G-",
    "nieve": "V V+ V- W W+ W- Z",
    "nieve_sol": "T T+ T- U U+ U-",
    "nube": "H H+ H- I I+",
    "nube_alta_sol": "E E+ E-",
    "nube_sol": "C C+ C- D D+ D- I- J J+ J- K K+ K- L L+ L-",
    "sol": "A A+ A- B B+ B-",
    "tormenta": "S S+ S-",
    "tormenta_sol": "R R+ R-"
  }
}
```


Lo que se nos muestra no es una página web, es una respuesta JSON. Un cliente, que en nuestro caso es JavaScript, consume este contenido. Esto se realiza de manera asíncrona (Ajax), por lo que la página no se recarga.

Flask, como otros tantos frameworks web para Python, utiliza internamente un API llamado WSGI. Sin embargo, no es posible la comunicación directa entre WSGI y un servidor web. Para ello, los servidores que implementan WSGI utilizan protocolos a más bajo nivel. Por ejemplo, Flup soporta FastCGI, SCGI, y también AJP. El servidor uWSGI soporta FastCGI, SCGI, y también su propio protocolo, el protocolo uwsgi. Hay que tener cuidado de no confundir el servidor con el protocolo, porque tienen el mismo nombre. Note el lector que el servidor se escribe con 4 letras mayúsculas mientras que en el protocolo son siempre minúsculas.

Yo utilizo el servidor uWSGI con su protocolo nativo, el protocolo uwsgi. Mis razones para esta elección son:

- Modo emperador: podemos lanzar varias aplicaciones python, llamadas vassals, controladas por una instancia uWSGI, el Emperador. El Emperador inicia los vassals y los reinicia en caso de cuelgue. Más adelante se verá la configuración en detalle.
- Rendimiento superior a otros servidores.

Podemos usar el servidor uWSGI como servidor web, pero para un mayor rendimiento se aconseja situarlo detrás de un proxy inverso. Tanto Cherokee y Nginx soportan nativamente el protocolo uwsgi. He escogido Nginx porque las guías que veía en internet utilizaban éste. Realmente, no he investigado Cherokee. En la documentación de uWSGI se pueden ver más opciones.

La Figura 1 ilustra lo explicado hasta ahora[3]:

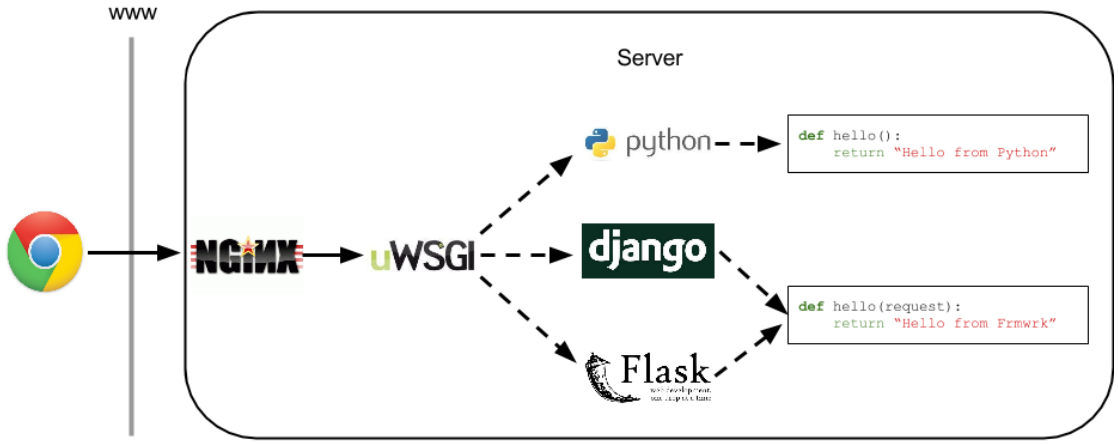

Figura 1

Observemos que uWSGI se puede comunicar tanto con python directamente como con frameworks web.

uWSGI

Vamos a ver la configuración de uWSGI antes que la de Nginx.

Como ya se ha comentado anteriormente, utilizo uWSGI en modo emperador. Recordemos que éste es una instancia uWSGI especial que controla a los vassals. La configuración de los vassals se encuentra en `/etc/uwsgi/vassals`. Cuando iniciamos el emperador, carga automáticamente todos los ficheros de configuración que encuentre, que pueden tener diversas extensiones: `.ini`, `.xml`, `.yml`, `.json`, etc. Si paramos el emperador, se paran todos los vassals. Si un vassal se bloquea, el emperador lo reinicia.

Veamos la configuración para los wdgets de Castilla-La Mancha (fichero `/etc/uwsgi/vassals/widget_cm_76_produccion.ini`):

```
[uwsgi]
chdir = /webapps/widget_cm_76_produccion
```

[3] Extraída de http://brianmcdonnell.github.io/pycon_ie_2013/#/3

```
virtualenv = widget_cm_venv
socket = /tmp/uwsgi_cm.sock
# Tengo que dar permisos al socket para que nginx pueda acceder a él.
De lo contrario, veo este error en /var/log/nginx/error.log:
# 2014/08/18 11:15:22 [crit] 25664#0: *4 connect() to
unix:///tmp/uwsgi.sock failed (13: Permission denied) while connecting
to upstream, client: 172.27.0.10, server: _, request: "GET
/prueba/poweredby.png HTTP/1.1", upstream:
"uwsgi://unix:///tmp/uwsgi.sock:", host: "212.81.192.246", referrer:
"http://212.81.192.246/prueba/"
# Ver:
#     http://stackoverflow.com/questions/21820444/nginx-error-13-
permission-denied-while-connecting-to-upstream
#         http://stackoverflow.com/a/22640924
chown-socket = nginx:nginx
mount = /rtvcm/=rest-server.py
manage-script-name=true
callable = app
processes = 4
threads = 2
```

- **Chdir**: indicamos el directorio raíz de la aplicación, donde se encuentra el fichero principal rest-server.py.
- **Virtualenv**: Widget_cm_venv es el "entorno virtual" en el que se ejecuta la aplicación. Virtualenv crea una copia aislada de Python aislada de otros entornos virtuales. No sólo se restringe a Python en sí mismo, sino también a todos los módulos instalados. De esta manera, podemos tener en producción varios proyectos con diferentes versiones de Python, Flask, etc. Esto es fundamental, porque nos permite utilizar versiones posteriores sin miedo a que aplicaciones anteriores fallen por alguna incompatibilidad. Esto es lo que ocurre entre Python 3 y Python 2, por ejemplo.
- **Socket**: fichero socket de Unix. Es el mecanismo por el que Nginx se comunica con el servidor uWSGI, a través del protocolo uwsgi. En lugar de un fichero socket, podríamos emplear la IP o nombre del servidor seguido del puerto (para nosotros localhost, 127.0.0.1). Sin embargo, en nuestro caso el servidor uWSGI está en la misma máquina virtual que Nginx, y en esta situación un socket de Unix supera en rendimiento a uno TCP/IP.
- **Chown-socket**: el propietario y grupo del fichero socket, separados por dos puntos. En muchas guías he visto www-data:www-data. En varias distribuciones Linux éste es el usuario sobre el que se ejecuta Apache. Sin embargo, no existe en CentOS, así que he empleado el usuario de Nginx.
- **Chmod-socket**: permisos 660 (rw-rw----). Por seguridad creo que es mejor que "others" no tenga permisos de ningún tipo.
- **Mount y manage-script-true**: El path asignado a la aplicación será /rtvcm/. Se cargará el archivo rest-server.py (sólo la primera vez que se cargue el vassal).
- **Callable**: el objeto "aplicación" en Flask es la función que WSGI necesita llamar. Su nombre es "app". El servidor de desarrollo (app.run(…)) dentro de if __name__ == '__main__': se ignora. Éste sólo funciona si ejecutamos directamente python rest-server.py.
- **Processes**: número de procesos.

- **Threads**: Número de hilos por proceso. Cada hilo puede atender una petición, por lo que en total se pueden atender procesos*hilos peticiones concurrentes.

La elección del número de procesos e hilos depende del sistema. Se tarda más en iniciar un proceso que un hilo, pero por lo visto los procesos son más robustos. En muchas guías he visto esa misma cantidad de 4 procesos y dos hilos por proceso, pero como se explica en este enlace, son valores orientativos que dependerán de la potencia de la máquina. Por lo visto se puede investigar esto con uwsgitop y newrelic.

Inicio automático

Uwsgi inicia automáticamente al arrancar la máquina. No hay un script de inicio System V, así que lo he puesto en /etc/rc.local, ejecutando en background (ampersand final):

```
/usr/local/bin/uwsgi --ini /etc/uwsgi/emperor.ini &
```

El contenido de emperor.ini es:

```
[uwsgi]
emperor = /etc/uwsgi/vassals
uid = nginx
gid = nginx
logto = /tmp/uwsgi.log
```

Emperor indica directorio donde se encuentra la configuración de los vassals. Uid y gid es el usuario y grupo con el que se va a ejecutar uWSGI. Finalmente, el directorio donde se guardará el log.

Reinicio de una aplicación concreta

Esto es útil si necesitamos actualizar el código. Sólo reiniciamos la aplicación deseada, sin afectar a las demás. Basta con un touch, tal y como se comenta en http://uwsgi-docs.readthedocs.org/en/latest/Emperor.html

```
touch –-no-dereference /etc/uwsgi/vassals/<aplicación>.ini
```

Creo que utiliza `–-no-dereference` por si `<aplicación>.ini` es un fichero simbólico.

En caso de querer actualizar el código, hay que subirlo previamente (con WinSCP, por ejemplo).

Driver ODBC en Linux

La aplicación de los widgets de Castilla-La Mancha extrae información del servidor floc.amd.local (instancia floc), que es una base de datos MSSQL (Microsoft SQL Server). Para ello, utiliza el módulo Python pyodbc. Pyodbc emplea ODBC, una especificación para comunicación con sistemas de gestión de base de datos (SGDB) independiente del desarrollador y sistema operativo. Pyodbc envía comandos ODBC a un driver ODBC, que se encarga de traducirlos a la sintaxis SQL concreta de cada base de datos. Para que esto funcione, se tiene que crear un driver ODBC para una SGDB en concreto (MySQL, MSSQL, PostgreSQL…).

En Windows ya vienen drivers ODBC incluidos en el sistema operativo para MSSQL. En Linux encontré estas opciones:

- Microsoft ODBC Driver 11 for SQL Server on Linux.
- EasySoft SQL Server ODBC Driver for Linux/Unix
- FreeTDS

El driver de Microsoft había que compilarlo, y además hacía falta una versión en concreto de una dependencia. EasySoft es soporta muchas características de MSSQL, y puede manejar hasta la versión 2014, pero es de pago. Finalmente, elegí FreeTDS porque, aparte de ser gratis (GNU LGPL), está en los repositorios y no tengo que compilar. Como curiosidad, aparte de ODBC, FreeTDS soporta las APIs db-lib y ct-lib.

En la guía de usuario de FreeTDS, se mencionan tres formas de configurar ODBC:

- Sin DSN: No utilizamos el fichero odbc.ini. Todo en freetds.conf. Esto permite utilizar otras características de FreeTDS que no son ODBC.
- Sólo ODBC: Todo en odbc.ini, sin freetds.conf. Bueno si sólo usamos la parte ODBC de FreeTDS.
- Combinado ODBC: la conexión se guarda en freetds.conf y el DSN en odbc.ini.

Como sólo voy a utilizar la parte ODBC de FreeTDS, escojo la opción "sólo ODBC".

Primero modificamos `/etc/odbcinst.ini`. Añadimos la definición del driver ODBC "FreeTDS"

```
[FreeTDS]
Description     = Free TDS driver
Driver64        = /usr/lib64/libtdsodbc.so.0
Setup64         = /usr/lib64/libtdsS.so
UsageCount      = 1
```

Ahora en `/etc/odbc.ini` hacemos referencia al driver que acabamos de definir:

```
[FLOC]
Driver      = FreeTDS
Server      = floc.amd.local\floc
TDS_Version = 8.0
```

NOTA: Odbc.ini estaba vacío antes de editarlo. Es normal.

Para terminar, la aplicación de los widgets cargará la sección [database] del fichero `rest-server.ini`, y se lo pasará a pyodbc:

```
[database]
dsn      = FLOC
user     = usr_am
password = 1suari_AM
```

Si el usuario o contraseña han cambiado, probablemente estén en KeePass.

Puertos dinámicos

Antes de configurar FreeTDS, hice algunas pruebas con el comando "tsql". No funcionaba porque el puerto especificado, 1433, no era donde la instancia estaba escuchando.

```
[root@ws ws]# TDSDUMPCONFIG=dump tsql -H floc.amd.local -p 1433 -U
usr_am -P 1suari_AM
locale is "en_US.UTF-8"
locale charset is "UTF-8"
using default charset "UTF-8"
Error 20009 (severity 9):
      Unable to connect: Adaptive Server is unavailable or does not
exist
      OS error 111, "Connection refused"
There was a problem connecting to the server


[root@ws ws]# cat dump
[…]


[root@ws ws]# rm -rf dump


[root@ws ws]# tsql -LH floc.amd.local
     ServerName FLOC
   InstanceName FLOC
    IsClustered Yes
        Version 9.00.4035.00
            tcp 1473
             np \\FLOC\pipe\MSSQL$FLOC\sql\query


[ws@ws ~]$ tsql -H floc.amd.local -p 1473 -U usr_am -P 1suari_AM
locale is "en_US.UTF-8"
locale charset is "UTF-8"
using default charset "UTF-8"
1> select @@version;
2> GO

Microsoft SQL Server 2005 - 9.00.4035.00 (Intel X86)
      Nov 24 2008 13:01:59
      Copyright (c) 1988-2005 Microsoft Corporation
      Enterprise Edition on Windows NT 5.2 (Build 3790: Service Pack
2)

(1 row affected)
1>
```

Sin embargo, el puerto es dinámico. Cada vez que se arranque la instancia puede cambiar. En la configuración de FreeTDS no es necesario poner ninguno. Con sólo escribir servidor\instancia lo descubre automáticamente.

Nginx

Al instalar Nginx desde los repositorios de CentOS se instala también un script de inicio System V, por lo que podemos pararlo, empezarlo y reiniciarlo, entre otras operaciones, mediante:

```
Usage: /etc/init.d/nginx {start|stop|reload|configtest|status|force-
reload|upgrade|restart|reopen_logs}
```

El archivo de configuración se encuentra en `/etc/nginx/conf.d/default.conf`. En este fichero encontramos "directivas" location, que cargan una "configuración" u otra según la URI pedida. Dentro de todos los location hay una directiva root, que establece el directorio raíz.

```
location /rtvcm/ {
    root /webapps/widget_cm_76_produccion/static;
    uwsgi_pass unix:///tmp/uwsgi_cm.sock;
    include uwsgi_params;
}
```

Un ejemplo: dentro de `/webapps/widget_cm_76_produccion/static` está [4] minimodulo-cabecera.html. Para acceder a él:

`<IP_o_nombre_de_WS>/rtvcm/minimodulo-cabecera.html`

A su vez, este html carga un estilo/estilo.css, lo que se traduce en

`<IP_o_nombre_de_WS>/rtvcm/estilo/estilo.css`

Arriba podemos ver la directiva uwsgi_pass, y a continuación una ruta a un fichero socket de Unix que ya mencionamos en la configuración de los vassals. El parámetro de uwsgi_pass también puede ser:

`<IP_o_nombre_de_servidor_uWSGI>(:<puerto>)`

… pero como también se comentó anteriormente, el rendimiento de un socket Unix es superior a uno TCP/IP.

Por supuesto, aparte de toda esta historia de aplicaciones Python, podemos seguir teniendo páginas estáticas de toda la vida. Por ejemplo, en la raíz tengo:

```
location / {
    root   /data/www;
    index  index.html index.htm;
}
```

Problemas encontrados

Múltiples interfaces

En principio había pensado poner tres interfaces: una para administración (ssh), otra para acceso a la base de datos, y otra pública. Las dos primeras estarían en la 172.27.0.0/19, mientras que la segunda en x.x.x.x/28 (IP x.x.x.x). Sin embargo, al activar la privada, la pública deja de funcionar. Pare que es el mismo problema que se comenta aquí. Sin embargo, todavía no me he puesto a mirarlo. De momento sólo he dejado la pública.

El inconveniente de tener sólo una interfaz es que hay que pasar por el Juniper para ir de la red pública a la privada o viceversa, lo que añade un poco de latencia:

[4] Otra buena ruta para las aplicaciones podría ser `/opt/apps`, como se menciona en la documentación de uWSGI Emperor.

```
[ws@ws ~]$ ping public-web.amd.local -c 100
PING public-web.amd.local (x.x.x.x) 56(84) bytes of data.
64 bytes from x.x.x.x: icmp_seq=1 ttl=128 time=1.27 ms
64 bytes from x.x.x.x: icmp_seq=2 ttl=128 time=0.306 ms
64 bytes from x.x.x.x: icmp_seq=3 ttl=128 time=0.300 ms
[…]
--- public-web.amd.local ping statistics ---
100 packets transmitted, 100 received, 0% packet loss, time 99006ms
rtt min/avg/max/mdev = 0.264/0.336/1.279/0.121 ms

[ws@ws ~]$ ping floc.amd.local -c 100
PING floc.amd.local (172.27.2.108) 56(84) bytes of data.
64 bytes from floc.amd.local (172.27.2.108): icmp_seq=1 ttl=126
time=0.779 ms
64 bytes from floc.amd.local (172.27.2.108): icmp_seq=2 ttl=126
time=0.904 ms
64 bytes from floc.amd.local (172.27.2.108): icmp_seq=3 ttl=126
time=0.876 ms

--- floc.amd.local ping statistics ---
100 packets transmitted, 100 received, 0% packet loss, time 99067ms
rtt min/avg/max/mdev = 0.741/1.190/4.771/0.871 ms
[ws@ws ~]$
```

Firewall de CentOS

Por defecto, CentOS viene con el firewall activado, y bloquea el puerto 80. Podemos permitir el puerto 80, o directamente deshabilitar el firewall, puesto que su función ya la realiza el Juniper SRX-240.

Firewall Juniper

Desde 172.27.0.0/19 podía acceder al servidor, pero desde fuera no. Comprobé esto con una de las muchas páginas “web proxy”. El Firewall de Juniper ya tenía una regla para permitir el tráfico desde XarxaDMZ (x.x.x.x/28) a la zona untrust (Internet):

```
        from-zone DMZ to-zone untrust {
            policy ServeisDMZ {
                match {
                    source-address XarxaDMZ;
                    destination-address any;
                    application [ junos-ident junos-smtp autodef-SQL
junos-http junos-https junos-tftp junos-ftp junos-icmp-all junos-dns-
udp junos-dns-tcp junos-ntp ];
                }
                then {
                    permit;
                    log {
                        session-init;
                        session-close;
                    }
                }
            }
```

Sin embargo, para el camino inverso hay reglas específicas por cada servidor, por lo que tuve que añadir una para el servidor WS:

Primero añadimos el servidor al address-book. Le he puesto el nombre “DMZ_WS”:

```
        security-zone DMZ {
```

```
            address-book {
                […]
                address DMZ_WS x.x.x.x/32;
            }
```

Ahora añadimos una regla “Allow_WS” desde la zona untrust a la DMZ

```
        from-zone untrust to-zone DMZ {

            policy Allow_WS {
                match {
                    source-address any;
                    destination-address DMZ_WS;
                    application [ junos-icmp-all junos-ssh junos-http
junos-https  ];
                }
                then {
                    permit;
                }
            }
```

URLs del API en cliente JavaScript

Cuando desarrollaba, el servidor de desarrollo de Flask (Werkzeug) estaba en la raíz, pero en Nginx y uwsgi hemos puesto la ruta /rtvcm/widgets. Por ejemplo, la antigua `/todo/api/v1.0/relacion-simbolos` es ahora `/rtvcm/widgets/todo/api/v1.0/relacion-simbolos`. Podría ir cambiando todas las URLs en el código del cliente JavaScript, pero lo he parametrizado. Siguiendo con el ejemplo anterior, APP_ROOT + `/todo/api/v1.0/relacion-simbolos`. APP_ROOT está en common.js. Podría establecer APP_ROOT=”/rtvcm”, pero si lo cambiamos en Nginx/uwsgi también habría que cambiarlo allí. Lo que he hecho ha sido:

```
var APP_ROOT = window.location.pathname.split('/').slice(0, -
1).join('/');
```

`window.location.pathname` devuelve la URL de la barra del explorador sin el dominio. Después lo dividimos por el carácter barra. Esto genera un array. A continuación acotamos desde el primero (0) hasta el penúltimo (-1). Finalmente unimos el array con el carácter barra. Por ejemplo, para http://www.meteoapps.com/rtvcm/widgets/minimodulo-cabecera.html se nos devuelve /rtvcm.

Compilación de Python 2.7.6

CentOS viene con Python 2.6. Sin embargo, yo he desarrollado con la versión 2.7.6, y quería utilizar la misma en producción. Básicamente, hay que compilar, y después hacer un enlace simbólico “python” en `/usr/local/bin`, apuntando al binario compilado. He seguido esta guía. Esta otra es parecida.

Samba no responde

Esto sucedió a partir de un parón por el traslado del CPD de un cuarto a otro. Apagamos directamente los hosts desde vSphere. No fuimos apagando VM por VM. Al encender, el Exportador M no podía escribir en las carpetas compartidas de WS. Pasaban dos cosas:

- Ent-11 había olvidado las credenciales de acceso (ver KeePass). Esta vez he seleccionado “recordar credenciales”. CUIDADO porque para contraseñas guardadas no se interpreta igual WS que WS.AMD.LOCAL. Yo lo he dejado todo como WS. El sufijo amd.local ya viene por DHCP. *NOTA: no hay más credenciales aparte de las de ws. Esto es debido a que las demás máquinas con las que se comunican los exportadores tienen el mismo usuario y contraseña que la cuenta sobre la que estos se ejecutan.*
- Samba parecía estar colgado. Requirió un reinicio:

```
[root@ws ~]# smbclient -U ws '\\localhost\a
Enter ws's password:
Connection to localhost failed (Error
NT_STATUS_CONNECTION_REFUSED)
[root@ws ~]#
[root@ws ~]#
[root@ws ~]#
[root@ws ~]#
[root@ws ~]# service smb restart
Shutting down SMB services:
[  OK  ]
Starting SMB services:
[  OK  ]
[root@ws ~]#
[root@ws ~]#
[root@ws ~]# smbclient -U ws '\\localhost\a
Enter ws's password:
Domain=[m] OS=[Unix] Server=[Samba 3.6.9-169.el6_5]
smb: \> ls
.                                   D        0  Mon Oct 13
10:05:50 2014
..                                  D        0  Mon Oct 13
12:05:15 2014
radar                               D        0  Mon Oct 13
10:24:18 2014
meteosat-ir                         D        0  Wed Oct 15
17:16:58 2014

33827 blocks of size 262144. 27286 blocks available
smb: \>
```