Kube-RPI

Antonio Luis Luna Márquez

18 de junio de 2017

Índice general

1. Introducción **5**

1.1. Objetivos 5

2. Crossbuild toolchain **7**

2.1. Introducción 7

2.2. Creación del crossbuild root 7

2.3. Instalación de las dependencias de compilación 8

2.4. Binutils 8

2.4.1. Descarga y compilación del código fuente de binutils 8

2.5. GCC 9

2.5.1. Descarga, compilación e instalación de las GCC 9

2.6. Añadir binarios a la variable 'PATH' 9

3. Kernel **11**

3.1. Introducción 11

3.2. Descarga del kernel 11

3.3. Configuración del kernel 12

3.4. Compilación del kernel 15

4. Creación de la imagen base del sistema operativo **17**

4.1. Introducción 17

4.2. Instalación de las dependencias necesarias 17

4.3. Buildroot 18

4.3.1. Creación y mapeo del fichero de imagen 18

4.3.2. Particionado del fichero de imagen 19

4.3.3. Creación de los sistemas de ficheros 20

4.3.4. Montaje de los sistemas de ficheros 21

4.4. Debootstrap 22

4.4.1. Debootstrap - First Stage 22

4.4.2. Debootstrap - Second Stage 23

4.5. Instalación del kernel 24

4.6. Instalación de los módulos del kernel 24

4.7. Instalación de la imagen del kernel 24

4.8. Instalación del firmware de la Raspberry Pi 3 25

4.8.1. Descarga del firmware 25

4.8.2. Instalación del firmware en el buildroot 25

4.9. Copia de la imagen a la tarjeta SD 26

5. Paquetería del clúster **27**
5.1. Docker 27
5.1.1. Modificación de las dependencias originales del paquete 28
5.2. Etcd 29
5.3. Network-preconfigure 30
5.4. Instalación de los paquetes modificados 30
5.4.1. Docker.io 30
5.4.2. Etcd 32
5.4.3. Network-preconfigure 32
5.5. Kubernetes 32

6. Aprovisionamiento del clúster **33**
6.1. Ansible 33
6.1.1. Kubernetes Master 36
6.1.2. Kubernetes Minion 39
6.1.3. Plantillas de configuración 41

Capítulo 1

Introducción

1.1. Objetivos

- Usar un Sistema Operativo de 64 bits en los dispositivos Raspberry Pi 3
- Configurar un clúster Kubernetes en los dispositivos Raspberry Pi 3

Capítulo 2

Crossbuild toolchain

2.1. Introducción

Para poder compilar el kernel en ARM 64 vamos a utilizar una técnica denominada çross building". A grandes rasgos hacer cross building es compilar el código fuente de un software en una arquitectura distinta a la arquitectura que usará el sistema que va a ejecutar dicho software.

Este procedimiento se suele usar en máquinas con más potencia que los dispositivos en los que se va a usar el software y de este modo ahorrar tiempo a la hora de compilar el código.

2.2. Creación del crossbuild root

El primer paso es configurar nuestro sistema de desarrollo para poder compilar el código que después se ejecutará en los dispositivos Raspberry. Para ello necesitaremos el siguiente software:

- Binutils

  Las GNU Binutils son un conjunto de herramientas destinadas a la manipulación de código objeto. Este software es indispensable para poder compilar el kernel en arquitectura ARM 64

- Gcc

  La GNU Compiler Collection son un conjunto de compiladores de distintos lenguajes y dependen directamente de las binutils. Las utilidades de GCC serán las encargadas de crear los objetos que después enlazarán las binutils, creando el conjunto de elementos que conformarán nuestro kernel.

2.3. Instalación de las dependencias de compilación

El primer paso será instalar las dependencias necesarias para poder compilar el software binutils:

```
root@cross:~# sudo apt-get install build-essential \
              libgmp-dev \
              libmpfr-dev \
              libmpc-dev \
              bc \
              libncurses5 \
              libncurses5-dev \
              git
```

2.4. Binutils

2.4.1. Descarga y compilación del código fuente de binutils

```
debian@cross:~$ curl -L https://ftp.gnu.org/gnu/binutils/binutils-2.28.tar.bz2 \
                -o 'binutils-2.28.tar.bz2'
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 25.3M  100 25.3M    0     0  1566k      0  0:00:16  0:00:16 --:--:-- 3494k

debian@cross:~$ tar xf binutils-2.28.tar.bz2
debian@cross:~$ cd binutils-2.28/
debian@cross:~/binutils-2.28$ ./configure --prefix=/opt/aarch64 \
                              --target=aarch64-linux-gnu
debian@cross:~/binutils-2.28$ make -j 3
debian@cross:~/binutils-2.28$ sudo make install
```

2.5. GCC

El siguiente paso es instalar las GCC para poder compilar nuestro kernel más adelante. Para ello seguiremos los siguientes pasos:

2.5.1. Descarga, compilación e instalación de las GCC

```
debian@cross:~$ curl -L 'https://ftp.gnu.org/gnu/gcc/gcc-6.3.0/gcc-6.3.0.tar.bz2' \
                -o 'gcc-6.3.0.tar.bz2'
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 95.2M  100 95.2M    0     0  2527k      0  0:00:38  0:00:38 --:--:-- 3835k


debian@cross:~$ tar xf gcc-6.3.0.tar.bz2

debian@cross:~/gcc-out$ ../gcc-6.3.0/configure \
                        --prefix=/opt/aarch64 \
                        --target=aarch64-linux-gnu \
                        --with-newlib \
                        --without-headers \
                        --disable-shared \
                        --enable-languages=c

debian@cross:~/gcc-out$ make all-gcc -j3

debian@cross:~/gcc-out$ sudo make install-gcc
```

2.6. Añadir binarios a la variable 'PATH'

Los sistemas operativos tipo Unix usan la variable 'PATH' para indicar al sistema operativo los directorios donde se encuentran los ficheros binarios de los distintos programas que están instalados en el sistema. Para añadir nuestros nuevos binarios recién compilados e instalados ejecutaremos el siguiente comando:

```
root@cross:~# echo "export PATH=$PATH:/opt/aarch64/bin/" >> /etc/profile
```

Este comando inserta el nuevo directorio al fichero '/etc/profile', donde se encuentran las configuraciones globales de los usuarios del sistema.

Capítulo 3

Kernel

3.1. Introducción

El kernel de un sistema operativo es el encargado de administrar los recursos hardware de los distintos dispositivos que se encuentren conectados a la máquina que ejecuta el sistema operativo.

En nuestro caso, usaremos una versión modificada específicamente para funcionar en los dispositivos Raspberry Pi del kernel Linux. Este kernel va a ser compilado en arquitectura ARM de 64 bits, que en la fecha de la realización de este documento se encuentra en estado de pruebas.

3.2. Descarga del kernel

Este proceso se realizará mediante el clonado del repositorio oficial de la fundación Raspberry Pi en Github. La versión del kernel que nos interesa es la '4.9.y', ya que es la más parecida a la versión de kernel que tiene el SO base que usaremos más adelante.

Para clonar el repositorio, usaremos el siguiente comando:

```
debian@cross:~$ git clone \
                --depth=1 \
                -b rpi-4.9.y \
                https://github.com/raspberrypi/linux.git
Cloning into 'linux'...
remote: Counting objects: 60248, done.
remote: Compressing objects: 100% (56615/56615), done.
remote: Total 60248 (delta 5262), reused 16408 (delta 2738), pack-reused 0
Receiving objects: 100% (60248/60248), 158.96 MiB | 7.04 MiB/s, done.
Resolving deltas: 100% (5262/5262), done.
Checking connectivity... done.
Checking out files: 100% (56749/56749), done.
```

3.3. Configuración del kernel

En este apartado configuraremos la compilación del kernel. La fundación Raspberry tiene configurados unos 'perfiles' para las distintas placas que ha comercializado, realizando así la configuración básica de la compilación adaptada al dispositivo donde se va a usar.

Para realizar dicha configuración, debemos ejecutar los siguientes comandos:

```
debian@cross:~$ mkdir kernel-out

debian@cross:~$ cd linux/

debian@cross:~/linux$ make O=../kernel-out/ \
                      ARCH=arm64 \
                      CROSS_COMPILE=aarch64-linux-gnu- \
                      bcmrpi3_defconfig
make[1]: Entering directory '/home/debian/kernel-out'
  HOSTCC  scripts/basic/fixdep
  GEN     ./Makefile
  HOSTCC  scripts/kconfig/conf.o
  SHIPPED scripts/kconfig/zconf.tab.c
  SHIPPED scripts/kconfig/zconf.lex.c
  SHIPPED scripts/kconfig/zconf.hash.c
  HOSTCC  scripts/kconfig/zconf.tab.o
  HOSTLD  scripts/kconfig/conf
#
# configuration written to .config
#
make[1]: Leaving directory '/home/debian/kernel-out'
```

Básicamente estamos creando el directorio kernel-out, que será el destino de los distintos objetos resultantes de la compilación. Una vez creado ese directorio, creamos la configuración de compilación con los parámetros por defecto para las placas "Raspberry Pi 3" indicando la arquitectura de destino **"arm64"** y el conjunto de compiladores **"aarch64-linux-gnu-"** que instalamos en los pasos anteriores.

El siguiente paso es configurar el resto de características de kernel ue necesitaremos para poder realizar el clúster con éxito. Estas características son las relacionadas con los control groups o **"cgroups"**

Los cgroups son un conjunto de características del kernel linux que permiten aislar y controlar el uso de recursos (I/O, memoria, CPU, etc...) de los procesos que se ejecutan en el sistema.

Para asegurarnos de que estos componentes de kernel están incluidos en la compilación buscamos las siguientes flags en el fichero “**.config“** (pueden estar como módulo “=m“ o enlazado estáticamente “=y“)

- CONFIG_CGROUPS
- CONFIG_BLK_CGROUP
- CONFIG_DEBUG_BLK_CGROUP
- CONFIG_CGROUP_WRITEBACK
- CONFIG_CGROUP_SCHED
- CONFIG_CGROUP_PIDS
- CONFIG_CGROUP_FREEZER
- CONFIG_CGROUP_DEVICE
- CONFIG_CGROUP_CPUACCT
- CONFIG_CGROUP_PERF
- CONFIG_CGROUP_DEBUG
- CONFIG_NET_CLS_CGROUP
- CONFIG_SOCK_CGROUP_DATA
- CONFIG_CGROUP_NET_CLASSID
- CONFIG_CPUSETS
- CONFIG_PROC_PID_CPUSET

Si alguna de estas características no está incluida, podremos usar el configurador de componentes incluido en el código fuente para habilitarlas con el siguiente comando:

```
debian@cross:~/linux$ make -j3 \
                     O=../kernel-out/ \
                     ARCH=arm64 \
                     CROSS_COMPILE=aarch64-linux-gnu- \
                     nconfig
```

Esta sentencia nos devolverá un menú tipo “curses“ donde podremos elegir a nuestro albedrío las distintas características que consideremos oportunas:

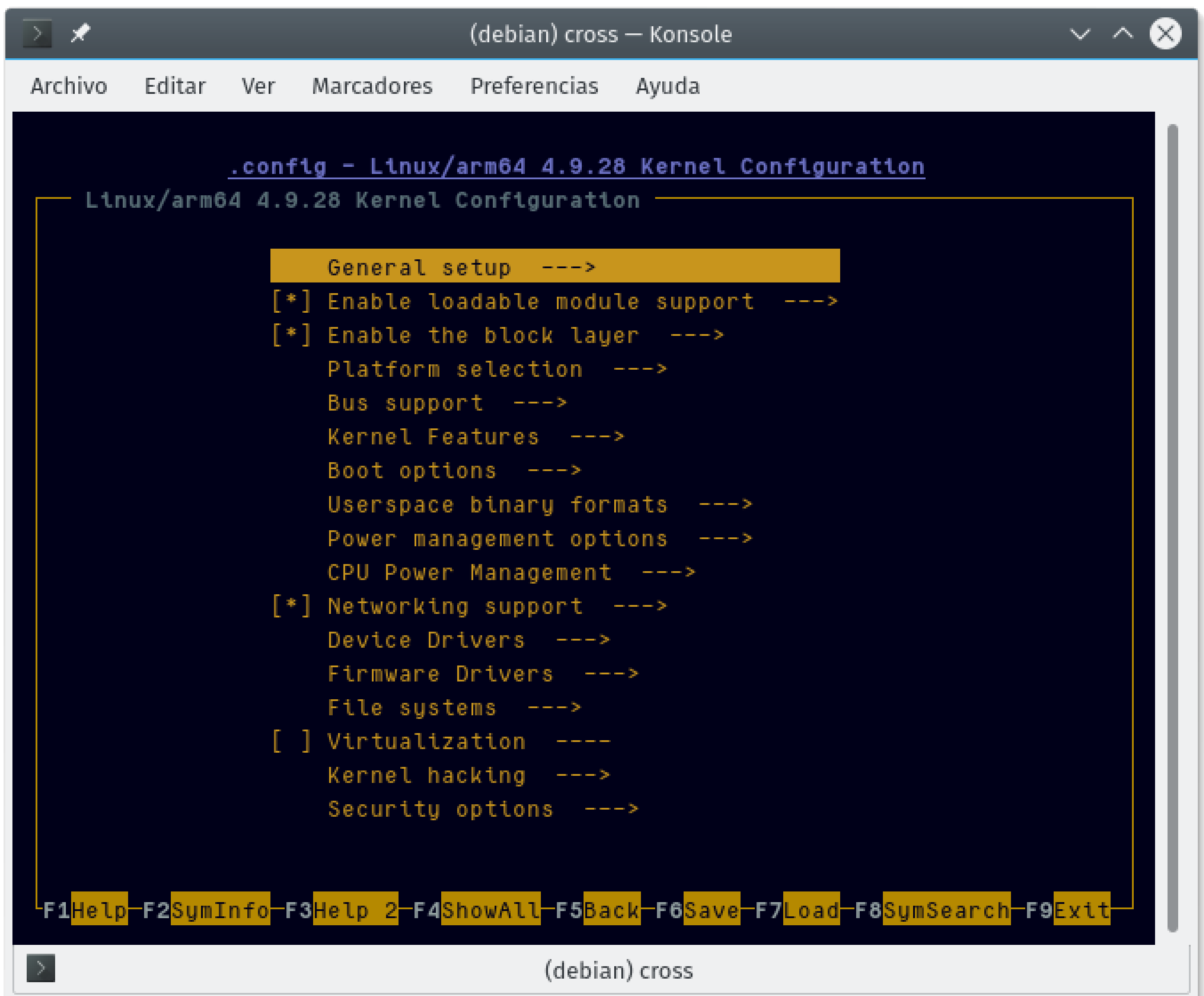

Figura 3.1: Menú de configuración tipo curses

3.4. Compilación del kernel

Una vez configuradas las características de nuestro kernel, es hora de compilarlo. Para ello sólo debemos ejecutar lo siguiente:

```
debian@cross:~/linux$ make O=../kernel-out/ \
                      ARCH=arm64 \
                      CROSS_COMPILE=aarch64-linux-gnu-
```

Observamos como esta vez no hemos introducido ningún target del fichero “Makefile“, lo hacemos así para que use el target “all“ y nos compile el kernel completo con nuestras opciones. Dependiendo de la potencia de la máquina donde se realice la operación, nos llevará mas o menos tiempo. En una máquina virtual con tres núcleos y 2 GB de ram sobre un Intel i7 4702MQ tarda aproximadamente 6 minutos en compilar.

Si la compilación ha resultado satisfactoria, podemos pasar al siguiente capítulo. Más adelante instalaremos el kernel recién compilado.

Capítulo 4

Creación de la imagen base del sistema operativo

4.1. Introducción

En este capítulo crearemos una imagen mínima de Debian Stretch 9 en su versión ARM 64. Dicha imagen no contendrá el kernel, sólo los binarios necesarios para tener un sistema Debian funcional.

Al final del mismo obtendremos una imagen base capaz de arrancar en los dispositivos Raspberry Pi 3. A partir de ese punto, todas las operaciones se realizarán en ellas, siendo innecesario el entorno crossbuild.

4.2. Instalación de las dependencias necesarias

Para llevar a cabo el objetivo de este capítulo, deberemos instalar las siguientes dependencias:

```
root@cross:~# apt-get install kpartx \
               dosfstools \
               debootstrap \
               qemu-user-static
```

4.3. Buildroot

Esta sección describirá el proceso de creación de la imagen que después sera copiada a la tarjeta SD que introduciremos en las Raspberry PI 3

4.3.1. Creación y mapeo del fichero de imagen

El primer paso que debemos realizar, es crear un fichero de dispositivo de bloques. Este fichero lo crearemos con la utilidad básica **"dd" (duplicate disk)**.

Como entrada usaremos el fichero especial "/dev/zero" que como su nombre indica, llenará nuestro fichero de bloques de ceros. El tamaño del fichero de bloques será de 2GB:

```
root@cross:~# dd if=/dev/zero of=./rpi.raw bs=1M count=2048
2048+0 registros leidos
2048+0 registros escritos
2147483648 bytes (2,1 GB) copiados, 4,85795 s, 442 MB/s
```

Ahora tenemos un fichero lleno de ceros, que podría usarse como dispositivo de bloques. Pero para ello debemos montarlo como un dispositivo "loop". Un dispositivo loop no es más que un mapeo que se realiza para poder acceder a un fichero como si de un dispositivo de bloques se tratase, para ello ejecutamos el siguiente comando:

```
root@cross:~# losetup -f rpi.raw
```

Podemos comprobar que el dispositivo se ha montado como dispositivo de bloques correctamente con los siguientes comandos:

```
root@cross:~# losetup -l
NAME       SIZELIMIT OFFSET AUTOCLEAR RO BACK-FILE
/dev/loop0         0      0         0  0 /root/rpi.raw


root@cross:~# lsblk /dev/loop0
NAME  MAJ:MIN RM SIZE RO TYPE MOUNTPOINT
loop0   7:0    0   2G  0 loop
```

Desde este momento podremos tratar el fichero "rpi.raw" como si se tratase de un dispositivo de bloques tradicional.

4.3.2. Particionado del fichero de imagen

Los dispositivos Raspberry necesitan una partición en formato **FAT32** para arrancar. De hecho en dicha partición es donde deberemos tener los ficheros de kernel y algunos otros que veremos más adelante, así que crearemos una partición en dicho formato y otra en formato ext4 para el resto del sistema:

- **Partición W95 FAT32**

```
root@cross:~# fdisk /dev/loop0

Orden (m para obtener ayuda): o
Created a new DOS disklabel with disk identifier 0xb0e40e3c.

Orden (m para obtener ayuda): n
Seleccionar (valor predeterminado p): pú
Nmero de óparticin (1-4, valor predeterminado 1): 1
Primer sector (2048-4194303, valor predeterminado 2048): 2048Ú
ltimo sector, +sectores o +ñtamao{K,M,G,T,P} (2048-4194303, valor predeterminado 4194
Crea una nueva óparticin 1 de tipo 'Linux' y de ñtamao 250 MiB.
Orden (m para obtener ayuda): t
Se ha seleccionado la óparticin 1ó
Cdigo hexadecimal (escriba L para ver todos los ócdigos): c
Se ha cambiado el tipo de la óparticin 'Linux' a 'W95 FAT32 (LBA)'.
```

- **Partición Ext4**

```
Orden (m para obtener ayuda): n
Tipo de óparticin
Seleccionar (valor predeterminado p): pú
Nmero de óparticin (2-4, valor predeterminado 2): 2
Primer sector (514048-4194303, valor predeterminado 514048): 514048Ú
ltimo sector, +sectores o +ñtamao{K,M,G,T,P} (514048-4194303, valor predeterminado 41
Crea una nueva óparticin 2 de tipo 'Linux' y de ñtamao 1,8 GiB.
Orden (m para obtener ayuda): w
```

4.3.3. Creación de los sistemas de ficheros

Aunque nuestro fichero de bloques esté correctamente particionado, aún el SO no tiene la información sobre dicho particionado, esto es normal al ser un fichero de bloques. Para solucionar este pequeño handicap usaremos **kpartx** (aunque existen otras soluciones, como indicar el offset a la hora de montar):

```
root@cross:~# kpartx -a /dev/loop0
```

Si todo es correcto, podremos ver nuestras particiones mapeadas:

```
root@cross:~# lsblk /dev/loop0
NAME      MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
loop0       7:0    0    2G  0 loop
  loop0p1 253:0    0  250M  0 part
  loop0p2 253:1    0  1,8G  0 part
```

Ahora podemos darle formato a esas particiones, para ello usaremos los siguientes comandos:

- **Sistema de ficheros FAT32**

```
root@cross:~# mkfs.msdos -F 32 /dev/mapper/loop0p1
mkfs.fat 3.0.27 (2014-11-12)
unable to get drive geometry, using default 255/63
```

- **Sistema de ficheros Ext4**

```
root@cross:~# mkfs.ext4 /dev/mapper/loop0p2
mke2fs 1.42.12 (29-Aug-2014)
Descartando los bloques del dispositivo: hecho
Se áest creando El sistema de ficheros con 460032 4k bloques y 115200 nodos-i

UUID del sistema de ficheros: c1f6976b-0148-409d-8c38-c637f70c3142
Respaldo del superbloque guardado en los bloques:
        32768, 98304, 163840, 229376, 294912

Reservando las tablas de grupo: hecho
Escribiendo las tablas de nodos-i: hecho
Creando el fichero de transacciones (8192 bloques): hecho
Escribiendo superbloques y la informacion contable del sistema de ficheros: hecho
```

4.3.4. Montaje de los sistemas de ficheros

Para terminar la creación del buildroot, crearemos dos directorios en “/mnt“, el se llamará “boot“, donde irán todos los ficheros necesarios para el arranque inicial del dispositivo, el segundo será “root“, donde irá el resto de la imagen:

```
root@cross:~# mkdir /mnt/{boot,root}
```

Por último, montamos los dos sistemas de ficheros que acabamos de crear en sendos directorios:

- **Boot**

```
root@cross:~# mount /dev/mapper/loop0p1 /mnt/boot/
```

- **Root**

```
root@cross:~# mount /dev/mapper/loop0p2 /mnt/root/
```

Comprobamos que todo sea correcto:

```
root@cross:~# lsblk -f /dev/loop0
NAME      FSTYPE LABEL UUID                                 MOUNTPOINT
loop0
  loop0p1 vfat         2B97-7202                            /mnt/boot
  loop0p2 ext4         c1f6976b-0148-409d-8c38-c637f70c3142 /mnt/root
```

4.4. Debootstrap

En esta sección crearemos el sistema base (sin kernel) del sistema operativo. Dicho sistema base estará basado en Debian 9 Stretch, que se encuentra en fase testing, por lo que tiene, como veremos más adelante, algunos paquetes con dependencias rotas.

Por el momento crearemos una versión con la paquetería básica de Debian con “debootstrap“. Debootstrrap nos permitirá instalar Debian en un diractorio de nuestra máquina crossbuild. Dicho directorio será **“/mnt/root“** donde está montado el sistema de ficheros **ext4** de nuestro fichero de bloques **“rpi.raw“**, pero existe un inconveniente: el sistema anfitrión es de arquitectura **AMD64** mientras que el sistema objetivo es **ARM64**. Por suerte debootstrap cuenta con ello y podremos realizar la instalación de nuestro sistema en dos pasos.

4.4.1. Debootstrap - First Stage

En este paso, debootstrap descargará la paquetería básica de la distribución que le indiquemos en el directorio indicado. Para realizar el proceso ejecutamos:

```
root@cross:~#  debootstrap \
               --arch arm64 \
               --foreign \
               stretch \
               /mnt/root
```

Detengámonos a observar el comando:

- **–arch arm64:** Indicamos la arquitectura que tendrá nuestro sistema objetivo
- **–foreign:** Con este flag, indicamos que la arquitectura objetivo es distinta a la del anfitrión (por lo que no intenta configurar el sistema aún, dando paso a la **second stage**).
- **stretch:** Versión del SO a instalar
- **/mnt/root:** Directorio donde se instalará

4.4.2. Debootstrap - Second Stage

Llegados a este punto, debemos ejecutar el segundo paso de debootstrap. Para ello debemos hacer un chroot al directorio donde hemos instalado el nuevo sistema Debian. Si lo intentamos tal y como está ahora, la shell nos devolverá un error, comentando que ha sido imposible realizar el chroot al destino, lo cual es lógico, ya que las arquitecturas de los binarios son diferentes por lo que necesitaremos un intérprete de los binarios ARM64.

Si volvemos la vista atrás, en las dependencias de este capítulo, podemos encontrar un paquete llamado **“qemu-user-static“**. Dicho paquete contiene el intérprete necesario para poder llevar a cabo nuestra labor, para poder hacer chroot, sólo debemos copiar el fichero **“qemu-aarch64-static“** al directorio **“/usr/bin del buildroot“**

```
root@cross:~# cp /usr/bin/qemu-aarch64-static /mnt/root/usr/bin/
```

Una vez copiado, podremos hacer el chroot:

```
root@cross:~# chroot /mnt/root /bin/bash

I have no name!@cross:/#
```

Ahora sólo debemos ejecutar lo siguiente para que debootstrap complete la instalación de nuestro sistema base:

```
I have no name!@cross:/# /debootstrap/debootstrap --second-stage
```

Para poder acceder a nuestro sistema debemos crear la contraseña para el usuario “root“ y opcionalmente añadir el usuario “debian“:

```
I have no name!@cross:/# passwd root
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully

I have no name!@cross:/# useradd -m -s /bin/bash debian

I have no name!@cross:/# passwd debian
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
```

Ahora podemos dar por terminada la configuración del sistema Debian.

4.5. Instalación del kernel

Aunque la imagen está casi lista, nos queda aún un paso más, que es la instalación del kernel que compilamos en el capítulo 1. Dicha instalación constará de tres partes:

- **Instalación de los módulos del kernel**
- **instalación de la imagen de kernel**
- **Instalación del firmware de la Raspberry Pi 3**

4.6. Instalación de los módulos del kernel

Para instalar los módulos de kernel, volvemos al directorio donde se encuentra el código fuente del mismo y ejecutamos lo siguiente:

```
debian@cross:~/linux$ sudo make -j3 O=../kernel-out/ ARCH=arm64 \
                      INSTALL_MOD_PATH=/mnt/root \
                      CROSS_COMPILE=aarch64-linux-gnu- \
                      modules_install
```

La variable **INSTALL_MOD_PATH** indica el directorio raíz donde se instalarán los módulos.

4.7. Instalación de la imagen del kernel

En este paso instalaremos la imagen con los componentes enlazados estáticamente del kernel (en otras distribuciones **vmlinuz**), los dtbs y las overlays

- **Imagen:**

```
debian@cross:~/linux$ sudo cp \
       ../kernel-out/arch/arm64/boot/Image \
       /mnt/boot/kernel8.img
```

- **Dtb:**

```
debian@cross:~/linux$ sudo cp \
       ../kernel-out/arch/arm64/boot/dts/broadcom/bcm2710-rpi-3-b.dtb \
       /mnt/boot/
```

- **Overlays:**

```
debian@cross:~/linux$ sudo cp -r \
       ../kernel-out/arch/arm64/boot/dts/overlays \
       /mnt/boot/
```

4.8. Instalación del firmware de la Raspberry Pi 3

El SOC de los dispositivos Raspberry Pi contienen en el SOC una memoria **OTP** (One Time Programmable) que busca varios archivos para arrancar el dispositivo. Entre estos ficheros se encuentran:

- **bootcode.bin**
- **start.elf**
- **start_cd.elf**
- **start_db.elf**
- **start_x.elf**
- **fixup_cd.dat**
- **fixup.dat**
- **fixup_db.dat**
- **fixup_x.dat**

4.8.1. Descarga del firmware

Para conseguir los ficheros que necesitamos, clonamos el repositorio github “firmware“ de la fundación:

```
debian@cross:~$ git clone --depth 1  https://github.com/raspberrypi/firmware
Cloning into 'firmware'...
remote: Counting objects: 4469, done.
remote: Compressing objects: 100% (2658/2658), done.
remote: Total 4469 (delta 1628), reused 2310 (delta 1484), pack-reused 0
Receiving objects: 100% (4469/4469), 82.58 MiB | 8.79 MiB/s, done.
Resolving deltas: 100% (1628/1628), done.
Checking connectivity... done.
```

4.8.2. Instalación del firmware en el buildroot

Una vez conseguido el firmware, sólo debemos ejecutar los siguientes comandos:

```
debian@cross:~$ cd firmware/boot/
debian@cross:~/firmware/boot$ sudo cp bootcode.bin start* fixup* /mnt/boot/
```

Realizado este paso podemos dar por concluida la realización de la imagen base. Ahora tenemos un sistema operativo funcional, capaz de arrancar en los dispositivos Raspberry.

4.9. Copia de la imagen a la tarjeta SD

Para poder arrancar nuestro SO en la Raspberry debemos copiar la imagen a una tarjeta SD. Para ello volvemos a usar el software **dd**:

```
antonio@Lenovo-G510 ~
sudo dd if=rpi.raw of=/dev/mmcblk0 bs=1M
2048+0 registros íledos
2048+0 registros escritos
2147483648 bytes (2,1 GB, 2,0 GiB) copied, 664,555 s, 3,2 MB/s
```

Es muy importante sincronizar los buffers del sistema de ficheros, para ello usamos el comando **“sync“**

```
antonio@Lenovo-G510 ~
sync
```

Una vez terminado el proceso, podemos retirar la SD e introducirla en la raspberry Pi 3.

Capítulo 5

Paquetería del clúster

En este capítulo configuraremos la paquetería necesaria en para poder realizar nuestro clúster de Kubernetes, todos los pasos de este capítulos se realizarán en el dispositivo, ya que al tener un SO funcional, no dependemos del crossbuild toolchain en absoluto.

5.1. Docker

Esta sección estará dedicada a la instalación del container runtime Docker. Este gestor de contenedores es el que ofrece mayor compatibilidad con Kubernetes, pero Debian Stretch tiene a día de hoy dependencias rotas, en concreto los paquetes esenciales **runc** y **containerd**, por lo que vamos a utilizar un paquete originario de **“Launchpad“** para Ubuntu 16.04 Xenial Xerus, que como comprobaremos más adelante, una vez modificado es completamente compatible con Debian Stretch.

Download docker.io

Architecture	Version
amd64	1.13.1~ds1-2
arm64	1.11.2~ds1-5
armel	1.11.2~ds1-6
armhf	1.11.2~ds1-6
i386	1.11.2~ds1-6
ppc64el	1.11.2~ds1-6

Figura 5.1: Estado del paquete “docker.io“ para Debian 9

5.1.1. Modificación de las dependencias originales del paquete

Para poder hacer el paquete compatible con nuestro sistema Debian, deberemos modificar algunas dependencias que son originarias de Ubuntu. Estas dependencias pueden editarse en el fichero **“control“** del paquete. También habrá que editar el fichero **“md5sums“** con el nuevo hash de nuestro fichero o nos dará un error en el proceso de empaquetado.

```
[antonio@Lenovo-G510:/docker.io-aluna]$: diff docker.io-aluna/DEBIAN/control control-or
2c2
< Version: 1.12.6-0-debian-aluna
---
> Version: 1.12.6-0ubuntu4
6,7c6,7
< Depends: adduser, containerd (>= 0.2.3~), iptables, runc (>= 1.0.0~),
init-system-helpers (>= 1.18~), libapparmor1 (>= 2.6~devel), libc6 (>= 2.17),
libdevmapper1.02.1 (>= 2:1.02.97), libseccomp2 (>= 2.1.0)
< Recommends: ca-certificates, cgroupfs-mount | cgroup-lite, git, xz-utils, apparmor
---
> Depends: adduser, containerd (>= 0.2.5~), iptables, runc (>= 1.0.0~rc2-0ubuntu1~),
init-system-helpers (>= 1.18~), libapparmor1 (>= 2.6~devel), libc6 (>= 2.17),
libdevmapper1.02.1 (>= 2:1.02.97), libseccomp2 (>= 2.1.0)
> Recommends: ca-certificates, cgroupfs-mount | cgroup-lite, git, ubuntu-fan,
xz-utils, apparmor
10a11
> Built-Using: glibc (= 2.24-7ubuntu2), golang-1.7 (= 1.7.4-1ubuntu1)
```

Podemos observar como cambiamos las dependencias de **runc** y **containerd**. También eliminamos el resto de paquetería recomendada original y la sustituimos por la de Debian

5.2. Etcd

El paquete etcd en arquitectura ARM64 tiene un pequeño fallo en el servicio systemd. Etcd actualmente está en fase testing en esta arquitectura, por lo que hay que añadir una pequeña variable de entorno en dicho servicio para que se ejecute el demonio correctamente.

Para solucionar este pequeño inconveniente, solo debemos añadir una línea **“environment“** al servicio systemd original incluido en el paquete de Debian Stretch

```
[antonio@Lenovo-G510:/etcd]$: diff etcd_3.1.4-aluna/lib/systemd/system/etcd.service etc
10d9
< Environment=ETCD_UNSUPPORTED_ARCH=arm64
```

Fichero de servicio systemd:

```
[Unit]
Description=etcd - highly-available key value store
Documentation=https://github.com/coreos/etcd
Documentation=man:etcd
After=network.target
Wants=network-online.target

[Service]
Environment=DAEMON_ARGS=
Environment=ETCD_UNSUPPORTED_ARCH=arm64
Environment=ETCD_NAME=%H
Environment=ETCD_DATA_DIR=/var/lib/etcd/default
EnvironmentFile=-/etc/default/%p
Type=notify
User=etcd
PermissionsStartOnly=true
ExecStart=/usr/bin/etcd $DAEMON_ARGS
Restart=on-abnormal
LimitNOFILE=65536

[Install]
WantedBy=multi-user.target
Alias=etcd2.service
```

Con esta pequeña modificación, tenemos los ficheros del paquete etcd listo para ser empaquetado.

5.3. Network-preconfigure

El kernel que hemos compilado, da un nombre distinto a la interfaz cableada dependiendo de la máquina donde esté corriendo. Para solucionar este problema crearemos un paquete deb que instale un pequeño script que se encargará de crear el fichero "/etc/network/interfaces" con la configuración correcta.

El código de dicho script es el siguiente:

```
1  #! /bin/bash
2
3  iface=$(ip l | grep 'enx' | awk '{print $2}')
4  iface=${iface//:}
5  iface_exists=$(grep "$iface" /etc/network/interfaces | wc -l)
6
7
8  if [[ ! -e "/etc/network/interfaces" ]] || [[ $iface_exists = 0 ]]; then
9
10 cat <<EOF > /etc/network/interfaces
11 # interfaces(5) file used by ifup(8) and ifdown(8)
12 # Include files from /etc/network/interfaces.d:
13 source-directory /etc/network/interfaces.d
14
15 auto $iface
16 iface $iface inet dhcp
17 EOF
18 fi
19
20 systemctl restart networking.service
```

Básicamente el script comprueba que el nombre de la interfaz se encuentra en el fichero, si no aparece o el fichero interfaces no existe, escribe la configuración.

5.4. Instalación de los paquetes modificados

Muchas de las dependencias del software que vamos a instalar se encuentran en el repositorio **"sid"**. Para añadir dicho repositorio ejecutaremos los siguientes comandos:

```
root@debian:~# echo "deb http://httpredir.debian.org/debian unstable main" >> /etc/apt/
root@debian:~# apt update
Hit:1 http://cdn-fastly.deb.debian.org/debian stretch InRelease
Get:2 http://cdn-fastly.deb.debian.org/debian stretch/main Translation-en [5397 kB]
Fetched 5397 kB in 10s (539 kB/s)
Reading package lists... Done
Building dependency tree... Done
All packages are up to date.
```

5.4.1. Docker.io

```
root@debian:~# dpkg -i /root/docker.io-aluna.deb
Selecting previously unselected package docker.io.
(Reading database ... 9660 files and directories currently installed.)
Preparing to unpack /root/docker.io-aluna.deb ...
Unpacking docker.io (1.12.6-0-debian-aluna) ...
dpkg: dependency problems prevent configuration of docker.io:
 docker.io depends on containerd (>= 0.2.3~); however:
```

```
  Package containerd is not installed.
 docker.io depends on runc (>= 1.0.0~); however:
  Package runc is not installed.

dpkg: error processing package docker.io (--install):
 dependency problems - leaving unconfigured
Processing triggers for systemd (232-23) ...
Errors were encountered while processing:
 docker.io
```

Podemos observar como nos devuelve un error. Esto es debido a que las dependencias necesarias no están instaladas. Para solucionarlo ejecutamos lo siguiente:

```
root@debian:~# apt -f install
Reading package lists... Done
Building dependency tree
Reading state information... Done
Correcting dependencies... Done
The following additional packages will be installed:
  containerd runc
The following NEW packages will be installed:
  containerd runc
0 upgraded, 2 newly installed, 0 to remove and 22 not upgraded.
1 not fully installed or removed.
Need to get 4883 kB of archives.
After this operation, 27.4 MB of additional disk space will be used.
Do you want to continue? [Y/n] y
```

5.4.2. Etcd

Este proceso será identico con el paquete etcd

```
root@debian:~# dpkg -i /root/etcd_3.1.4-aluna.deb
Selecting previously unselected package etcd.
(Reading database ... 9515 files and directories currently installed.)
Preparing to unpack /root/etcd_3.1.4-aluna.deb ...
Unpacking etcd (3.1.4+dfsg-1+b1) ...
dpkg: dependency problems prevent configuration of etcd:
 etcd depends on pipexec; however:
  Package pipexec is not installed.

dpkg: error processing package etcd (--install):
 dependency problems - leaving unconfigured
Processing triggers for systemd (232-23) ...
Errors were encountered while processing:
 etcd
```

5.4.3. Network-preconfigure

```
root@debian:~#
```

5.5. Kubernetes

```
root@debian:~# apt install -y console-setup openssh-server \
               kubernetes-client kubernetes-master kubernetes-node \
               flannel
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  console-setup-linux dbus kbd keyboard-configuration libdbus-1-3 libedit2
  libexpat1 libpam-systemd libwrap0 libx11-6 libx11-data libxau6 libxcb1
  libxdmcp6 libxext6 libxmuu1 ncurses-term openssh-client openssh-sftp-server
  tcpd ucf xauth xkb-data
Suggested packages:
  locales default-dbus-session-bus | dbus-session-bus keychain libpam-ssh monkeysphere
The following NEW packages will be installed:
  console-setup console-setup-linux dbus flannel kbd keyboard-configuration kubernetes-
  libxext6 libxmuu1 ncurses-term openssh-client openssh-server openssh-sftp-server tcpd
0 upgraded, 29 newly installed, 0 to remove and 22 not upgraded.
Need to get 69.3 MB of archives.
```

Capítulo 6

Aprovisionamiento del clúster

Este es el capitulo donde describiremos el aprovisionamiento de los distintos nodos que compondrán nuestro clúster. Como la imagen base contiene todo el software necesario (tanto para ser master como para realizar labores de nodo) el despliegue de un clúster completo debería tardar apenas unos minutos.

6.1. Ansible

Para el aprovisionamiento del clúster usaremos Ansible. Ansible es un gestor de configuraciones (CMS) escrito en Python y Microsoft Powershell (para interactuar con clientes Windows). Una de las características por la que nos vamos a decantar por este software es que es “agent free“, lo que sifnifica que los equipos cliente, sólo deben tener instalado Python 2.7 o MS Powershell en el caso de los sistemas Windows para funcionar.

Debido a los módulos que usaremos (como el de systemd), la versión necesaria debe ser superior a la 2.0.

```
[antonio@Lenovo-G510:/~] $: ansible --version
ansible 2.2.1.0
```

Vista general de playbook Ansible

Para comprender mejor lo que vamos a hacer, echaremos un vistazo a la estructura de nuestro “playbook“.

Un playbook de Ansible en resumidas cuentas se trata de uno, o un conjunto de ficheros en formato Yaml que contienen instrucciones sobre las operaciones que se van a realizar sobre los nodos cliente.

En nuestro caso se tratará de los siguientes **ficheros** principales:

- **ansible.cfg:** Este fichero contendrá la configuración básica del playbook, como por ejemplo la clave ssh a utilizar, el usuario por defecto sobre el que actuaremos en el equipo remoto o el fichero de inventario a usar:

```
1 [defaults]
2 hostfile = ./hosts
3 remote_user = root
4 private_key_file = ~/.ssh/id_rsa
```

- **hosts:** Este es el fichero de inventario. Contendrá la lista de servidores sobre los que se realizarán la tareas. Se clasificarán por grupos (en nuestro caso **"Kubernetes_Master" y "Kubernetes_Minion"**).

  Además podemos incluir información extra sobre el nodo, por ejemplo usuario remoto o contraseña:

```
1 [Kubernetes_Master]
2 kbpi_1          ansible_ssh_host=10.0.0.187        ansible_ssh_user=root
3
4 [Kubernetes_Minion]
5 kbpi_2          ansible_ssh_host=10.0.0.124        ansible_ssh_user=root
6 kbpi_3          ansible_ssh_host=10.0.0.201        ansible_ssh_user=root
7 kbpi_4          ansible_ssh_host=10.0.0.240        ansible_ssh_user=root
```

  En este caso, tenemos configurados 1 nodo que actuará de máster y 3 nodos sobre los que desplegaremos los distintos pods.

- **main.yml:** Este fichero contendrá un inventario de todos los ficheros yaml de lso distintos playbooks que ejecutemos. De este modo podemos controlar mejor qué se ejecuta y qué no.

```
1 - include: playbooks/common_master/main.yml
2 - include: playbooks/common_minion/main.yml
```

Por otro lado, crearemos el directorio **“playbooks“** donde se encontraran los ficheros yaml con las instrucciones para los distintos tipos de nodo que se crearán los siguientes **directorios**

- **common_master:**

```
1 [antonio@Lenovo-G510:/ansible] $: ls playbooks/common_master/
2 main.yml  templates  vars_master.yml
```

- **common_minion:**

```
1 [antonio@Lenovo-G510:/ansible] $: ls playbooks/common_minion/
2 main.yml  templates  vars_minion.yml
```

Estos directorios contienen las plantillas con las configuraciones y los ficheros yaml con las distintas operaciones que realizaremos a los grupos de nodos.

6.1.1. Kubernetes Master

Como hemos podido observar en la sección anterior, tenemos tres elementos en cada directorio. En este apartado veremos con más detalle los ficheros que intervienen en la configuración del nodo que actuará como master en el cñúster de kubernetes.

main.yml

Esta plantilla toma las variables del ficher **“vars_master.yml“** para completar las distintas configuraciones de las plantillas. A su vez copia estas plantillas en los directorios adecuados, con los permisos y propietarios adecuados.

Por último activa y reinicia los servicios dependiendo del rol (master) y se asegura que los roles de los nodos “Minion“ estan parados y desactivados. Estos servicios son:

- **etcd**
- **flannel**
- **Kubernetes Api Server**
- **Kubernetes Scheduler**
- **Kubernetes Controller Manager**

```
1  ---
2
3  - hosts: Kubernetes_Master
4    vars_files:
5      - vars_master.yml
6    tasks:
7
8    - name: Configurando Kubernetes Api Server
9      template: >
10       src=templates/kube-apiserver
11       dest=/etc/default/kube-apiserver
12       owner=root
13       group=root
14       mode=0644
15
16   - name: Configurando Etcd
17     template: >
18       src=templates/etcd
19       dest=/etc/default/etcd
20       owner=root
21       group=root
22       mode=0644
23
24   - name: Configurando NTP
25     template: >
26       src=templates/timesyncd.conf
27       dest=/etc/systemd/timesyncd.conf
28       owner=root
29       group=root
30       mode=0644
31
32   - name: Configurando CoreOS Flannel
```

```
33        template: >
34          src=templates/flannel
35          dest=/etc/default/flannel
36          owner=root
37          group=root
38          mode=0644
39
40    - name: Instalando óconfiguracin de Docker
41        template: >
42          src=templates/docker-network
43          dest=/etc/default/docker-network
44          owner=root
45          group=root
46          mode=0644
47
48    - name: Instalando servicio Flannel
49        template: >
50          src=templates/flanneld.service
51          dest=/lib/systemd/system/flanneld.service
52          owner=root
53          group=root
54          mode=0644
55
56    - name: Creando directorio de script
57        file: path=/usr/libexec/flannel state=directory
58
59    - name: Instalando scripts de Flannel
60        template: >
61          src=templates/mk-docker-opts.sh
62          dest=/usr/libexec/flannel/mk-docker-opts.sh
63          owner=root
64          group=root
65          mode=0755
66
67
68    - name: Instalando servicio Docker
69        template: >
70          src=templates/docker.service
71          dest=/lib/systemd/system/docker.service
72          owner=root
73          group=root
74          mode=0644
75
76    - name: Creando óconfiguracin global de Kubernetes
77        template: >
78          src=templates/kubernetes-config
79          dest=/etc/kubernetes/config
80          owner=root
81          group=root
82          mode=0644
83
84    - name: Reiniciando Timesyncd
85        service: name=systemd-timesyncd state=restarted enabled=yes
86
87    - name: Desactivando Docker
88        service: name=docker state=stopped enabled=no
89
90    - name: Desactivando Kubernetes Proxy
91        service: name=kube-proxy state=stopped enabled=no
92
```

```
93      - name: Desactivando Kubernetes Kubelet
94        service: name=kubelet state=stopped enabled=no
95
96      - name: Activando etcd
97        service: name=etcd state=restarted enabled=yes
98
99      - name: Creando directorio etcd de Flannel
100       shell: etcdctl mkdir /atomic.io/network/
101
102     - name: Creando red Flannel para el cluster
103       shell: 'etcdctl mk /atomic.io/network/config "{ \"Network\": \"172.30.0.0/16\", \"S
104
105     - name: Activando flannel
106       systemd: name=flanneld state=started enabled=yes
107
108     - name: Activando Kubernetes Api Server
109       service: name=kube-apiserver state=restarted enabled=yes
110
111     - name: Activando Kubernetes Scheduler
112       service: name=kube-scheduler state=restarted enabled=yes
113
114     - name: Activando Kubernetes Controller Manager
115       service: name=kube-controller-manager state=restarted enabled=yes
```

vars_master.yml

Este fichero contiene las distintas variables que se usarán en las plantillas.

```
1 master_addr: '"--master=http://10.0.0.187:8080"'
2 infra_ntpserver: 0.debian.pool.ntp.org
3 etcd_endpoint: '"http://10.0.0.187:2379"'
4 api_server: '"--api-servers=http://10.0.0.187:8080"'
```

6.1.2. Kubernetes Minion

main.yml

Los servicios que se activan en los nodos “Minion“ son:

- **Docker**
- **Kubernetes Proxy**
- **flannel**
- **Kubelet**

```
1  ---
2
3  - hosts: Kubernetes_Minion
4    vars_files:
5      - vars_minion.yml
6  #  vars:
7  #    etcd_endpoint: '"http://10.0.0.187:2379"'
8  #    api_server: '"--api-servers=http://10.0.0.187:8080"'
9  #    infra_ntpserver: 0.debian.pool.ntp.org
10   tasks:
11
12   - name: Configurando Kubernetes Kubelet
13     template: >
14       src=templates/kubelet
15       dest=/etc/default/kubelet
16       owner=root
17       group=root
18       mode=0644
19
20   - name: Configurando CoreOS Flannel
21     template: >
22       src=templates/flannel
23       dest=/etc/default/flannel
24       owner=root
25       group=root
26       mode=0644
27
28   - name: Instalando servicio Flannel
29     template: >
30       src=templates/flanneld.service
31       dest=/lib/systemd/system/flanneld.service
32       owner=root
33       group=root
34       mode=0644
35
36   - name: Creando directorio de script
37     file: path=/usr/libexec/flannel state=directory
38
39   - name: Instalando scripts de Flannel
40     template: >
41       src=templates/mk-docker-opts.sh
42       dest=/usr/libexec/flannel/mk-docker-opts.sh
43       owner=root
44       group=root
45       mode=0755
```

```
46
47    - name: Instalando óconfiguracin de Docker
48      template: >
49        src=templates/docker-network
50        dest=/etc/default/docker-network
51        owner=root
52        group=root
53        mode=0644
54
55    - name: Instalando servicio Docker
56      template: >
57        src=templates/docker.service
58        dest=/lib/systemd/system/docker.service
59        owner=root
60        group=root
61        mode=0644
62
63    - name: Configurando NTP
64      template: >
65        src=templates/timesyncd.conf
66        dest=/etc/systemd/timesyncd.conf
67        owner=root
68        group=root
69        mode=0644
70    - name: Creando óconfiguracin global de Kubernetes
71      template: >
72        src=templates/kubernetes-config
73        dest=/etc/kubernetes/config
74        owner=root
75        group=root
76        mode=0644
77
78    - name: Reiniciando Timesyncd
79      systemd: name=systemd-timesyncd state=restarted enabled=yes
80
81    - name: Desactivando etcd
82      systemd: name=etcd state=stopped enabled=no
83
84    - name: Desactivando Kubernetes API Server
85      systemd: name=kube-apiserver state=stopped enabled=no
86
87    - name: Desactivando Kubernetes Scheduler
88      systemd: name=kube-scheduler state=stopped enabled=no
89
90    - name: Descativando Kubernetes Controller Manager
91      systemd: name=kube-controller-manager state=stopped enabled=no
92
93    - name: Activando Docker
94      systemd: name=docker state=restarted enabled=yes
95
96    - name: Activando Kubernetes Proxy
97      systemd: name=kube-proxy state=restarted enabled=yes
98
99    - name: Activando flannel
100     systemd: name=flanneld state=started enabled=yes
101
102   - name: Activando Kubelet
103     systemd: name=kubelet state=restarted enabled=yes
```

vars_minion.yml

```
master_addr: '"--master=http://10.0.0.187:8080"'
infra_ntpserver: 0.debian.pool.ntp.org
etcd_endpoint: '"http://10.0.0.187:2379"'
api_server: '"--api-servers=http://10.0.0.187:8080"'
```

6.1.3. Plantillas de configuración

Estas plantillas se encuentran en el directorio “templates“ de cada tipo de nodo:

- **timesyncd.conf**

  Kubernetes es un software muy exigente con la fecha de los sistemas, por lo que configuraremos el servidor NTP que usaremos.

```
#  This file is part of systemd.
#
#  systemd is free software; you can redistribute it and/or modify it
#  under the terms of the GNU Lesser General Public License as published by
#  the Free Software Foundation; either version 2.1 of the License, or
#  (at your option) any later version.
#
# Entries in this file show the compile time defaults.
# You can change settings by editing this file.
# Defaults can be restored by simply deleting this file.
#
# See timesyncd.conf(5) for details.

[Time]
NTP={{ infra_ntpserver }}
FallbackNTP=0.debian.pool.ntp.org 1.debian.pool.ntp.org 2.debian.pool.ntp.org 3.debia
```

- **kubernetes-config**

  En este fichero se encuentra la configuración global de Kubernetes:

```
###
# Kubernetes: common config for the following services:
##
#   kube-apiserver.service
#   kube-controller-manager.service
#   kube-scheduler.service
#   kubelet.service
#   kube-proxy.service
##

# logging to stderr means we get it in the systemd journal
KUBE_LOGTOSTDERR="--logtostderr=true"

# journal message level, 0 is debug
KUBE_LOG_LEVEL="--v=0"

# Should this cluster be allowed to run privileged docker containers
KUBE_ALLOW_PRIV="--allow-privileged=false"

# How the controller-manager, scheduler, and proxy find the apiserver
KUBE_MASTER={{ master_addr }}
```

- **kube-apiserver**

  Fichero con la configuración de la API. Lo configuramos apra que escuche en todas las interfaces

```
###
# kubernetes system config
#
# The following values are used to configure the kube-apiserver
#

# The address on the local server to listen to.
KUBE_API_ADDRESS="--insecure-bind-address=0.0.0.0"

# The port on the local server to listen on.
# KUBE_API_PORT="--port=8080"

# Port minions listen on
# KUBELET_PORT="--kubelet-port=10250"

# Comma separated list of nodes in the etcd cluster
KUBE_ETCD_SERVERS="--etcd-servers=http://127.0.0.1:4001,http://127.0.0.1:2379"

# Address range to use for services
KUBE_SERVICE_ADDRESSES="--service-cluster-ip-range=10.254.0.0/16"

# default admission control policies
#KUBE_ADMISSION_CONTROL="--admission-control=NamespaceLifecycle,NamespaceExists,Limit
KUBE_ADMISSION_CONTROL=""

# Other options:
# --cloud-provider={aws|gce|mesos|openshift|ovirt|rackspace|vagrant}
# --cluster-name="clustername"
#DAEMON_ARGS=""
```

- **etcd**

  Configuramos etcd para que sirva en todas las interfaces.

```
#### Daemon parameters:
# DAEMON_ARGS=""

ETCD_NAME=default
ETCD_DATA_DIR="/var/lib/etcd/default.etcd"
ETCD_LISTEN_CLIENT_URLS="http://0.0.0.0:2379"

#[cluster]
ETCD_ADVERTISE_CLIENT_URLS="http://0.0.0.0:2379"
```

- **etcd.service**

  Debemos añadir la variable de entorno **ETCD_UNSUPPORTED_ARCH=arm64** en el servicio systemd para que etcd se ejecute en arquitectura ARM64. Actualmente Etcd no está soportado en esta arquitectura, por loq ue si no le añadimos esta variable de entorno, nos dará error a la hora de ejecutarse

```
1  [Unit]
2  Description=etcd - highly-available key value store
3  Documentation=https://github.com/coreos/etcd
4  Documentation=man:etcd
5  After=network.target
6  Wants=network-online.target
7  
8  [Service]
9  Environment=DAEMON_ARGS=
10 Environment=ETCD_UNSUPPORTED_ARCH=arm64
11 Environment=ETCD_NAME=%H
12 Environment=ETCD_DATA_DIR=/var/lib/etcd/default
13 EnvironmentFile=-/etc/default/%p
14 Type=notify
15 User=etcd
16 PermissionsStartOnly=true
17 #ExecStart=/bin/sh -c "GOMAXPROCS=$(nproc) /usr/bin/etcd $DAEMON_ARGS"
18 ExecStart=/usr/bin/etcd $DAEMON_ARGS
19 Restart=on-abnormal
20 #RestartSec=10s
21 LimitNOFILE=65536
22 
23 [Install]
24 WantedBy=multi-user.target
25 Alias=etcd2.service
```

- **flanneld.service**

  Servicio Systemd para el software flannel. El paquete Debian de flannel no incluye ningún tipo de configuración, por lo que usaremos los ficheros de configuración de otras distribuciones como base.

```
1  [Unit]
2  Description=Flanneld overlay address etcd agent
3  After=network.target
4  After=network-online.target
5  Wants=network-online.target
6  After=etcd.service
7  Before=docker.service
8  
9  [Service]
10 Type=notify
11 EnvironmentFile=/etc/default/flannel
12 EnvironmentFile=-/etc/default/docker-network
13 ExecStart=/usr/bin/flannel -etcd-endpoints=${FLANNEL_ETCD_ENDPOINTS} -etcd-prefix=${F
14 ExecStartPost=/usr/libexec/flannel/mk-docker-opts.sh -k DOCKER_NETWORK_OPTIONS -d /ru
15 Restart=on-failure
16 
17 [Install]
18 WantedBy=multi-user.target
19 RequiredBy=docker.service
```

- **flannel**

  En esta plantilla debemos configurar el servidor **etcd** del clúster

```
# Flanneld configuration options

# etcd url location.  Point this to the server where etcd runs
#FLANNEL_ETCD_ENDPOINTS="http://127.0.0.1:2379"
FLANNEL_ETCD_ENDPOINTS={{ etcd_endpoint }}

# etcd config key.  This is the configuration key that flannel queries
# For address range assignment
FLANNEL_ETCD_PREFIX="/atomic.io/network"

# Any additional options that you want to pass
#FLANNEL_OPTIONS=""
```

- **mk-docker-opts.sh**

  Este script genera la configuración necesaria **para la configuración de red en Docker**. Una vez creada la configuración, Docker usará la misma para crear su red.

```
#!/bin/bash

usage() {
        echo "$0 [-f FLANNEL-ENV-FILE] [-d DOCKER-ENV-FILE] [-i] [-c] [-m] [-k COMBIN

Generate Docker daemon options based on flannel env file
OPTIONS:
        -f      Path to flannel env file. Defaults to /run/flannel/subnet.env
        -d      Path to Docker env file to write to. Defaults to /run/docker_opts.env
        -i      Output each Docker option as individual var. e.g. DOCKER_OPT_MTU=1500
        -c      Output combined Docker options into DOCKER_OPTS var
        -k      Set the combined options key to this value (default DOCKER_OPTS=)
        -m      Do not output --ip-masq (useful for older Docker version)
" >/dev/stderr

        exit 1
}

flannel_env="/run/flannel/subnet.env"
docker_env="/run/docker_opts.env"
combined_opts_key="DOCKER_OPTS"
indiv_opts=false
combined_opts=false
ipmasq=true

while getopts "f:d:icmk:" opt; do
        case $opt in
                f)
                        flannel_env=$OPTARG
                        ;;
                d)
                        docker_env=$OPTARG
                        ;;
                i)
                        indiv_opts=true
                        ;;
                c)
                        combined_opts=true
                        ;;
                m)
                        ipmasq=false
```

```
42                          ;;
43                  k)
44                          combined_opts_key=$OPTARG
45                          ;;
46                  \?)
47                          usage
48                          ;;
49          esac
50  done
51
52  if [ $indiv_opts = false ] && [ $combined_opts = false ]; then
53          indiv_opts=true
54          combined_opts=true
55  fi
56
57  if [ -f "$flannel_env" ]; then
58          source $flannel_env
59  fi
60
61  if [ -n "$FLANNEL_SUBNET" ]; then
62          DOCKER_OPT_BIP="--bip=$FLANNEL_SUBNET"
63  fi
64
65  if [ -n "$FLANNEL_MTU" ]; then
66          DOCKER_OPT_MTU="--mtu=$FLANNEL_MTU"
67  fi
68
69  if [ -n "$FLANNEL_IPMASQ" ] && [ $ipmasq = true ] ; then
70          if [ "$FLANNEL_IPMASQ" = true ] ; then
71                  DOCKER_OPT_IPMASQ="--ip-masq=false"
72          elif [ "$FLANNEL_IPMASQ" = false ] ; then
73                  DOCKER_OPT_IPMASQ="--ip-masq=true"
74          else
75                  echo "Invalid value of FLANNEL_IPMASQ: $FLANNEL_IPMASQ" > /dev/stderr
76                  exit 1
77          fi
78  fi
79
80  eval docker_opts="\$${combined_opts_key}"
81  docker_opts+=" "
82
83  echo -n "" >$docker_env
84  for opt in $(compgen -v DOCKER_OPT_); do
85          eval val=\$$opt
86
87          if [ "$indiv_opts" = true ]; then
88                  echo "$opt=\"$val\"" >>$docker_env
89          fi
90
91          docker_opts+="$val "
92  done
93
94  if [ "$combined_opts" = true ]; then
95          echo "${combined_opts_key}=\"${docker_opts}\"" >>$docker_env
96  fi
```

- **kubelet**

  Kubelet es el agente de la container runtime, este template lo configura:

```
###
# kubernetes kubelet (minion) config

# The address for the info server to serve on (set to 0.0.0.0 or "" for all interface
KUBELET_ADDRESS="--address=0.0.0.0"

# The port for the info server to serve on
# KUBELET_PORT="--port=10250"

# You may leave this blank to use the actual hostname
#KUBELET_HOSTNAME="--hostname-override=10.0.0.203"
KUBELET_HOSTNAME="--hostname-override={{ ansible_default_ipv4.address }}"

# location of the api-server
#KUBELET_API_SERVER="--api-servers=http://10.0.0.201:8080"
KUBELET_API_SERVER={{ api_server }}

# Docker endpoint to connect to
# Default: unix:///var/run/docker.sock
#DOCKER_ENDPOINT="--docker-endpoint=unix:///var/run/docker.sock"

# Port to listen on
#CADVISOR_PORT="--cadvisor-port=4194"

# Other options:
# --container_runtime=rkt
# --configure-cbr0={true|false}
#DAEMON_ARGS=""
```

- **docker.service**

  Debemos modificar el servicio de Docker, incluyendo el fichero de variables de entorno **“/run/flannel/docker“** que es el fichero que genera el script **“mk-docker-opts.sh“**. Así mismo en la línea **“ExecStart“** debemos añadir la variable **$DOCKER_NETWORK_OPTIONS** para que esta configuración se haga efectiva.

```
[Unit]
Description=Docker Application Container Engine
Documentation=https://docs.docker.com
After=network.target docker.socket
Requires=docker.socket

[Service]
Type=notify
# the default is not to use systemd for cgroups because the delegate issues still
# exists and systemd currently does not support the cgroup feature set required
# for containers run by docker
EnvironmentFile=-/etc/default/docker
EnvironmentFile=-/run/flannel/docker
ExecStart=/usr/bin/dockerd -H fd:// $DOCKER_OPTS $DOCKER_NETWORK_OPTIONS
ExecReload=/bin/kill -s HUP $MAINPID
# Having non-zero Limit*s causes performance problems due to accounting overhead
# in the kernel. We recommend using cgroups to do container-local accounting.
LimitNOFILE=infinity
LimitNPROC=infinity
LimitCORE=infinity
# Uncomment TasksMax if your systemd version supports it.
# Only systemd 226 and above support this version.
TasksMax=infinity
```

```
24 TimeoutStartSec=0
25 # set delegate yes so that systemd does not reset the cgroups of docker containers
26 Delegate=yes
27 # kill only the docker process, not all processes in the cgroup
28 KillMode=process
29
30 [Install]
31 WantedBy=multi-user.target
```

- **docker-network**

  Fichero usado por el script mk-docker-opts.sh

```
1 # /etc/default/docker-network
2 DOCKER_NETWORK_OPTIONS=
```