UNIVERSITAT POLITÈCNICA DE VALÈNCIA

_ TELECOM ESCUELA TÉCNICA VLC SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

Escuela Técnica Superior de Ingeniería de Telecomunicación
Universitat Politècnica de València

Automatización del despliegue, configuración y administración de servicios en GNU/Linux en sistemas de bajo coste

TRABAJO FIN DE GRADO

Grado en Ingeniería de Tecnologías y Servicios de Telecomunicación

Autor: Josep Ribes Rodríguez-Moldes
Tutor: Francisco José Martínez Zaldívar

Curso 2018-2019

Resumen

En este Trabajo de Fin de Grado se pretende automatizar distintos aspectos de la instalación de servicios en GNU/Linux en sistemas de bajo coste, concretamente en una Raspberry Pi. La primera tarea a automatizar es la instalación del sistema operativo. Para ello se han implementado unos *scripts* que permiten generar una imagen de disco preconfigurada de Arch Linux ARM.

Una vez instalada la imagen en la tarjeta SD, debido a que viene preconfigurada, podremos comunicarnos con la Raspberry Pi conectandola directamente al puerto Ethernet de nuestro ordenador personal, lo cual requiere automatizar también la modificación de la configuración de red de nuestro ordenador.

Otra tarea importante es el despliegue de servicios, por lo que se han implementado *scripts* capaces de desplegar algunos servicios y su configuración.

Así pues, el objetivo principal es implementar programas que realicen tareas que sería tedioso hacer manualmente, lo cual permite al administrador del servidor hacer su trabajo más rápido.

Palabras clave: Bash, Script, Administración de servidores, GNU/Linux, Raspberry Pi

Resum

En este Treball de Fi de Grau es preté automatitzar diferents aspectes de la instal·lació de serveis en GNU/Linux en sistemes de baix cost, concretament en una Raspberry Pi. La primera tasca a automatitzar es la instal·lació del sistema operatiu. Amb este propòsit, s'han implementat uns *scripts* que permeten generar una image de disc preconfigurada d'Arch Linux ARM.

Una volta hem instal·lat la image a la targeta SD, degut a que está preconfigurada, podem comunicar-nos amb la Raspberry Pi connectant-la directament al port Ethernet del nostre ordinador personal, la qual cosa requerix automatitzar també la modificació de la configuració de red del nostre ordinador.

Una altra tasca important és el desplegament de serveis, per la qual cosa s'han implementat *scripts* capaços de desplegar alguns serveis i la seua configuració.

Tenint en compte els factors anteriors, l'objectiu principal és implementar programes que realitzen tasques que seria tediós fer manualment, la qual cosa permet a l'admimnistrador del servidor fer el seu treball més ràpid.

Paraules clau: Bash, Script, Administració de servidors, GNU/Linux, Raspberry Pi

Abstract

In this Bachelor's Thesis we try to automate different aspects of the installation of services in GNU/Linux in low cost systems, specifically in a Raspberry Pi. The first task we have to automate is the installation of the operative system itself. In order to accomplish that, we have implemented scripts that generate a preconfigured Arch Linux ARM disk image.

Once we have installed the image in the SD card, as it has already been preconfigured, we can communicate with the Raspberry Pi connecting it to the Ethernet port of our personal computer. This requires the automation of the modification of the network configuration of our personal computer.

Another important task is the deployment of services, so we have implemented scripts capable of deploying some services and their configuration.

Taking into account the previous statements, the main aim of the project is to implement programs that perform tasks that would be tedious to do manually. This allows the server administrator to make his work faster.

Key words: Bash, Script, Server administration, GNU/Linux, Raspberry Pi

Índice general

CAPÍTULO 1
Introducción

1.1 Introducción

El modelo cliente-servidor es una de las arquitecturas más utilizadas a la hora de diseñar aplicaciones distribuidas. Consiste principalmente en que los llamados clientes hacen peticiones al servidor, y este realiza la tarea pedida y les devuelve una respuesta. De esta forma, el servidor puede compartir sus recursos (ya sea información, capacidad de cómputo o un servicio). Esta arquitectura es muy importante en el internet actual, puesto que prácticamente todos los servicios que utilizamos en nuestros *smartphones* y ordenadores la utilizan, como por ejemplo navegar por la web, utilizar el correo electrónico, servicios de mensajería o almacenamiento en la nube.

La palabra servidor se utiliza tanto para referirse a los programas que actúan como servidores como a las propias computadoras que tienen como objetivo principal actuar de servidor para otras computadoras. Dicho esto, nótese que los programas clientes y los servidores pueden estar instalados en la misma máquina, el modelo cliente-servidor es escalable a cada contexto. Sin embargo, en este proyecto, nos centraremos principalmente en el contexto en que cliente y servidor son computadoras distintas.

Normalmente, los servidores son la parte más complicada de implementar y mantener, debido a que muchos servicios exigen estar disponibles prácticamente siempre, y en caso de tener que realizar actualizaciones y verse obligados a parar un servicio en producción, el tiempo de restablecimiento ha de ser muy pequeño, por lo que la mayoría de las tareas han de automatizarse. Debido a ello, el campo de la administración de servidores es un campo en constante expansión, y los profesionales con tales conocimientos son cada vez más demandados en el mercado laboral.

Hoy en día para montar un servidor y aprender a configurarlo no se necesitan grandes recursos económicos. Simplemente, podemos adquirir un ordenador de placa reducida tal como una Raspberry Pi, y utilizarlo para nuestro proyecto. Además, debido a que estos ordenadores son capaces de ejecutar el mismo software que los grandes servidores (solo que con mucha menos potencia) los conocimientos y programas desarrollados para nuestro pequeño servidor sirven para servidores reales. Todo ello es posible gracias a la gran escalabilidad que proporciona el sistema operativo GNU/Linux, tal y como veremos más adelante.

1.2 Motivación

Normalmente, la Raspberry Pi se utiliza junto con el sistema operativo Raspbian, el cual es muy útil como herramienta educativa. Sin embargo, Raspbian tiene la mayoría

de su software preconfigurado, con lo cual el usuario no se ve forzado a tener que configurarlo él mismo y aprender cómo se hace. Además, viene con mucho software preinstalado, lo que hace que consumamos recursos de manera innecesaria si no lo vamos a utilizar. Esto no es nada bueno si queremos utilizar la Raspberry Pi como servidor.

En Raspbian, como en todos los sistemas operativos *Unix-like*, la configuración está distribuida en diferentes archivos de configuración, por lo que el usuario inexperto no sabe dónde mirar cuando quiere modificar algún parámetro. El objetivo de este trabajo es utilizar un sistema operativo más minimalista donde prácticamente nada venga preconfigurado, y a partir de ahí, automatizar la configuración y el despliegue de diferentes programas mediante pequeños *scripts*, de forma que toda la configuración realizada, además de estar automatizada, esté centralizada en un mismo sitio, lo que permite el aprendizaje, el estudio y la fácil modificación por parte de terceras personas.

1.3 Objetivos

Los principales objetivos de este trabajo son:

- Automatizar la instalación de una distribución GNU/Linux con la mínima interacción necesaria por parte del administrador.
- Automatizar la conexión con la Raspberry Pi vía Ethernet directamente con otro ordenador durante el desarrollo y pruebas realizadas.
- Realizar la automatización del despliegue de software y configuración de distintos servicios en GNU/Linux.
- Implementar una aplicación gráfica para poder controlar aspectos básicos de nuestro servidor y los servicios instalados desde otra computadora.
- Hacer el proyecto de forma que los programas desarrollados puedan ser reutilizados y estudiados por otras personas.

1.4 Contenido y estructura de la memoria

En el capítulo 2 (Estado del Arte) se explica las características básicas de la Raspberry Pi y se argumenta por qué la hemos elegido para este proyecto, además de hacer un listado de qué material se necesita para replicar el proyecto y por qué. También se ofrece una explicación histórica sobre Unix y GNU/Linux, se justifica la elección de la distribución que vamos a utilizar y se explican algunos conceptos necesarios para entender el resto del trabajo.

En el capítulo 3 (Diseño) se explica qué van a hacer los principales *scripts* desarrollados. También se ofrecen algunas alternativas que se podrían implementar y se justifica por qué se han descartado.

En el capítulo 4 (Desarrollo y uso de los *scripts* principales) se explica poco a poco cómo se van escribiendo los *scripts* que conforman la parte de instalación del sistema operativo. Tambén se explica el *script* que permite instalar la imagen de disco generada en la Raspberry Pi.

En el capítulo 5 (Configuración del ordenador personal) se explican los *scripts* utilizados en nuestro ordenador personal. El más relevante es el que permite que nos comuniquemos con la Raspberry Pi vía SSH conectandola directamente a través del puerto Ethernet de nuestro ordenador personal.

En el capítulo 6 (Servicios) se implementan *scripts* que permiten instalar diferentes servicios en la Raspberry Pi, así como la realización de unas mediciones con un RAID. También se explica el programa que implementa una interfaz gráfica capaz de encender y apagar servicios desde nuestro ordenador personal.

Por último, en el capítulo 7 (Conclusiones) se hace un repaso de lo que se ha hecho en este TFG y se proponen diferentes mejoras que podrían implementarse.

CAPÍTULO 2
Estado del arte

2.1 Hardware

2.1.1. Raspberry Pi

El ordenador que utilizaremos como servidor será una *Raspberry Pi 2 model B*, de ahora en adelante Raspberry Pi. Éste es un ordenador de placa reducida, lo cual significa que se trata de un ordenador completo en un solo PCB. Tradicionalmente, los ordenadores tienen una placa base, y su diseño modular permite cambiar componentes como el procesador, la memoria RAM, disco duro, ... Sin embargo, en un ordenador de placa reducida, todos los componentes se encuentran soldados en una sola placa.

Las principales ventajas de este tipo de ordenadores son las siguientes:

- Tienen un coste económico muy reducido
- Su tamaño es muy reducido
- El consumo energético es mucho más bajo respecto a otro tipo de computadoras, por lo cual son perfectos para proyectos de IoT (*Internet of Things* o *Internet de las Cosas* en castellano)

Además, en el caso de la Raspberry Pi, su enorme popularidad hace que haya una comunidad de usuarios muy grande, por lo que tiene mucha cantidad de información y soporte disponible en internet. La única desventaja es que es mucho menos potente que una computadora personal o un servidor normal.

Figura 2.1: Raspberry Pi 2 Model B

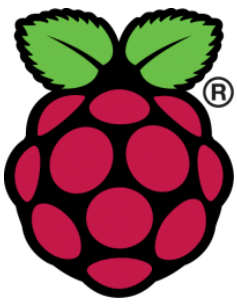

Figura 2.2: Raspberry Pi logo

2.1.2. Material necesario

Así pues, para nuestro proyecto necesitaremos:

- Una *Raspberry Pi 2 model B*
- Una tarjeta SD donde instalar el sistema operativo y los programas que utilizará la Raspberry Pi
- Un cable de Ethernet para conectar la Raspberry Pi a nuestro ordenador personal
- 4 USB del mismo tamaño para poder utilizar las características de RAID (*Redundant Array of Independent Disks* o *Array Redundante de Discos Independientes* en castellano)
- Un alimentador de 5V para la Raspberry Pi (suficiente con que llegue 2 A de intensidad de salida)
- Un ordenador personal desde el cual interactuar con la Raspberry Pi

2.2 Software

2.2.1. Unix

Unix es un sistema operativo que empezó a desarrollarse en los Bell Labs (centro de investigación de la AT&T) en el año 1969. Su precursor es Multics, un proyecto de Bell Labs junto con General Electric y el MIT, donde el objetivo era crear un sistema operativo de tiempo compartido, en el cual varios usuarios pudiesen utilizar la computadora al mismo tiempo, encargándose el sistema operativo de repartir el tiempo de cómputo entre los distintos usuarios. Fue desarrollado para la unidad central GE 645, una computadora de grandes dimensiones y gran coste económico.

En 1969 Bell Labs abandonó el proyecto lo cual llevó a Ken Thompson, quien había participado en el proyecto Multics, a empezar el desarrollo de un nuevo sistema operativo. El nuevo sistema operativo fue nombrado Unix y fue desarrollado principalmente por Ken Thompson y Dennis Ritchie, aunque contó con muchos otros participantes, como Brian Kernighan, quien es especialmente importante por su labor de documentación del sistema operativo y sus diferentes utilidades.

El desarrollo de Unix dio a luz el concepto de filosofía Unix, una filosofía de desarrollo de software donde los programas han de ser minimalistas, y colaborando entre varios programas se adquiere funcionalidad adicional. Una de las frases más populares para describir esta filosofía es que los programas deben de hacer solo una cosa y hacerla bien.

2.2.2. GNU/Linux

Uno de los problemas que tiene Unix es que es un sistema operativo propietario, es decir, el usuario no tiene acceso al código fuente, por lo cual no puede modificarlo para satisfacer sus necesidades.

En el 1983 Richard Stallman, en ese momento investigador en el laboratorio de inteligencia artificial del MIT, fundó el proyecto GNU, el cual tiene como objetivo escribir un clon de Unix completamente libre, es decir, que respete las Cuatro Libertades (definidas por la Free Software Fundation en el 1986, también fundada por Richard Stallman):

- La libertad de ejecutar el programa como se desee, con cualquier propósito (libertad 0).
- La libertad de estudiar cómo funciona el programa, y cambiarlo para que haga lo que usted quiera (libertad 1). El acceso al código fuente es una condición necesaria para ello.
- La libertad de redistribuir copias para ayudar a otros (libertad 2).
- La libertad de distribuir copias de sus versiones modificadas a terceros (libertad 3). Esto le permite ofrecer a toda la comunidad la oportunidad de beneficiarse de las modificaciones. El acceso al código fuente es una condición necesaria para ello.

Durante los siguientes años, el proyecto GNU logró escribir muchos programas y utilidades, tales como un compilador de C, las coreutils, la librería estándar de C (glib),... pero faltaba una pieza clave, el kernel. Esta pieza es esencial en cualquier sistema operativo, pues se encarga de interactuar con el hardware para que las aplicaciones del usuario sean ejecutadas. GNU Hurd era el proyecto que GNU tenían para implementar un kernel, pero faltaban desarrolladores, y todavía estaba lejos de estar completo.

Figura 2.3: GNU logo

En 1991, Linus Torvalds, un estudiante universitario, empezó a escribir un kernel compatible con Unix. Su intención era hacer un clon de Minix (una versión de Unix para estudiantes) para aprender sobre sistemas operativos. El proyecto ganó atención al poco tiempo, y en 1992 licenció el kernel, el cual fue bautizado como Linux, bajo la licencia GPL (GNU General Public License). Esto permitió que se pudiera utilizar en un mismo sistema Linux junto con todos los programas desarrollados por GNU, así como otros proyectos lanzados bajo una licencia libre, tales como el X Window System.

A partir de ese momento, nacieron distintas distribuciones de GNU/Linux, tales como Slackware o Debian. El termino distribución GNU/Linux se refiere a un conjunto de programas que conforman un sistema operativo se distribuyen conjuntamente y permiten a los usuarios descargarlos y instalarlos en su computadora. Normalmente los desarrolladores de la distribución también mantienen las aplicaciones más utilizadas por la comunidad y escriben documentación para que los usuarios puedan solucionar sus problemas.

Raspbian

Raspbian es una distribución GNU/Linux basada en Debian específicamente creada para la Raspberry Pi. Está enfocada al uso educacional, por lo que viene con muchos programas preinstalados, tales como `Python`, `Java` o `Scratch`. También viene con LXDE como gestor de escritorio, por lo que una vez instalado, ya viene preconfigurado para empezar a utilizarse.

La Raspberry Pi es una computadora con recursos bastante limitados, y Raspbian viene con mucho software preinstalado que no vamos a utilizar (por ejemplo el gestor de escritorio) y que consumirían parte de estos recursos, por lo que no es la mejor opción para utilizar la Raspberry Pi como servidor. Además Raspbian viene con el software preconfigurado, lo cual no nos conviene si queremos aprender sobre sistemas *Unix-like*. Es preferible utilizar una distribución más minimalista que nos fuerce a configurarla a nosotros. Así pues, no utilizaremos Raspbian en este proyecto.

Debian

Debian es una de las distribuciones más antiguas (fundada en el 1993) y más importantes. Cuenta con más de 51 000 paquetes de software precompilados disponibles en sus repositorios y listos para instalar, lo que la convierte en una de las distribuciones con mas programas disponibles. Además, ha sido portada a múltiples arquitecturas (probablemente la distribución más portada), por lo que muchas veces es la única opción que tenemos para instalar GNU/Linux en computadores con arquitecturas no demasiado populares. Otras distribuciones populares tales como Ubuntu o Kali Linux están basadas en Debian.

Utiliza un modelo *fixed-release*, lo cual puede hacer que muchas veces el software instalado no esté a la última versión. Sin embargo, Debian es una opción popular para servidores, debido a que cuenta con una versión de instalación mínima, donde solo está lo más básico para el funcionamiento del sistema operativo y unas pocas utilidades para que a partir de la instalación podamos instalar el software que deseemos. Además, Debian tiene fama de ser una distribución muy robusta y estable, deseable para servidores en producción.

Figura 2.4: Debian logo

Arch Linux

Arch Linux es una distribución nacida en el 2001 cuyo objetivo es crear un sistema operativo versátil destinado a usuarios con cierta experiencia con sistemas operativos *Unix-like*. La versatilidad la consigue debido a que la instalación base (mínimo número de programas que vienen preinstalados por defecto) es muy pequeña, lo cual permite al usuario instalar solo aquellos programas que realmente quiere tener. En vez de intentar ganar un gran número de usuarios haciendo la distribución *user-friendly*, Arch Linux se centra en satisfacer a los propios contribuyentes a la distribución. Tiene un espíritu *do-it-yourself*, ya que no cuenta con un instalador, y es el usuario el que debe leer la guía de instalación y instalar él mismo la distribución y solucionar los problemas que surjan, por lo que solamente instalarlo es una fantástica forma de aprender sobre GNU/Linux.

Es una distribución que utiliza un modelo *rolling-release*, lo cual significa que solo hay una versión de la distribución, es decir no existe Arch Linux 10.1 o 13.2. Los desarrolla-

dores de los programas que conforman Arch Linux trabajan en una única versión de la distribución, y actualizan esta frecuentemente. El usuario por su parte, debe ir actualizando el sistema periódicamente si quiere tener siempre sus programas actualizados.

Este modelo de *rolling-release* tiene la ventaja de que el software instalado siempre está actualizado, en contraposición con el modelo *fixed-release*. Esto hace que el sistema sea más seguro, pues todos los fallos de seguridad y *bugs* detectados que sean corregidos por el *upstream* de cada programa, siempre estarán instalados.

Una de las desventajas de este modelo es que el sistema tiene más tendencia a la inestabilidad que en el modelo *fixed-release*. Esto se debe a que estamos introduciendo cambios constantes en las aplicaciones y sistema operativo, lo cual puede hacer que el sistema tenga algún fallo después de una actualización. Por ello, es importante siempre leer el *feed* de noticias de la distribución antes de actualizar, pues si se sabe que surge un problema y hay una solución que hemos de aplicar manualmente, habrá una noticia con explicaciones detalladas.

El gestor de paquetes de Arch Linux es `pacman`, y se utiliza para descargar y instalar los programas ya compilados disponibles en los repositorios oficiales, además de actualizarlos. Con él también podemos instalar programas compilados por el usuario.

Otra de las características de Arch Linux es la AUR (*Arch User Repository* o Repositorio de Usuarios de Arch en castellano) un repositorio mantenido por la comunidad donde podemos encontrar PKGBUILD, con los cuales podremos instalar los programas presentes en el AUR utilizando la herramienta `makepkg`. La mayoría los programas disponibles aquí hay que compilarlos (las instrucciones de compilación están en el PKGBUILD), por lo que es más fácil que surjan errores que con los programas de los repositorios oficiales.

Figura 2.5: Arch Linux logo

2.2.3. Elección de la distribución

La distribución de GNU/Linux elegida para la realización del proyecto es Arch Linux. La principal razón para ello es la Arch Wiki, la principal fuente de documentación sobre Arch Linux. Cuenta con multitud de guías y artículos, muchos de los cuales también son útiles para usuarios de otras distribuciones. Por ejemplo, si queremos información sobre cómo funciona el sistema de usuarios y grupos de GNU/Linux, buscamos en la wiki y enseguida encontramos un largo archivo explicando multitud de cosas sobre el tema.

Arch Linux solo mantiene oficialmente la arquitectura amd64 (o x86_64), y el procesador de la Raspberry Pi es ARM, lo cual hace que no podemos utilizar la distribución oficial. Sin embargo, existe Arch Linux ARM, un proyecto que se encarga de portar Arch Linux a la arquitectura ARM. Este proyecto cuenta con un repositorio propio, puesto que los programas han de ser compilados para ARM. El proyecto no cuenta con un AUR propio, sin embargo, los usuarios pueden intentar adaptar el PKGBUILD del AUR y pasar los flags necesarios al compilador para que compile para ARM.

2.3 Conceptos necesarios

2.3.1. ¿Qué es un *script*?

La palabra *script* se utiliza con diferentes connotaciones. Por ejemplo, se utiliza para designar programas pequeños donde un lenguaje de propósito general de alto nivel y interpretado (como por ejemplo Python, Perl o Ruby) se utiliza como conexión entre diferentes módulos de un programa más grande. También se utiliza en el ámbito de la programación de videojuegos para designar los programas que controlan las acciones de un objeto o personaje.

En este proyecto, la palabra *script* se utiliza para designar programas que contienen instrucciones que podría realizar un operario humano. Por ejemplo, la tarea de instalación de Arch Linux ARM tiene un conjunto de llamadas a programas definido que podríamos ejecutar paso a paso cada vez que quisiéramos instalarlo. Sin embargo, si prevemos que vamos a realizar la tarea en diversas ocasiones, es deseable hacerlo ejecutando un solo programa que realice todos los pasos por nosotros.

Este tipo de *scripts* son escritos para un intérprete de comandos, también llamado *shell* en el entorno Unix. Los intérpretes de comandos suelen proporcionar un modo interactivo en el que el usuario introduce comandos y se le devuelve una respuesta, además de un modo no interactivo donde el intérprete de comandos es capaz de ejecutar los comandos escritos en un fichero, actuando como lenguaje de programación de *scripts*.

En Unix históricamente la *Bourne Shell* (`sh`) ha sido la *shell* más utilizada, sin embargo la implementación más utilizada no es la original, sino que es la *Bourne-again Shell* (`bash`), que es la implementación del proyecto GNU de sh. `bash` será la que utilicemos en nuestro proyecto debido a su gran popularidad y a que está disponible por defecto en la gran mayoría de los sistemas *Unix-like*.

2.3.2. ¿Qué es un *chroot jail*?

En los sistemas operativos *Unix-like*, se utiliza el comando `chroot` para crear un *chroot jail*, el cual consiste en cambiar el directorio raíz que ve un proceso, lo que permite que los programas que se ejecuten en un *chroot jail* solo tengan acceso a los archivos que estén por debajo del directorio que nosotros hemos especificado. Tiene diferentes usos, sin embargo el más importante para nosotros es el de ejecutar comandos "como si estuviéramos ejecutándolos en otro ordenador que todavía no está preparado para funcionar".

Por ejemplo, la instalación de Arch Linux se realiza desde un Live USB y se copian los ficheros esenciales para la utilización del sistema operativo a la partición raíz que utiliza nuestro sistema operativo. Una vez se han copiado dichos ficheros, se realiza un *chroot jail* al sistema de ficheros que hemos desplegado en la partición raíz de el sistema operativo que estamos instalado, permitiendo que ejecutemos comandos para configurar diferentes aspectos del sistema operativo, "como si estuviéramos ejecutándolos en el sistema operativo que estamos instalando", cuando en realidad los programas se ejecutan en el sistema operativo del Live USB.

2.3.3. ¿Qué es un imagen de disco?

Una imagen de disco es un tipo de fichero que consiste en una copia de el contenido de un dispositivo de almacenamiento como por ejemplo un disco duro, un CD o una memoria *flash*). En este proyecto las llamaremos simplemente imágenes, ya que suelen denominarse así en el contexto de instalación de sistemas operativos.

Normalmente, estos archivos se utilizan como medio de instalación de sistemas operativos, ya que permiten copiar un sistema de ficheros a un CD o memoria *flash* creando un llamado Live CD o Live USB. A partir de este Live CD o Live USB, podemos arrancar una computadora y copiar los archivos necesarios al disco duro (normalmente el proceso es guiado por medio de un instalador), permitiendo así instalar un sistema operativo.

Las imágenes también tienen otros usos, como por ejemplo realizar copias de seguridad o clonar discos.

2.3.4. ¿Qué es POSIX?

POSIX (*Portable Operating System Interface* o Interfaz de Sistema Operativo Portable) es un conjunto de estándares del *IEEE Computer Society* que definen una API (*Application Programming Interface* o Interfaz de Programación de Aplicaciones) para mantener la compatibilidad entre sistemas operativos *Unix-like*.

Este concepto suele aplicarse a las *shells*, y se dice que una *shell* es POSIX si cumple con el estándar. `bash`, por ejemplo, tiene un modo POSIX que puede utilizarse para comprobar que un *script* es portable, y un modo no-POSIX donde podemos utilizar características propias de `bash`.

Es importante tener presente el estándar POSIX en caso de que queramos que nuestros *scripts* sean realmente portables. No obstante, en este proyecto escribiremos nuestros *scripts* para `bash` sin preocuparnos de sí son POSIX o no, ya que `bash` está disponible por defecto en prácticamente todos los sistemas operativos `Unix-like`.

CAPÍTULO 3
Diseño

Antes de ponernos a implementar los diferentes programas que automaticen las diferentes tareas de este proyecto, debemos realizar un diseño general de qué es lo que vamos a implementar, donde se van a ejecutar los programas y por qué.

3.1 Programas de ejecución en el ordenador personal

En el proyecto utilizamos dos computadoras, la Raspberry Pi y el ordenador personal desde el cual interactuaremos con ella. Por conveniencia, el ordenador personal que utilizamos para desarrollar el proyecto tendrá instalado Arch Linux. La instalación y configuración de Arch Linux en un ordenador que tiene como uso ser un ordenador personal o de trabajo está fuera del alcance de este proyecto.

Los *scripts* desarrollados en este proyecto que se tengan que ejecutar en el ordenador personal y no en la Raspberry Pi están pensados para que funcionen en Arch Linux. No obstante, los *scripts* deberían funcionar en otras distribuciones, y en caso de que no funcionen, deberían hacerlo tras unas pocas modificaciones.

3.1.1. *Script* de instalación de dependencias: `installDependencies.sh`

En caso de querer utilizar los *scripts* que desarrollaremos a continuación, es necesario instalar las dependencias que estos utilizan en nuestro sistema operativo, ya que sin ellas, los *scripts* no funcionarán.

3.1.2. *Scripts* de generación de la imagen de disco y configuración básica: `setup.sh` y `configure.sh`

Este es el *scripts* que generará la imagen de Arch ARM y configurará los aspectos más básicos de el sistema operativo.

Normalmente, para escribir este *script* solo haría falta seguir las instrucciones de la guía de instalación de Arch Linux ARM. Sin embargo, si hacemos solamente esto, es incómodo trabajar con la Raspberry Pi para configurarla como servidor, debido a que por defecto no tiene una dirección IP fija. Los pasos que deberíamos seguir serían los siguientes

- Instalar Arch Linux ARM en la tarjeta SD
- Conectar la Raspberry Pi mediante Ethernet a nuestro ordenador o *router*

- Utilizar el servidor DHCP del *router* o el de nuestro ordenador para dar una dirección a la Raspberry Pi
- Averiguar qué dirección tiene la Raspberry Pi
- Conectarnos y configurar una dirección IP fija.

Este proceso es bastante engorroso en el caso de conectar la Raspberry Pi directamente a nuestro ordenador personal, ya que hemos de modificar la configuración de red de este dos veces.

En nuestro caso, utilizaremos dos *scripts*. El primero se encarga de crear la imagen y un *chroot jail*. Una vez hecho esto, utilizamos un segundo *script* que se encargará de, una vez dentro del *chroot jail*, configurará una dirección IP fija en la Raspberry Pi, así como actualizar el *software* y otros aspectos básicos de la configuración del sistema operativo.

Los pasos que seguiremos con nuestro método son los siguientes

- Ejecutar el *script* de *setup*.
- Ejecutar el *script* de configuración dentro del *chroot jail*.
- Copiar la imagen generada a la Raspberry Pi.
- Conectarnos y configurar una dirección IP fija.

Como puede verse, el proceso requiere menos interacción por parte del usuario, lo cual lo hace más rápido. Además tiene otras ventajas. Por ejemplo, la imagen resultante después de configurar la Raspberry Pi en el *chroot jail* puede ser copiada a múltiples tarjetas SD, permitiendo la configuración más rápida de múltiples Raspberry Pi. Si lo hiciéramos de la manera tradicional, tendríamos que modificar manualmente cada Raspberry Pi, lo cual sería muy tedioso.

Otra ventaja es que podemos actualizar el *software* a la última versión disponible desde el *chroot jail*. Esto es una ventaja cuando tenemos una conexión de Internet lenta. Si instaláramos Arch Linux ARM de la manera tradicional, deberíamos actualizar cada Raspberry Pi que instalamos, lo cual puede ser un proceso largo. Sin embargo, con nuestro método, podemos generar una imagen actualizada y instalarla múltiples veces.

Una desventaja del *chroot jail* es que no podemos activar servicios en `systemd` desde la interfaz que proporcionan comandos como `systemctl` o `timedatectl`, debido a limitaciones en la arquitectura de `systemd`. No obstante, existen maneras de conseguir el mismo resultado, o de activar servicios para que `systemd` los inicie cuando arranque por primera vez la Raspberry Pi, por lo que esto no es un problema grave.

3.1.3. *Scripts* de configuración de red: `setupNetwork.sh`

Para poder conectar la Raspberry Pi a nuestro ordenador personal y poder comunicarnos con ella tenemos dos opciones, utilizar DHCP o utilizar una dirección fija.

Si utilizamos DHCP, no sabremos a priori qué dirección IP obtendrá la Raspberry Pi, por lo que esta opción es menos adecuada para servidores. En cambio, si configuramos una dirección IP fija, sabremos qué dirección IP tendrá la Raspberry Pi cuando la conectemos, por lo que la comunicación será más sencilla.

Para ambas opciones, debemos utilizar *scripts* que configuren nuestro ordenador personal para comunicarse con la Raspberry Pi, por lo que desde el punto de vista de el usuario, ambas opciones son igual de complejas.

3.2 Programas de ejecución en la Raspberry Pi

Normalmente la configuración de servicios específicos, como por ejemplo un servidor FTP o un servidor `git`, cambian a que las necesidades cambian con el tiempo, por lo que estas configuraciones se realizarán una vez la Raspberry Pi tenga el sistema operativo instalado, tenga la configuración básica y podamos comunicarnos con ella.

En este proyecto implementaremos configuraremos los siguientes servicios:

- Servidor FTP
- Servidor SFTP
- Servidor git
- Servidor web (utilizando Node-RED)
- GUI sencilla que permita apagar y encender servicios.
- Configuración de RAID y test

Nótese que para comunicarnos con la Raspberry Pi nos hará falta SSH, que en sí mismo puede considerarse otro servicio. No obstante, la configuración SSH se hace junto con la configuración básica del sistema operativo, ya que es esencial para comunicarnos con la Raspberry Pi.

CAPÍTULO 4
Desarrollo de los *scripts* de instalación y configuración básica

En este apartado explicaremos el funcionamiento y implementaremos los *scripts* utilizados para la instalación y configuración básica del sistema operativo. Todos los *scripts* desarrollados en este trabajo pueden encontrarse en https://github.com/joseprrm/TFG, así como otros *scripts* que se han escrito pero no se han explicado en la memoria.

Nótese que no se explica la totalidad de los *scripts*, sino que se dan por entendidas las partes que ya se han explicado en *scripts* anteriores o que son irrelevantes.

4.1 Instalación de Arch Linux ARM

Ya que hemos decidido instalar Arch Linux ARM en nuestra Raspberry Pi, debemos visitar su página web para obtener las instrucciones de instalación y el *software* necesario. Arch Linux ARM cuenta con instrucciones especificas para la Raspberry Pi 2, las cuales se encuentran aquí https://archlinuxarm.org/platforms/armv7/broadcom/raspberry-pi-2. Debemos ayudarnos de estas instrucciones para crear nuestro *script* de instalación.

Según las instrucciones solamente hace falta crear la tabla de particiones en la SD, formatear las particiones adecuadamente y descomprimir un archivo .tar.gz en la SD, aunque como hemos explicado anteriormente, nosotros no seguiremos al pie de la letra las instrucciones de instalación, sino que modificaremos el proceso para conseguir crear una imagen ya configurada. Así pues, crearemos un *script* con todas estas instrucciones para no tener que hacerlo cada vez que queramos instalar el sistema operativo.

La técnica utilizada para la creación del *chroot jail* y la emulación de ARM se basa en gran medida en [1].

setup.sh

```
1 #!/bin/bash
```

La primera línea de todo *script* ha de tener es la secuencia #! (llamada *shebang* o *hashbang*) seguido de la ruta hacia el intérprete que se ha de utilizar para ejecutar el *script*. Esto se debe a que Unix no sabe a priori qué intérprete utilizar, ya que el *script* puede estar escrito en distintos lenguajes. Esta línea no será ejecutada por el *script*, ya que en `bash`, el carácter # se utiliza para los comentarios. En nuestro caso, utilizamos `bash`, y el ejecutable se encuentra en `/bin/bash` (como en la mayoría de los sistemas *Unix-like*).

setup.sh

```
set -e
```

El programa `set` se utiliza para modificar opciones de la *shell*. Nótese que no es especifico de `bash`, sino que cualquier *shell* POSIX debe implementar estas opciones. En concreto `set -e` se utiliza para detener la ejecución del *script* si algún comando falla.

El código de retorno, también llamado estado de salida, es un número de 0 a 255 que el comando retorna cuando finaliza su ejecución. El código 0 indica normalmente que la ejecución ha sido satisfactoria, y un número más grande suele indicar algún tipo dé error. Así pues, con la opción `set -e` el *script* se detendrá si algún código de retorno es mayor que 0.

setup.sh

```
function usage {
    program_name=$0
    cat <<EOF
This script installs Arch Linux ARM on specified drive.
It must be executed as the root user. Do not use it with sudo, use su.
The tarball containing the filesystem can be downloaded previously with:
    wget http://os.archlinuxarm.org/os/ArchLinuxARM-rpi-2-latest.tar.gz

Dependencies: dosfstools TOOD

Usage: $program_name imageName tarball callback
        or
       $program_name [--help|-h]
  Where:
    imageName  > name of the iso this programs creates
    tarball   -> tarball containing the root filesystem
    callback  -> script that is executed inside the chroot jail

$program_name [--help|-h] displays this message
EOF
}
```

Normalmente, es útil documentar los *scripts* con mensaje que se ejecute cuando no introducimos bien los argumentos del *script*, usamos opciones incorrectas o ejecutamos las opciones `-help` o `-h`. Típicamente el mensaje que se muestra contiene una descripción corta de qué hace el programa y una explicación de cómo utilizarlo.

Por comodidad, es preferible escribir una función que muestre el mensaje por pantalla, y después llamar a esta función en caso de que queramos que el mensaje se imprima. Esto se hace con: `function <nombre de la función>{ ... }`.

Para asignar variables en `bash`, simplemente hacemos `variable=valor`. Es típico poner los nombres de las variables en minúsculas, ya que así nos aseguramos de que no interfieren con las variables del sistema, las cuales siempre van en mayúsculas.

En `bash` el nombre del *script* se encuentra en la variable especial `0`. Para expandir el valor de las variables (obtener el valor) en `bash` se utiliza el operador $. Así pues, asignamos el valor de la variable 0 a la variable program_name para dar nombre un nombre con más sentido a la variable que tiene el nombre del *script*.

Lo siguiente que hacemos es imprimir en pantalla el mensaje de ayuda. Para ello utilizamos el programa `cat` junto con un *here document* para mostrar un mensaje multilínea donde las variables que haya dentro puedan ser expandidas. `cat` es un programa muy simple, lo único que hace es concatenar uno o más fichero a la salida estándar. Como por defecto la salida estándar es la *shell*, mostrará el contenido del fichero por pantalla. El *here document* (la parte de << EOF) se utiliza cuando queremos que por la entrada

estándar de un programa entre un flujo de texto donde se se pueda expandir variables. La EOF (end of file) se utiliza para indicar cuándo finaliza el *here document*. En realidad EOF puede ser cualquier secuencia de texto, pero se suele utilizar EOF por convenio.

setup.sh

```
1
2 if [ "$1" == "-h" ] || [ "$1" == "--help" ]
3 then
4     usage
5     exit 0
6 fi
7
8 if [ -z "$1" ]
9 then
10    echo "No image name provided, please provide one" 1>&2
11    usage 1>&2
12    exit 1
13 fi
```

El siguiente código se utiliza para notificar al usuario cuando el usuario no ha introducido todos los parámetros o cuando queremos utilizar la opción de ayuda. Las variables $1, $2, ... son parámetros posicionales, y contienen los argumentos que le damos al *script* cuando lo llamamos. `-z <string>` en `bash` devuelve *True* si un string tiene tamaño 0, así pues la utilizamos para comprobar si se han pasado los argumentos o no al *script*, ya que en `bash` podemos expandir variables no declaradas sin que de error (con las opciones por defecto). `exit` se utiliza para terminar la ejecución del *script* y devolver el código de retorno adecuado.

setup.sh

```
1 # stop if a variable is unset
2 set -u
```

`set -u` se utiliza para que cuando una variable no haya sido inicializada y intentemos expandirla, el script finalice su ejecución. Es útil para evitar errores que no prevemos y que puedan dañar el sistema de alguna forma. Por ejemplo, si ejecutáramos `rm -r /${path}` y la variable no estuviera inicializada, borraríamos todo el árbol de directorios, en cambio, con `set -u`, solo dará un error y terminará el programa.

setup.sh

```
1 image=$(realpath $1)
2 tar_archive=$(realpath $2)
3 callback=$(realpath $3)
```

Después de los `if` anteriores sabemos que los argumentos del comando existen, así que los asignamos a otras variables con un nombre significativo. Además utilizamos el comando `realpath` para obtener la ruta absoluta de los argumentos. Esto es una buena práctica, ya que si después cambiáramos a otro directorio y intentáramos utilizar estas variables, no podríamos localizar los ficheros, ya que las rutas serían relativas al directorio en el que estábamos anteriormente. Además, si el *script* moviera o borrara archivos, el resultado podría ser desastroso.

```
1 fallocate -l 2000M ${image}
```

Ahora es cuando comenzamos con el proceso de creación de la imagen. Primeramente utilizamos `fallocate` para crear un fichero vacío que ocupe 2000MB. Este valor debe ser superior a lo que prevemos que va a ocupar la instalación del sistema operativo. El `tar` que contiene el sistema de ficheros ocupa más o menos 365MB, por lo que una

vez descomprimido y al realizar las actualizaciones pertinentes, debería haber espacio suficiente.

Este fichero vacío contendrá la imagen una vez realicemos el resto del proceso, por lo que de ahora en adelante lo llamaremos imagen.

setup.sh

```
sed -e 's/\s*\([\+0-9a-zA-Z]*\).*/\1/' << EOF | fdisk -W always ${image}
  o # clear the in memory partition table
  n # new partition
  p # primary partition
  1 # partition number 1
    # default - start at beginning of disk
  +100M # 100 MB boot parttion
  t # type
  c # set the first partition to W95 FAT32 (LBA) type
  n # new partition
  p # primary partition
  2 # partition number 2
    # default - start next to the partiton1
    # default - until the end of the disk
  w # write the partition table
  q # and we're done
EOF
```

Esta parte del *script* se utiliza para automatizar la creación de la tabla de particiones de la imagen (cuando se instale será la tabla de particiones de la tarjeta SD). `fdisk` es el programa que se utiliza para ello, pero normalmente se utiliza como un programa interactivo en el que el usuario introduce las instrucciones a mano. Sin embargo, utilizamos una *pipeline* para introducir por la entrada estándar de `fdisk` los comandos necesarios especificados por las instrucciones de instalación. La *flag* `-W always` se utiliza para que `fdisk` elimine siempre la información residual de la anterior tabla de particiones y sistemas de ficheros anteriores sin preguntar al usuario antes. La técnica utilizada se ha obtenido en [2].

A la izquierda de la *pipeline* utilizamos el programa `sed` (*Stream EDitor*) para quitar los comentarios a la derecha de cada comando de `fdisk`. `sed` es un programa basado `ed`, un editor de texto orientado a líneas y uno de los primeros programas de Unix. Basándose en la capacidad de *scripting* de `ed` nació `sed`, el cual aplica un comando a todas las líneas que le entran por la entrada estándar. En nuestro caso, la entrada estándar es el *here document* que contiene los comandos de `fdisk`, y el comando que ejecuta a cada línea es una substitución, donde se utilizan expresiones regulares para quedarnos solo con la primera palabra de cada línea, pudiendo así comentar el código sin que los comentarios acaben en la entrada estándar de `fdisk`.

setup.sh

```
loop_device=$(losetup -Pf --show ${image})
```

A continuación utilizamos el comando `losetup` para asociar la imagen con un *loop device*, lo cual hace que el fichero sea visto por el sistema operativo como un *block device* Esto quiere decir que el sistema operativo nos permite manipular el contenido de la imagen como si fuera una memoria externa, como por ejemplo un disco duro o una memoria *flash*. Además, guardamos en una variable qué *loop device* es el que estamos utilizando.

setup.sh

```
boot_partition=${loop_device}p1
ext4_partition=${loop_device}p2

mkfs.vfat $boot_partition
```

```
mkfs.ext4 $ext4_partition
```

Una vez tenemos la imagen asociada con un *loop device*, procedemos a formatear las particiones con el sistema de ficheros indicado en la guía de instalación.

setup.sh

```
mount $boot_partition $boot
mount $ext4_partition $chroot

bsdtar -xpf $tar_archive -C $chroot
sync

mv ${chroot}/boot/* ${boot}
```

Ahora montamos las particiones en nuestro sistema de ficheros con `mount`, extraemos el *tarball* con `bsdtar` en la partición raíz, utilizamos `sync` para vaciar los *buffers* de escritura y forzar que los ficheros se escriban y movemos el contenido de `boot` a la partición de `boot`, tal como dicta la guía de instalación.

```
if [ ! -f /proc/sys/fs/binfmt_misc/arm ]; then
    echo ':arm:M::\x7fELF\x01\x01\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\
        x02\x00\x28\x00:\xff\xff\xff\xff\xff\xff\xff\x00\xff\xff\xff\xff\
        xff\xff\xff\xff\xfe\xff\xff\xff:/usr/bin/qemu-arm-static:' > /proc/
        sys/fs/binfmt_misc/register
fi
```

Si el sistema operativo que estubiéramos instalando fuera de arquitectura `amd64`, ya podríamos crear el *chroot jail*, ya que nuestro ordenador personal también es `amd64`. Sin embargo, queremos hacer `chroot` a un sistema de ficheros cuyos programas están en arquitectura `arm`. Esto hace que nuestro procesador no pueda ejecutarlos directamente. Para poder ejecutarlos debemos emular la arquitectura *arm*, por lo que utilizamos `qemu` para ello.

En el comando anterior, lo que hacemos es registrar `arm` en el módulo del *kernel* `binfmt_misc` (si no estaba ya registrado). Este módulo permite reconocer la arquitectura de los ejecutables y preparar el espacio de usuario de manera acorde. Nótese que este no es el emulador en sí, sin embargo, es necesario para que la emulación funcione sin intervención manual por parte del usuario al cambiar de un programa a otro.

setup.sh

```
cp /usr/bin/qemu-arm-static ${chroot}/usr/bin
```

Copiamos el emulador `qemu-static-arm` a la ruta donde `binfmt_misc` espera que esté. Esta ruta se la hemos pasado a `binfmt_misc` al registrar `arm`.

setup.sh

```
mount -t proc /proc ${chroot}/proc
mount -o bind /dev ${chroot}/dev
mount -o bind /dev/pts ${chroot}/dev/pts
mount -o bind /sys ${chroot}/sys
```

Realizamos un *bind mount* de los directorios con sistemas de ficheros virtuales y otros recursos importantes de nuestro ordenador personal en directorio donde crearemos el *chroot jail*. Esto se hace debido a qué algunos programas utilizan estos recursos para obtener información o cambiar parámetros del sistema, por lo que si no los montamos, nos encontraremos con fallos.

Ahora ya podríamos hacer `chroot` y ejecutar `bash` dentro de la `chroot jail`, emulando *arm* correctamente.

setup.sh

```
# we will need enough entropy to be able initialize the pacman keyring
if ! systemctl is-active --quiet rngd ; then
    systemctl start rngd
fi
```

Como en el *script* de configuración vamos a inicializar claves GPG, necesitamos que el *kernel* tenga suficiente entropía para poder realizar el proceso rápidamente. En computación, entropía se refiere a la cantidad de eventos aleatorios que el sistema operativo ha recolectado, y que utilizará para generar números aleatorios. Para ello, nos aseguramos de que el *daemon* `rngd` esté funcionando, y este se encargará de poner en funcionamiento el generador de números aleatorios *hardware* de nuestra computadora para que genere suficiente entropía para el *kernel*.

Podemos ver el nivel de entropía ejecutando `cat /proc/sys/kernel/random/entropy_avail`.

setup.sh

```
chroot ${chroot} bash <${callback}
```

Finalmente, podemos ejecutar `chroot`. La técnica que utilizamos para ejecutar un *script* dentro del *chroot jail* de manera automática es conocida como *callback*, y consiste en pasar el nombre del *script* de configuración al *script* principal y hacer que la entrada estándar del *chroot jail* sea el *script* de configuración.

4.2 *Script* de configuración

En este apartado explicaremos qué hará el *script* de configuración básica, así como su implementación. Este es el *script* que pasamos al *script* de instalación y que se ejecutará dentro del *chroot jail*:

4.2.1. SSH

La implementación de SSH que utilizaremos es `openssh`. Además de su utilidad principal que es permitir comunicarse con otra computadora a través de una *shell* y donde la comunicación está encriptada, cuenta con muchas otras utilidades, tales como un servidor SFTP o el programa `scp`, que permite copiar ficheros de una computadora a otra (con la comunicación encriptada) de manera muy sencilla.

La configuración por defecto que tiene Arch Linux ARM permite utilizar las credenciales de los usuarios creados en el sistema para autenticarse. Por defecto solo podemos hacer esto con el usuario "alarm", ya que es el único usuario por defecto a parte de "root". "root" es el denominado superusuario, y tiene un control total sobre la computadora. Los desarrolladores de `openssh` han desactivado la opción que permite autenticarse como "root".

Utilizar las credenciales de los usuarios no es una buena idea normalmente, ya que de una forma u otra, hemos de enviar la contraseña a través de la red para poder autenticarnos, lo que hace que método sea inseguro y susceptible a ataques. `openssh` permite utilizar criptografía asimétrica o criptografía de clave pública para autenticarnos. Este método consiste en que el servidor tiene nuestra clave pública y nosotros la clave privada, la cual solo es conocida por nosotros, y para autenticarnos en el servidor, nos manda un reto generado a partir de la clave pública que solamente podemos resolver si conocemos la clave privada. De esta forma, nunca se envía nuestra clave a través de la red. Esto

es una simplificación del proceso, la explicación rigurosa matemáticamente queda fuera del alcance de este proyecto.

Uno de los problemas de seguridad que presenta este sistema es que, si alguien consigue robarnos la clave privada, puede utilizarla para autenticarse sin nuestro consentimiento. No obstante, `openssh` permite utilizar una contraseña para acceder a nuestra clave privada, de forma que la clave privada se guarde cifrada y solo se pueda conocerla si la desciframos previamente con nuestra contraseña. Esto soluciona el problema de seguridad ocasionado por el robo de la clave privada. No obstante, es importante que la contraseña sea segura y de no menos de 10 caracteres de longitud (según el manual de `ssh-keygen`, ya que en caso de robo de la clave privada se podrían realizar ataques de fuerza bruta o de diccionario para intentar averiguar la contraseña.

Otra de las ventajas que tiene la criptografía asimétrica en SSH es que no tenemos que recordar la clave privada nosotros mismos, sino que el ordenador se encarga de todo el proceso de autenticación. Por ejemplo, si nuestra clave privada no tiene contraseña, podemos hacer `ssh usuario@ip` y automáticamente obtener la *shell*. En caso de tener nuestra clave privada cifrada, podemos utilizar un programa llamado `keychain` que nos permite que con escribir una vez nuestra contraseña, podamos utilizar la clave privada sin volver a poner la contraseña cada vez que utilizamos SSH hasta que reiniciemos el ordenador.

4.2.2. Creación del usuario de administrador

Arch Linux ARM viene con el usuario "alarm" como punto de entrada para empezar a configurar el sistema. Sin embargo, es mejor que creemos nuestro propio usuario de administrador desde el cual accedemos a"root". Posteriormente, cuando ya tengamos configurado el nuevo usuario, borraremos "alarm".

4.2.3. `.bashrc`

Si nos conectamos a la Raspberry Pi con el comando `ssh` obtenemos acceso al programa `bash` ejecutándose en la Raspberry Pi. Podemos escribir cualquier comando y la Raspberry Pi lo ejecutará, pero es posible que si utilizamos un emulador de terminal distinto a `xterm` haya problemas con los caracteres de control que se encargan de controlar cómo se ve el texto en la pantalla. Por ejemplo, si utilizamos `urxvt` como emulador de terminal, cuando borremos caracteres, los cambios no se verán reflejados en el texto que aparece en pantalla. Tampoco podremos utilizar "`Control-L`" para borrar la pantalla. Esto se debe a que la variable de entorno `TERM` que tiene la Raspberry Pi por defecto es la misma que tiene el proceso de `bash` desde el que llamamos a `ssh`. En la realización de las pruebas, el emulador de terminal utilizado es `urxvt`, y la variable TERM es `rxvt-unicode-256color`, lo que hace que el proceso de `bash` de la Raspberry Pi no entienda bien los caracteres de control.

Para solucionar este problema, debemos modificar la variable `TERM` cada vez que comience un proceso de `bash` interactivo en la Raspberry Pi. `bashrc` es el fichero que siempre se ejecuta cuando `bash` empieza, así que ese es un buen sitio para poner el valor de `TERM` que queramos. El valor que le daremos será `TERM=xterm`, y de esta manera todo funcionará correctamente. `xterm` es uno de los emulador de terminal más utilizados y podría considerarse casi un estándar, por lo que al decirle que estamos utilizando `xterm` los caracteres de control funcionan.

Además, también añadiremos el comando `sudo su` al final del `.bashrc` para que automáticamente obtengamos privilegios de superusuario, ya que planeamos utilizar el

usuario que vamos a configurar como punto de entrada a "root". Al no estar instalado el programa `sudo`, al principio dará un error sin mayores consecuencias y no escalaremos privilegios automáticamente, pero más tarde cuando instalemos `sudo` y modifiquemos adecuadamente las opciones, nada más iniciar la *shell*, escalaremos privilegios sin tener que escribir nada.

4.2.4. Implementación del *script*

configure.sh

```
1 ln -sf /usr/share/zoneinfo/Europe/Madrid /etc/localtime
```

Configuramos la zona horaria donde se va a utilizar la Raspberry Pi, la cual en nuestro caso es la de Madrid. La manera de configurarla es creando un *soft link* en `/etc/localtime` que apunte a el fichero que simboliza la zona horaria. Esta forma de configurar la zona horaria está documentada en [3]. Este tipo de ficheros se conocen como `tzfiles`.

Un *soft link*, también conocido como *symlink* o *symbolic link*, es un tipo de fichero que contiene una referencia a otro. Esta referencia no es al fichero en sí, sino a la ruta del fichero, por lo que si borramos el fichero al que apunta un *soft link*, perderemos el archivo y el *soft link* apuntará a nada. Existe otro tipo de *link* llamado *hard link*, donde la referencia sí que es al fichero en sí, aunque tiene la desventaja de no poder hacer *hard links* a directorios (ya que podríamos crear bucles infinitos al ejecutar programas que recorran el árbol de directorios).

configure.sh

```
1 ln -sf /usr/lib/systemd/system/systemd-timesyncd.service /etc/systemd/
      system/sysinit.target.wants/systemd-timesyncd.service
```

Activamos el cliente NTP para que sincronice se sincronice nuestra hora con un servidor NTP. Nótese que para realizar esta acción, normalmente lo que haríamos sería ejecutar `timedatectl set-ntp true`. Sin embargo, como estamos dentro de un *chroot jail*, no podemos ejecutar esté comando, ya que *systemd* no está disponible. Así que realizamos la acción que haría *systemd*, que es hacer el *soft link* anterior. Así pues, cuando encendamos la Raspberry Pi, el cliente NTP se encenderá automáticamente.

configure.sh

```
1 echo "en_US.UTF-8 UTF-8">>/etc/locale.gen
2 locale-gen
```

Configuramos el idioma en el que queremos que esté nuestro sistema operativo, así como la codificación de caracteres. Esta configuración se conoce como `locales`. La ponemos en inglés y utilizando UTF-8, ya que la mayoría de programas están escritos con el inglés como idioma principal y UTF-8 da soporte a la mayoría de caracteres existentes. Esta es la configuración con la que menos problemas tendremos debido a los *locales*.

configure.sh

```
1 rm /etc/resolv.conf
2 echo "nameserver 8.8.8.8" >/etc/resolv.conf
```

En el fichero `resolv.conf` configuramos el servidor DNS que vamos a utilizar. Debido a algunos problemas generados por el la emulación ARM, no podemos reescribirlo, sino que debemos borrarlo y escribir otra vez el fichero.

configure.sh

```
pacman-key --init
pacman-key --populate archlinuxarm
```

Inicializamos las claves de pacman `pacman`. Este es el paso que necesita suficiente entropía, ya que sino tardaría mucho tiempo en completarse. Como en el *script* anterior nos hemos asegurado de que haya suficiente, la ejecución será rápida.

configure.sh

```
yes | pacman -Syu
```

Actualizamos todos los paquetes para que estén en la última versión. Utilizamos el comando `yes` para que cuando nos pregunte si queremos seguir con el proceso, automáticamente `pacman` reciba el carácter "y" y el proceso siga sin intervención del usuario.

configure.sh

```
yes | pacman -S sudo cowsay vim
```

Instalamos los programas que prevemos que vamos a utilizar.

configure.sh

```
cowsay "Welcome to the Raspberry Pi Arch Linux Server" >/etc/motd
```

Utilizamos el programa `cowsay` para generar un mensaje de bienvenida que veremos cuando nos conectemos con la Raspberry Pi por SSH.

configure.sh

```
cat >/etc/systemd/network/eth0.network <<EOF
[Match]
Name=eth0

[Network]
Address=192.168.1.40
Gateway=192.168.1.50
EOF
```

Configuramos la interfaz Ethernet escribiendo en `eth0.network`. `eth0` es el nombre de la interfaz Ethernet de la Raspberry Pi, que en este caso no utiliza los nombres predecibles de `udev`, sino que utiliza la nomenclatura tradicional de llamar a los interfaces Ethernet "ethX", siendo "X" 0, 1 ,2 ... dependiendo del orden en que el sistema detecte la interfaz. Como la Raspberry Pi solo tiene una interfaz Ethernet, siempre será `eth0`.

`Address` y `Gateway` son las direcciones IP de la Raspberry Pi y la puerta de enlace predeterminada. Esta información la utilizará `networkd` para asignar la dirección IP al arrancar la Raspberry Pi y para configurar apropiadamente la tabla de *enrutamiento*.

Es preferible configurar una dirección IP estática a utilizar DHCP (que es como viene configurado Arch Linux ARM por defecto), de forma que la Raspberry Pi siempre tenga la misma dirección IP. De esta forma, será mucho más sencillo acceder a ella y podremos realizar configuraciones más avanzadas, tales como activar el *forwarding* IP (hacer que nuestro ordenador personal haga de router de la Raspberry Pi) de manera mucho más sencilla.

configure.sh

```
echo '%wheel ALL=(ALL) NOPASSWD: ALL' >> /etc/sudoers
```

Escribimos en el fichero `sudoers` para configurar los permisos de `sudo`, el programa que se utiliza para escalar privilegios (ejecutar un comando como `root`) de forma temporal.

Hacer esto es poco recomendable si no estamos seguros de lo que hacemos, ya que si la sintaxis de `sudoers` no es correcta, no podremos escalar privilegios y por tanto no podremos modificar el fichero, dejando al programa `sudo` sin poder utilizarse. Dependiendo de los permisos que haya configurados en en ordenador, esto puedes ser desastroso, ya que es posible que se necesite volver a instalar el sistema operativo para solucionar el problema.

En nuestro caso, solo debemos modificar una línea para conseguir que los usuarios del grupo `wheel` puedan utilizar `sudo` sin tener que escribir ninguna contraseña, por lo que al ser una modificación menor y al ser durante la instalación podemos permitirnos modificar `sudoers` a mano.

configure.sh

```
userdel -f -r alarm
useradd -m -G wheel -s /bin/bash admin
```

Borramos el usuario por defecto `alarm` y creamos uno nuevo llamado `admin`, el cual utilizaremos para conectarnos a la Raspberry Pi por SSH.

configure.sh

```
mkdir /home/admin/.ssh
ssh-keygen -f /home/admin/.ssh/newKey

mv /home/admin/.ssh/newKey.pub /home/admin/.ssh/authorized_keys
chmod 644 /home/admin/.ssh/authorized_keys
chown -R admin:admin /home/admin/.ssh
```

Generamos las claves pública y privada que utilizaremos para conectarnos con la Raspberry Pi vía SSH. Hemos de tener especial cuidado en poner bien los permisos adecuados. Una vez finalice este *script*, hemos de acordarnos de copiarnos la clave privada a nuestro ordenador personal y borrarla de la imagen. No podemos hacer esto ahora, ya que estamos dentro del *chroot jail*.

configure.sh

```
cat >/etc/ssh/sshd_config <<EOF
Port 28374

AuthorizedKeysFile /etc/ssh/authorized_keys/%u  .ssh/authorized_keys

PasswordAuthentication no
PermitEmptyPasswords no

ChallengeResponseAuthentication no
PubkeyAuthentication yes
UsePAM yes
PrintMotd no # pam does that
EOF
```

Configuramos el servicio `sshd` (el servidor SSH) para que solamente utilice criptografía asimétrica. Además, cambiamos el puerto por defecto de SSH (puerto 22) a un puerto más alto. Esto es útil para evitar ataques de fuerza bruta que realizan los *crackers*, ya que normalmente atacan solamente el puerto 22. Además, así evitamos que el `sshd` procese todos los intentos fallidos de acceso, ya que la conexión se cortará antes de que le llegue.

configure.sh

```
cat >/home/admin/.bashrc <<EOF
#
# ~/.bashrc
```

```
#

# If not running interactively, don't do anything
[[ \$- != *i* ]] && return

export TERM=xterm
export EDITOR=vim

alias ls='ls --color=auto'
PS1='[\u@\h \W]\$ '

sudo su
EOF
```

Configuramos el `bashrc`. Es importante que escapemos el carácter "$", ya que sino intentará hacer una expansión de variable y después el *bashrc* no se ejecutará cuando nos conectemos a la Raspberry Pi.

4.3 *Script* de escritura de la imagen en la tarjeta SD

Una vez tenemos los *scripts* anteriores, podemos generar una imagen con la configuración que queramos. El siguiente paso es escribir la imagen en una tarjeta SD.

Debido a que la imagen ocupa originalmente lo que hayamos estimado anteriormente al utilizar el comando `fallocate`, una vez copiada la imagen a la tarjeta SD, vemos que el tamaño de la partición *root* no ocupa todo el tamaño de la SD. Una posible solución sería que cuando utilizamos `fallocate`, generáramos un archivo igual de grande que la SD. Sin embargo, tendríamos que generar un archivo diferente para cada tamaño de SD, lo cual es incómodo. Además, cuando más grande es el archivo, más tarda en escribirse, lo que hace que el proceso de copiado sea más largo.

La solución que proponemos es copiar una imagen pequeña en la SD (del tamaño estimado anteriormente) y posteriormente hacer más grande el sistema de ficheros de la partición *root*.

4.3.1. Implementación del *script*

burnSD.sh

```
dd if=$image of=$target bs=4M status=progress oflag=sync
```

Escribimos el contenido de la imagen en la tarjeta SD utilizando el comando `dd`. Este es un comando básicos que se utilizan para copiar archivos de un sitio a otro. Utilizamos el argumento `status=progress` para que se muestre el progreso durante la copia. Utilizamos el argumento `oflag=sync` para que los datos se escriban en el disco y no se queden en los *buffers* de escritura.

burnSD.sh

```
sync
```

Aunque hayamos utilizado `oflag=sync`, es conveniente utilizar `sync` para asegurar que se completa la escritura.

burnSD.sh

```
# Do not put blank lines until EOF (unless you know what you are doing),
    because they will be interpreted as a <CR> inside fdisk
```

```
sed -e 's/\s*\([\+0-9a-zA-Z]*\).*/\1/' << EOF | fdisk -W always ${target}
  d # delete
  2 # partition number 2 (the root partition)
  n # new partition
  p # primary partition
  2 # partition number 2
    # default - start next to the partiton1
    # default - until the end of the disk
  w # write the partition table
EOF
```

Hacemos más grande la partición *root* borrándola y volviéndola a crear, de forma que ocupe hasta el final de la tarjeta SD.

burnSD.sh

```
partprobe
```

Nos aseguramos de que el `kernel` lee la tabla de particiones de la tarjeta SD. Esto es importante, ya que al ejecutar todo en un *script*, los comandos se ejecutan uno detrás de otro sin dejar tiempo entre comando y comando. Si no forzáramos que se vuelvan a leer las tablas de particiones, los comandos siguientes fallarían, ya que intentarían utilizar la tabla de particiones antigua.

burnSD.sh

```
e2fsck -y -f  ${root_partition}

resize2fs  ${root_partition}
```

Hacemos que el sistema de ficheros ocupe hasta el final de la partición *root*.

burnSD.sh

```
sync
```

Por último, nos aseguramos de que todos los datos se escriben en disco.

4.4 Uso de los *scripts*

El proceso que hay que seguir para ejecutar estos *scripts* es muy simple.

- Ejecutar `setup.sh <nombreImagen><tarball>configure.sh`. Nótese que `configure.sh` puede ser cualquier otro *script* que creemos.
- Ejecutar `burnSD.sh <nombreImagen><tarjetaSD>`.

Una vez ejecutados estos comandos, ya tenemos Arch Linux ARM en la Raspberry Pi con la configuración deseada.

CAPÍTULO 5
Configuración del ordenador personal

5.1 *Script* de instalación de dependencias

El *script* que se encarga de instalar las dependencias que vamos a utilizar es el siguiente:

installDependencies.sh

```
su root<<EOF
pacman -S dhcp iptables dosfstools net-tools python python-kivy wget
    sshpass partprobe rng-tools haveged
EOF
```

Utilizamos *pacman* para descargar los programas necesarios. El *script* está pensado para ser utilizado como un sin privilegios, por lo que ejecutamos *su* para escalar privilegios.

installDependencies.sh

```
cd /tmp
git clone https://aur.archlinux.org/qemu-arm-static.git
cd qemu-arm-static
yes | makepkg -si
```

Debido a que *qemu-arm-static* solo está disponible en el AUR, hemos de descargarlo e instalarlo utilizando *makepkg*.

5.2 Conexión con la Raspberry Pi

Una vez hemos ejecutado los *scripts* anteriores, ya tenemos Arch Linux ARM instalado en nuestra tarjeta SD. Insertamos la tarjeta en la Raspberry Pi, la conectamos a través de Ethernet a nuestro ordenador personal y la conectamos al alimentador para suministrarle energía y que funcione. La Raspberry Pi arrancará, pero no podremos interactuar con ella de momento.

La Raspberry Pi tiene configurada una IP fija, pero las tablas de *enrutamiento* de nuestro ordenador personal son las mismas que antes, es decir, no se actualizan solas. Así pues, hemos de crear un *script* capaz de configurar adecuadamente nuestro ordenador personal para que se comunique con la Raspberry Pi. Supondremos que nuestro personal ordenador está conectado a una red Wifi, por lo que no perderemos la conexión a Internet, y además, el *script* debe ser capaz de hacer que nuestro ordenador personal actúe como

La técnica que utilizaremos para evitar problemas será borrar completamente la configuración que pudiera haber y crearla de nuevo. Si no hiciéramos esto, las antiguas entradas que pudiera haber en la tabla de *enrutamiento* interferirían con la nueva configuración.

5.2.1. Implementación

Para implementar este *script* se ha utilizado el artículo [4].

setupNetwork.sh

```
ip address flush dev $ethernet_dev
```

Borramos cualquier dirección IP que pueda haber sido asignada a la interfaz Ethernet.

setupNetwork.sh

```
ip link set $ethernet_dev up
```

Nos aseguramos de que la interfaz Ethernet esté activo.

setupNetwork.sh

```
ip route flush table 254
```

Borramos la tabla de *enrutamiento* principal (la número 254).

setupNetwork.sh

```
ip address add $ethernet_address_and_mask broadcast + dev $ethernet_dev
```

Asigmanos la dirección IP elegida a la interfaz Ethernet.

setupNetwork.sh

```
route add -host $raspberry_pi_address metric 1 dev $ethernet_dev
route add -net $wifi_lan_address_and_mask metric 10 dev $wifi_dev
route add default gw $wifi_gateway_address metric 20
```

Actualizamos la tabla de *enrutamiento* principal. Configuramos tres entradas:

- Si la IP es la de la Raspberry Pi, enviar por la interfaz Ethernet.
- Si la IP es del área local del Wifi, enviar por la interfaz Wifi.
- Si es cualquier otra dirección, enviar a la puerta de enlace de la red Wifi.

Con estas reglas conseguimos poder comunicarnos con la Raspberry Pi sin perder la conexión a Internet.

setupNetwork.sh

```
sysctl net.ipv4.ip_forward=1 > /dev/null
iptables -t nat -A POSTROUTING -o $wifi_dev -j MASQUERADE
iptables -A FORWARD -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
iptables -A FORWARD -i $ethernet_dev -o $wifi_dev -j ACCEPT
```

Configuramos `iptables` para que nuestro ordenador personal haga de *router* de la Raspberry Pi.

5.2.2. Utilización del *script*

Para poder conectarnos a la Raspberry Pi, simplemente ejecutamos `setupNetwork.sh`. Si ejecutamos `ping <IP Raspberry Pi>`, veremos que tenemos conexión.

Además, si hacemos *ping* desde la Raspberry Pi a otro ordenador, y en el otro ordenador observamos el tráfico ICMP con *tcpdump* o *wireshark*, podremos ver que la dirección desde la cual se reciben los *ping* es desde la del ordenador personal, lo cual prueba que el *forwarding* IP funciona.

5.2.3. Configuración de SSH

Para poder conectarnos a la Raspberry Pi por SSH, necesitamos la clave privada generada anteriormente. La clave está disponible en el directorio donde hemos ejecutado los *scripts* de instalación y configuración. Es conveniente que la copiemos al directorio `.ssh` y nos aseguremos que tiene los permisos necesarios (ha de pertenecer al usuario que ejecuta `ssh` y solamente él ha de poder leer y escribir en el fichero).

A continuación, ejecutamos el comando `ssh admin@<IPraspberryPi>-p <puerto>-i </ruta/a/la/clave/privada>` para conectarnos a la Raspberry Pi. Veremos como aparece el mensaje de bienvenida y tenemos acceso a `bash`.

Sin embargo, tener que escribir cada vez este comando es incómodo, por lo que escribimos una entrada en el fichero `.ssh/config`. Quedaría de la siguiente manera:

.ssh/config

```
1 Host raspi
2     HostName <IPraspberryPi>
3     User admin
4     Port <puerto>
5     IdentityFile </ruta/a/la/clave/privada>
```

De esta forma, podemos escribir `ssh raspi`, y `ssh` se encargará de utilizar la información de `.ssh/config`. Además, al utilizar este archivo de configuración obtenemos la ventaja de que todos los comandos de `openssh` lo utilizarán. Otra ventaja es que programas como `rsync` también son capaces de utilizar esta configuración, por lo que su utilización será más cómoda.

CAPÍTULO 6
Servicios

Ahora que ya tenemos los aspectos básicos de la Raspberry Pi configurados y podemos comunicarnos con ella fácilmente, podemos empezar a configurar servicios concretos.

6.1 FTP

FTP (*File Transfer Protocol)* es un protocolo utilizado para el intercambio de archivos que utiliza el modelo cliente-servidor. Solamente tiene sentido utilizarlo cuando el cliente y el servidor están en computadoras distintas, permitiendo el intercambio de archivos a través de una red. La primera versión del protocolo que estaba pensado para funcionar sobre una red TCP/IP está descrita en la RFC765 (*Request For Comments*), escrita en el 1980.

No fue diseñado pensando en la seguridad (por ejemplo, las contraseñas de autenticación no se encriptan en ningún momento), lo cual ha provocado que haya alternativas que sí implementan medidas de seguridad, como por ejemplo SFTP (SSH *File Transfer Protocol*). Sin embargo, FTP tiene la ventaja de ser más simple, lo que permite que incluso podamos utilizar FTP desde nuestro navegador, ya que los navegadores más utilizados, tales como Google Chrome o Mozilla Firefox implementan un cliente FTP.

6.1.1. `bftpd`

`bftpd` es un servidor FTP fácil de configurar y disponible en los repositorios de Arch Linux ARM. La documentación principal de este programa está en el archivo de configuración mismo, localizado en `/etc/bftpd.conf`, donde explica que hace cada opción.

Configuración sin autenticación

Si queremos desplegar un servidor FTP donde cualquier persona pueda descargar contenido, los cambios que hemos de hacer en la configuración son mínimos. En el archivo de configuración vemos que la opción `DENY_LOGIN` activada, por lo que no nos dejará conectarnos sin realizar autenticación.

bftpd.conf

```
user ftp {
  #Any password fits.
  ANONYMOUS_USER="yes"
  DENY_LOGIN="Anonymous login disabled."
```

```
5   # bftpd interprets ROOTDIR="%h" (the default), as ROOTDIR="/" for the
        anonymous user, override it
6   ROOTDIR="/srv/ftp"
7 }
```

Para que nos deje conectarnos sin utilizar ningún tipo de credenciales comentamos esa opción poniendo un # al principio de la línea.

La opción `ROOTDIR` es el directorio raíz al que podemos acceder, y podemos poner cualquier directorio. Para realizar las pruebas dejamos la opción por defecto `/srv/ftp`.

Para activar el servicio una vez configurado utilizamos el comando `systemctl start bftpd`. Una vez activado, debemos poner algún archivo para intentar descargarlo y ver si todo funciona correctamente. Podemos poner cualquier archivo, así que con utilizar el comando `echo "Prueba bftpd» /srv/ftp/prueba.txt` podemos crear uno.

Podemos automatizar el proceso con el siguiente *script*:

ftpNoUser.sh

```
#!/bin/bash
yes | pacman -S bftpd

file=/etc/bftpd.conf
declare -r tempdir=/tmp/ftpNoUser
mkdir $tempdir

sed 's/DENY_LOGIN="Anonymous login disabled."/#&/' $file > ${tempdir}/
    output
mv ${tempdir}/output $file

rm -r $tempdir

service=bftpd
systemctl start $service
if ! systemctl is-active --quiet $service ; then
    systemctl start $service
else
    systemctl restart $service
fi
```

Para comprobar que tenemos acceso al servicio podemos utilizar el comando (`curl ftp://ipDeLaRaspberryPi/prueba.txt` y veremos que se muestra por pantalla el contenido del archivo que habíamos creado anteriormente, lo que demuestra que todo funciona correctamente.

También podemos utilizar Chrome, ya que cuenta con un cliente FTP. Así pues, en la barra de direcciones escribimos la URL `ftp://<ipDeLaRaspberryPi>` y veremos que aparece una lista de archivos y directorios (en este caso solamente el fichero "prueba.txt" que hemos puesto anteriormente. Si hacemos `click` el contenido del fichero aparecerá en pantalla.

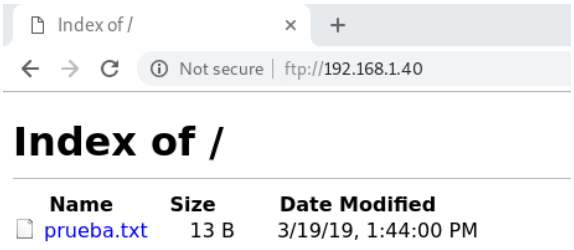

Figura 6.1: Cliente FTP de Chrome

Configuración con autenticación

También podemos configurar `bftpd` para que nos pida un usuario y contraseña para poder acceder al servicio. Podemos automatizar la configuración con el siguiente *script*:

ftpUser.sh

```
#!/bin/bash
set -u
set -e
user=$1
if ! $(id $user &>/dev/null); then
    useradd -s /sbin/nologin -m $user

cat >> /etc/bftpd.conf << EOF
user $user{
}
EOF
fi

passwd $user

service=bftpd
systemctl start $service
if ! systemctl is-active --quiet $service ; then
    systemctl start $service
else
    systemctl restart $service
fi
```

`bftpd` puede utilizar las credenciales de los usuarios de Unix para autenticarse en el servidor FTP. Así pues, para crear un usuario que no pueda acceder a ninguna `shell` y que se cree automáticamente su directorio de usuario en `/home` utilizamos el comando `useradd -s /sbin/nologin -m`. A continuación ponemos una contraseña al usuario, la cual utilizará también `bftpd`.

En `bftpd.conf` hemos de indicar que queremos utilizar las credenciales del usuario para acceder al servidor FTP. Podemos añadiendo a la configuración por defecto con el contenido del *here document*. Si no indicamos ninguna opción, el directorio raíz del servidor FTP para ese usuario será su directorio de usuario, y la contraseña será la que hayamos configurado con `passwd`.

Para realizar una prueba, ponemos un archivo en el directorio del usuario con `echo "Prueba ftp para el usuario <prueba>» /home/<usuario>/prueba` y utilizamos el comando `curl ftp://<ipDeLaRaspberryPi>/archivo -u <usuario>`. Veremos que sale el contenido del archivo por pantalla.

También podemos utilizar un cliente FTP con una interfaz gráfica, como por ejemplo Filezilla. Campo correspondientes escribimos la IP de la Raspberry Pi, el nombre de usuario y la contraseña. Vemos que el fichero que hemos creado aparece en el servidor.

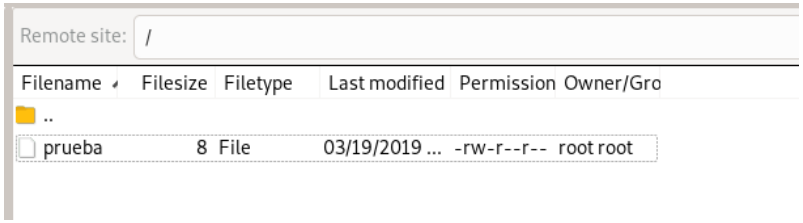

Figura 6.2: Filezilla

6.2 SFTP

SFTP es una extensión de SSH que permite el intercambio de archivos a través de una red de forma segura. No es solamente un protocolo para intercambio de archivos de forma segura, sino permite además montar en nuestro sistema de ficheros el contenido del servidor.

Este servicio es más complicado de configurar que `bftpd`, pero por suerte tenemos una guía en la Arch Wiki [5]. Para automatizar el proceso sugerido, creamos el siguiente *script*:

sftpJailroot.sh

```
#!/bin/bash
set -u
set -e
key=$1
mkdir -p /mnt/data/share
mkdir -p /srv/ssh/jail
echo "hola"
mount -o bind /mnt/data/share /srv/ssh/jail
echo "/mnt/data/share /srv/ssh/jail  none   bind   0   0" >> /etc/fstab
groupadd sftponly
useradd -g sftponly -s /usr/bin/nologin -d /srv/ssh/jail sftponly

cat >> /etc/ssh/sshd_config <<EOF
Subsystem sftp /usr/lib/ssh/sftp-server

Match Group sftponly
  ChrootDirectory %h
  ForceCommand internal-sftp
  AllowTcpForwarding no
  X11Forwarding no
  PasswordAuthentication no
EOF

mkdir /etc/ssh/authorized_keys
cat $key >> /etc/ssh/authorized_keys/sftponly
chmod 644 /etc/ssh/authorized_keys/sftponly
```

Este `script` se encarga de crear una `chroot jail` en el directorio `/srv/ssh/jail`. Cuando un usuario ejecute `sftp`, solamente tendrá acceso a los archivos por debajo de este directorio.

Nótese que para añadir usuarios al servidor SFTP no creamos usuarios nuevos, sino que los usuarios deben darnos su clave pública y la hemos de añadir a `/etc/ssh/authorized_keys/sftponly`. De esta forma conseguimos más seguridad que en un servidor FTP normal, ya que para que un *cracker* accediera a nuestro servidor SFTP, tendría que robarle la clave pública a un usuario y después tendría que adivinar su *passphrase*. En un servidor FTP normal, simplemente tendría que capturar su tráfico y vería la contraseña en texto plano, lo cual es mucho más sencillo.

Para hacer más cómodo el uso del servidor SFTP, podemos crear una entrada en `.ssh/config` tal y como hicimos en la configuración de SSH. Una vez creado, tan solo debemos ejecutar `sftp <alias>` para tener acceso al servidor.

6.3 Git

Git es un sistema de control de versiones lanzado en el 2005. Fue diseñado por Linus Torvalds para ser utilizado como sistema de control de versiones de Linux. Debido a esto, su uso se ha extendido notablemente desde su lanzamiento, hasta convertirse casi en un estándar hoy en día. Su diseño hace que sea un sistema distribuido, donde no hay un repositorio central sino que se realiza una copia del repositorio en la computadora de cada persona que trabaja en el proyecto. Sin embargo, pese a ser distribuido, es útil tener una copia del repositorio en un servidor desde la cual sepamos que siempre podemos descargar el repositorio.

Git soporta varios modos de acceso a repositorios remotos. Tiene un protocolo propio llamado git, puede ir sobre HTTPS o puede ir sobre SSH. En nuestro caso vamos a git sobre SSH para acceder a el repositorio que pondremos en el servidor. Con el siguiente *script* se puede configurar rápidamente un repositorio.

git.sh

```
#!/bin/bash
set -e
set -u
key=$(realpath $1)
repo=$2
if  [[ -z $(grep "^git" /etc/passwd) ]]; then
    useradd -s /usr/bin/git-shell git
fi

cat $key >> /etc/ssh/authorized_keys/git
mkdir -p /srv/git/$repo
cd /srv/git/$repo
git init --bare
cd ..
chown -R git:git $repo
```

Lo que hacemos es crear un usuario "git" que solo sirva para acceder git. Esto se consigue haciendo que la *shell* del usuario sea git-shell. En realidad no es una *shell* estrictamente hablando, sino que es una *login shell* que permite utilizar solamente los comandos de *git* en la parte de servidor.

En el fichero que tiene las claves públicas (para permitir el acceso a quien tenga la clave privada) en este caso ponemos una clave, pero podríamos poner varias, de forma que diferentes personas pudieran acceder al repositorio. Posteriormente inicializamos el repositorio en modo -bare, lo cual hace que el repositorio no tenga un directorio de trabajo, que es la mejor opción cuando solo vamos a hacer push y pull al repositorio y no vamos a trabajar directamente en él. Por último, nos aseguramos que el repositorio pertenece al usuario "git".

Desde un ordenador donde tengamos la clave privada, añadimos al fichero .ssh/config las siguientes líneas para crear un alias ssh:

.ssh/config

```
Host git
    HostName <IPraspberryPi>
    User git
    Port <puerto>
    IdentityFile </ruta/a/la/clave/privada>
```

De esta forma, podremos utilizar el comando git clone git:/srv/git/nombreRepositorio para descargar una copia del repositorio. Como

el usuario "git" tiene permisos de lectura y escritura en el repositorio, si hacemos un *commit* y hacemos `push` al servidor, subiremos la actualización del repositorio al servidor.

6.4 RAID

RAID es una tecnología que consiste en juntar diferentes dispositivos de almacenamiento y tratarlos como si fuera un único disco. Se puede implementar por *hardware* o por *software*, siendo el RAID por *hardware* una opción cara (ya que requiere la compra de una controladora de RAID) y más enfocada a servidores. Así pues, en nuestro caso lo haremos por *software* utilizando el programa *mdadm*.

Existen varios niveles de RAID, y cada uno indica una configuración distinta. Por ejemplo, RAID 0 combina los discos de forma que guardan distintas partes de la información en cada disco, lo que hace que la información se pueda leer y escribir más rápido en caso que el cuello de botella sea el ancho de banda de transmisión de datos de los discos. Sin embargo su desventaja es que si un de los discos se rompe, toda la información se pierde.

RAID 1 combina los discos de forma que los discos guardan la misma información de forma redundante, de forma que si uno falla, la información sigue estando en el otro. Una vez se retira el disco que ha fallado, se puede poner un disco nuevo y automáticamente la información se copiara en él. La desventaja es que perdemos capacidad de almacenamiento, ya que estamos guardando la misma información dos veces.

Las otras configuraciones son una mezcla de estas dos, lo que permite tener redundancia y beneficio de velocidad de lectura/escritura al mismo tiempo (aunque menor que en los casos anteriores) y tener una menor penalización de perdida de almacenamiento, lo cual es deseable.

6.4.1. Programas utilizados

Utilizamos el siguiente *script* para escribir la tabla de particiones y formatear un USB para ser utilizado de forma individual. El sistema de ficheros elegido es `ext4`.

partitionAndFormatExt4.sh

```
#!/bin/bash
set -u
set -e
target_device=${1}

mdadm --misc --zero-superblock ${target_device}

# Do not put blank lines until EOF (unless you know what you are doing), because they will be interpreted as a <CR> inside fdisk
sed -e 's/\s*\([\+0-9a-zA-Z]*\).*/\1/' << EOF | fdisk -W always ${target_device}
  o # clear the in memory partition table
  n # new partition
  p # primary partition
    # default - partition 1
    # default - start at beginning of disk
    # default - until the end of the disk
  w # write the partition table
  q # and we're done
EOF

partprobe
partition=${target_device}1
```

```
yes | mkfs.ext4 ${partition}
echo "DONE"
```

Utilizamos el siguiente *script* para escribir la tabla de particiones para un USB que va a ser utilizado en RAID. No tiene sentido formatearlo, ya que lo que hay que formatear para utilizarlo es el dispositivo MD (el que da acceso al RAID) que crearemos después.

prepareForRaid.sh

```
#!/bin/bash
set -u
set -e
target_device=${1}

mdadm --misc --zero-superblock ${target_device}

# Do not put blank lines until EOF (unless you know what you are doing), because they will be interpreted as a <CR> inside fdisk
sed -e 's/\s*\([\+0-9a-zA-Z]*\).*/\1/' << EOF | fdisk -W always ${target_device}
  o # clear the in memory partition table
  n # new partition
  p # primary partition
    # default - partition 1
    # default - start at beginning of disk
    # default - until the end of the disk
  t # change type
  fd # type: Linux raid autodetect
  w # write the partition table
EOF
#partprobe
echo "DONE"
```

El siguiente script crea y formatea un RAID 0 de 4 dispositivos.

raid0_4drives.sh

```
#!/bin/bash
drive1=$1
drive2=$2
drive3=$3
drive4=$4

mdadm --create --verbose --level=0 --metadata=1.2 --raid-devices=4 /dev/md/raid0_4drives ${drive1} ${drive2} ${drive3} ${drive4}
mdadm --detail --scan >> /etc/mdadm.conf
mdadm --assemble --scan
mkfs.ext4 -v -L raid0_4drives -m 0.5 -b 4096 -E stride=128,stripe-width=512 /dev/md/raid0_4drives
#128: 512/4
#512: 4*128
echo "DONE"
```

El siguiente script crea y formatea un RAID 1 de 2 dispositivos.

raid1_2drives.sh

```
#!/bin/bash
drive1=$1
drive2=$2

mdadm --create --verbose --level=1 --metadata=1.2 --raid-devices=2 /dev/md/raid1_2drives ${drive1} ${drive2}
mdadm --detail --scan >> /etc/mdadm.conf
mdadm --assemble --scan
mkfs.ext4 /dev/md/raid1_2drives
```

```
9 echo "DONE"
```

6.4.2. RAID 0

En este apartado vamos a configurar RAID 0 y comprobar si obtenemos mejoras en la velocidad de lectura/escritura. Primeramente realizaremos una serie de pruebas con un único *pen drive* y mediremos distintos parámetros y después realizaremos las mismas pruebas con una configuración de RAID 0 para comprobar las mejorías obtenidas.

Para realizar las pruebas primero utilizamos el comando `dd` para realizar escrituras y lecturas. En estas pruebas medimos la velocidad de lectura y escritura de dos formas distintas: leyendo/escribiendo directamente en el *block device* y leyendo/escribiendo en el sistema de ficheros. Además, realizaremos estas pruebas realizando la escritura de un fichero grande y la escritura de muchos ficheros pequeños, ya que existen diferencias importantes . Realizaremos tres mediciones para evitar valores atípicos.

Utilizamos el *flag* `direct`, para que la escritura en disco se haga sin pasar por la *caché* de Linux. De esta forma las medidas obtenidas son más fiables, ya que nos aseguramos de que los datos se escriben en disco.

Pruebas con un único *pen drive*

Los datos obtenidos utilizando `dd` son los siguientes:

Un fichero grande raw				
dd if=/dev/zero of=/dev/sda1 bs=200M count=1 oflag=direct	Escritura	10.5MB/s	10.5MB/s	10.5MB/s
dd if=/dev/sda1 of=/dev/null bs=200M count=1 iflag=direct	Lectura	35.3MB/s	35.3MB/s	35.3MB/s
Un fichero grande sistema de ficheros				
dd if=/dev/zero of=/mnt/1USB/prueba bs=200M count=1 oflag=direct	Escritura	10.1MB/s	10.0MB/s	10.2MB/s
dd if=/mnt/1USB/prueba of=/dev/null bs=200M count=1 iflag=direct	Lectura	34.7 MB/s	34.7 MB/s	34.5 MB/s
Múltiples ficheros pequeños raw				
dd if=/dev/zero of=/dev/sda1 bs=512 count=10000 oflag=direct	Escritura	1.1MB/s	1.0MB/s	1.1MB/s
dd if=/dev/sda1 of=/dev/null bs=512 count=10000 iflag=direct	Lectura	1.5MB/s	1.4MB/s	1.4MB/s
Múltiples ficheros pequeños sistema de ficheros				
dd if=/dev/zero of=/mnt/1USB/prueba bs=512 count=10000 oflag=direct	Escritura	378 kB/s	436 kB/s	428 kB/s
dd if=/mnt/1USB/prueba of=/dev/null bs=512 count=10000 iflag=direct	Lectura	977 kB/s	964 kB/s	965 kB/s

Figura 6.3: Resultados `dd`

Vemos que en todos los casos la escritura directa en el *pen drive* es más lenta que la lectura. Podemos comprobar que utilizar un sistema de ficheros hace que las lecturas y escrituras sean más lentas, sobretodo en el caso de escribir muchos ficheros pequeños.

También vemos que la escritura de un solo archivo grande es más rápida que el de muchos archivos pequeños.

Pruebas con RAID 0 con 4 *pen drives*

Los datos obtenidos utilizando `dd` son los siguientes:

Un fichero grande raw				
dd if=/dev/zero of=/dev/md127 bs=200M count=1 oflag=direct	Escritura	31.3 MB/s	31.5 MB/s	31.2 MB/s
dd if=/dev/md127 of=/dev/null bs=200M count=1 iflag=direct	Lectura	44.7 MB/s	44.8 MB/s	44.7 MB/s
Un fichero grande sistema de ficheros				
dd if=/dev/zero of=/mnt/raid0/prueba bs=200M count=1 oflag=direct	Escritura	28.1 MB/s	26.4 MB/s	27.1 MB/s
dd if=/mnt/raid0/prueba of=/dev/null bs=200M count=1 iflag=direct	Lectura	44.6 MB/s	44.4 MB/s	44.6 MB/s
Múltiples ficheros pequeños raw				
dd if=/dev/zero of=/dev/md127 bs=512 count=10000 oflag=direct	Escritura	1.1 MB/s	1.0 MB/s	1.0 MB/s
dd if=/dev/md127 of=/dev/null bs=512 count=10000 iflag=direct	Lectura	1.5MB/s	1.5MB/s	1.5MB/s
Múltiples ficheros pequeños sistema de ficheros				
dd if=/dev/zero of=/mnt/raid0/prueba bs=512 count=10000 oflag=direct	Escritura	416 kB/s	369 kB/s	465 kB/s
dd if=/mnt/raid0/prueba of=/dev/null bs=512 count=10000 iflag=direct	Lectura	1.3 MB/s	1.1 MB/s	1.2 MB/s

Figura 6.4: Resultados `dd`

Puede observarse que al escribir un archivo grande, la velocidad de lectura/escritura ha aumentado considerablemente. Sin embargo, en el caso de leer o escribir muchos archivos pequeños la mejora ha sido mínima o nula. Esto se debe probablemente a que en esta tarea el cuello de botella no es el ancho de banda de lectura/escritura en disco, sino la latencia al que existe en cada operación de lectura/escritura. También podría ser que el procesador de la Raspberry Pi no pueda procesar estas operaciones más rápido.

6.4.3. RAID 1

Ahora vamos a configurar RAID 1 con 2 discos, y comprobaremos que si quitamos uno (simulando un fallo) y ponemos un disco nuevo, la información se copia al nuevo automáticamente.

Una vez ejecutado el *script* que crea el RAID 1, podemos montar el disco en nuestro sistema de ficheros y empezar a añadir información como si fuera un solo disco.

Si ejecutamos el comando `cat /proc/mdstat` podremos ver información sobre el estado del los dispositivos RAID. Si acabamos de configurarlo, veremos una barra indicando un porcentaje que va creciendo, lo cual indica el proceso de clonación de un disco en el otro.

Cuando el proceso se ha completado utilizamos los comandos `mdadm /dev/mdx -r /dev/sdyy` y `mdadm -remove /dev/mdx /dev/sdyy` para marcar uno de los dos discos como fallido y lo eliminamos del RAID. Una vez ejecutados los comandos, retiramos el *pen drive* del puerto USB. Si intentamos acceder a los archivos vemos que todo funciona sin ningún problema.

Para añadir un nuevo disco al RAID utilizamos el comando `mdadm -add /dev/mdx /dev/sdzz`. Si ejecutamos `cat /proc/mdstat` veremos otra vez la barra que indica que los datos del disco que ya estaba en el RAID se están copiando en el nuevo disco.

6.5 Node-RED

`Node-RED` es un herramienta de programación gráfica que nos permite crear aplicaciones juntando bloques. Esta orientado al diseño rápido y prototipado de programas, siendo especialmente útil para aplicaciones relacionadas con *IoT*. Está basado en la plataforma `Node.js`, por lo que para realizar aplicaciones complejas debemos saber `javascript`, ya que aparte de los bloques que vamos juntando, también podemos escribir código en algunos de ellos.

`Node-RED` está preparado para interactuar con los *pines* GPIO (*General Purpose Input Output*), no obstante, necesita utilizar Python y la librería `RPi.GPIO` para hacerlo. Podemos instalar esta librería fácilmente, ya que está disponible en el AUR.

El *script* que utilizamos para instalar Node-RED junto con RPi.GPIO es el siguiente:

6.6 Node-RED

installNodeRed.sh

```
#!/bin/bash

yes | pacman -S fakeroot npm gcc
su admin<<EOF
cd /tmp
git clone https://aur.archlinux.org/python-raspberry-gpio.git
cd python-raspberry-gpio
yes | makepkg -si
EOF

#https://glasstty.com/wiki/index.php/Installing_Node_Red
npm install -g --unsafe-perm node-red
touch /usr/share/doc/python-rpi.gpio
```

Para instalar un programa del AUR se utiliza git, y para instalar el programa se utiliza makepkg. Nótese que hemos de instalar las dependencias que va a utilizar makepkg, ya que sino fallará la instalación. En este caso las dependencias son fakeroot y gcc. Es importante ejecutar makepkg con un usuario diferente a "root", debido a razones de seguridad (los programas del AUR no son oficiales y pueden contener malware).

Al instalar las dependencias del instalador de RPi.GPIO también instalamos npm, el gestor de paquetes de Node.js, desde el cual instalamos Node-RED.

Por último, debemos solucionar un *bug* que hay en la Node.js cuando lo instalamos en Arch Linux ARM. En https://github.com/node-red/node-red/issues/945 solucionaron el *bug* para Python 2.7, pero ahora que vamos por Python 3.7 el la solución no funciona. El *bug* es ocasionado porqué en el archivo /usr/lib/node_modules/node-red/nodes/core/hardware/36-rpi-gpio.js Node-RED utiliza la instrucción fs.statSync(/usr/lib/python2.7/site-packages/RPi/GPIO"); para comprobar que RPi.GPIO está instalado en nuestro sistema. En la Arch Linux ARM ese directorio no está, por tanto Node-RED detecta un error y dice que RPi.GPIO no está instalado.

Es mejor no crear a mano el directorio y hacer que esta comprobación pase, ya se encuentra en /usr/lib/ y es mejor no tocar nada por si en el futuro instalamos Python 2.7 y provoca algún tipo de error que ese directorio esté creado. La comprobación de Raspbian la hace de la siguiente forma: fs.statSync(/usr/share/doc/python-rpi.gpio");, así que si creamos el directorio python-rpi.gpio, pasará el test y todo funcionará correctamente. Podemos hacer esto porque Node-RED realiza todas las comprobaciones una detrás de otra sin comprobar qué distribución tenemos antes de realizar la comprobación de si existe o no el archivo.

6.6.1. Pruebas

Para arrancar Node-RED utilizamos el comando node-red-pi y esperamos a que arranque. Después abrimos un navegador y en la barra de direcciones escribimos direccionRaspberryPi:1880 y nos saldrá un página desde donde podemos elaborar los programas.

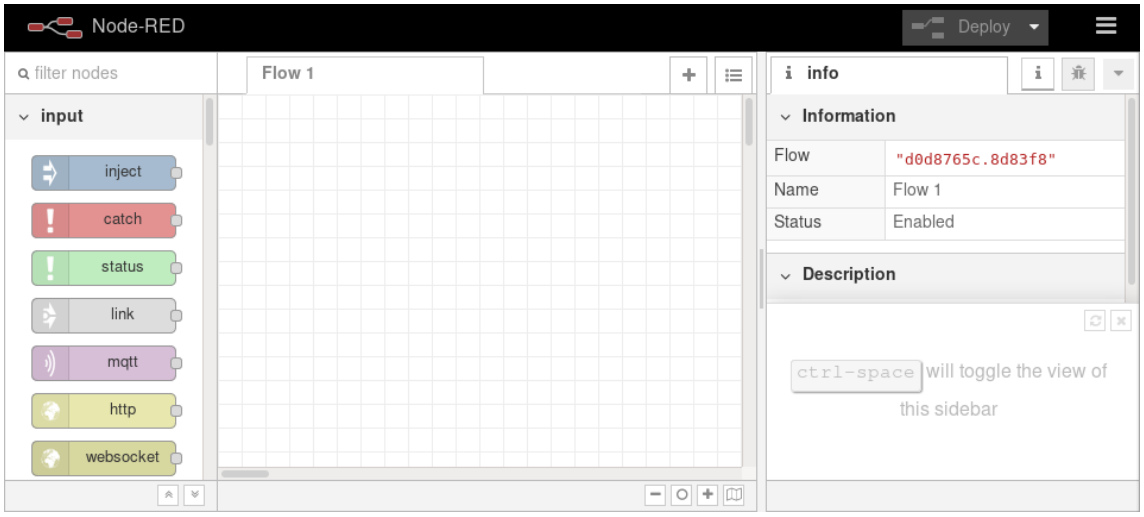

Figura 6.5: Node-red

Web básica

Uno de los programas más sencillos que podemos crear es hacer una web donde salga un texto por pantalla. Para ello utilizamos los bloques `http`, `template` y `http response` de la siguiente forma:

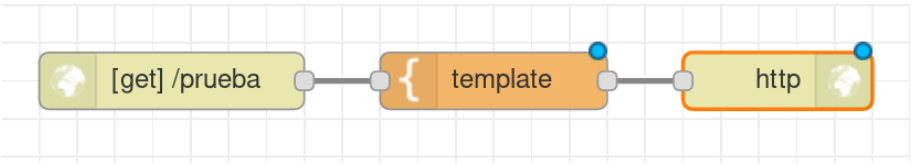

Figura 6.6: Servidor web simple

En la configuración del bloque `http` ponemos la URL donde queremos que esté nuestra página web. En el bloque `template` escribimos una página web lo más simple posible con HTML, como por ejemplo:

Contenido de template

```
<!DOCTYPE html>
<html>
<head>
<title>Prueba Node-Red</title>
</head>
<body>

<h1>Prueba Node-Red</h1>
<p>Prueba Node-Red</p>

</body>
</html>
```

Por último hacemos *click* en `Deploy` para desplegar la aplicación. Si vamos a la URL que elegida anteriormente deberíamos ver la web creada.

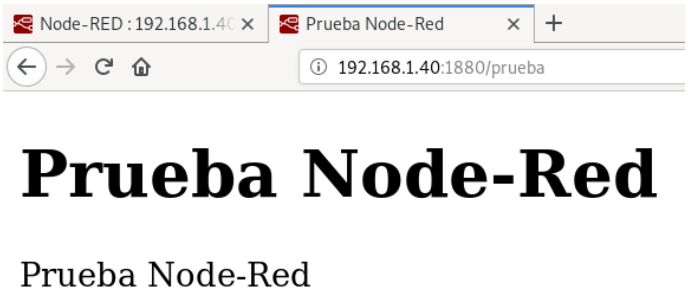

Figura 6.7: Web simple

Uso de *pin* GPIO

Desde `Node-RED` podemos utilizar muy fácilmente los *pines* GPIO. Utilizamos bloque `rpi gpio` para leer el valor de un *pin*. En la configuración podemos elegir poner una resistencia de *pullup*, *pulldown* o no poner nada, así como utilizar un filtro que evite rebotes eléctricos donde podemos configurar el número de milisegundos que la Raspberry Pi no leerá el *pin* después de haber detectado el primer cambio (actuando como un filtro paso bajo).

Para observar la salida podemos utilizar el bloque `debug`, el cual nos muestra la salida de un bloque en el panel de *debug*.

La configuración empleada es la siguiente:

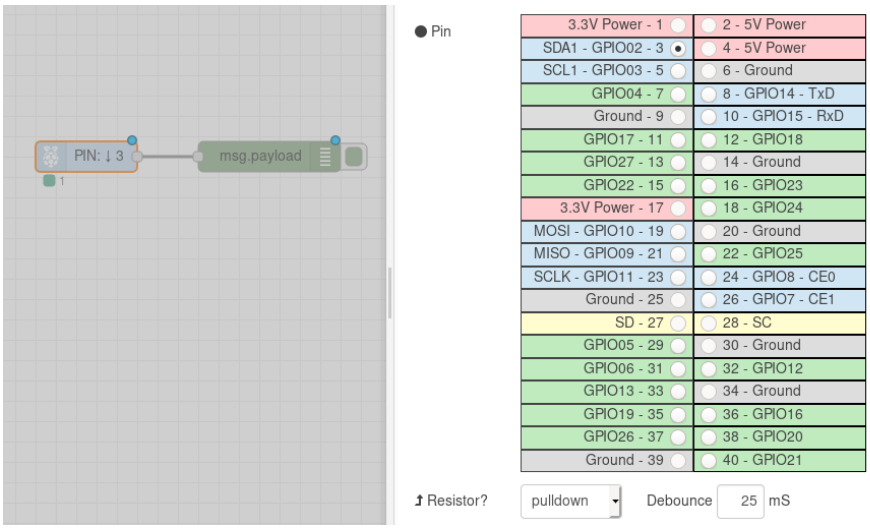

Figura 6.8: Configuración GPIO

Elegimos utilizar el *pin* 3, ya que tiene un *pin* de $5V$ al lado y podemos conectarlos fácilmente. Configuramos una resistencia de *pulldown*, para que cuando el pin no esté conectado a nada su estado sea "0". Una vez configurado hacemos *click* en `Deploy` y conectamos y desconectamos el *pin* 3 y el de $5V$, y veremos como en la ventana de *debug* se aprecian los cambios en el estado del "pin".

Módulo `node-red-node-pi-sense-hat-simulator`

El Sense HAT es una placa de extensión de la Raspberry Pi orientada a IoT. Esta placa se comunica con la Raspberry Pi a través de los *pines* GPIO, y cuenta con sensores de humedad, presión atmosférica, temperatura, giroscopio, acelerómetro, y magnetómetro, además de un *joystick*, cinco botones y una matriz de *LED* 8x8.

`Node-RED` cuenta con un simulador del Sense HAT llamado `node-red-node-pi-sense-hat-simulator`. Para instalarlo podemos hacerlo desde la interfaz de *Node-red*. Vamos a la opción *Manage palette*, buscamos el nombre del módulo y lo instalamos.

La utilización del simulador es muy simple. Simplemente conectamos los módulos de la siguiente forma y hacemos click en *Deploy*.

Figura 6.9: Simulador Sense HAT

Una vez desplegado el servicio, vamos a la URL `http://<IPRaspberryPi>:1880/sensehat-simulator` y tendremos acceso al simulador:

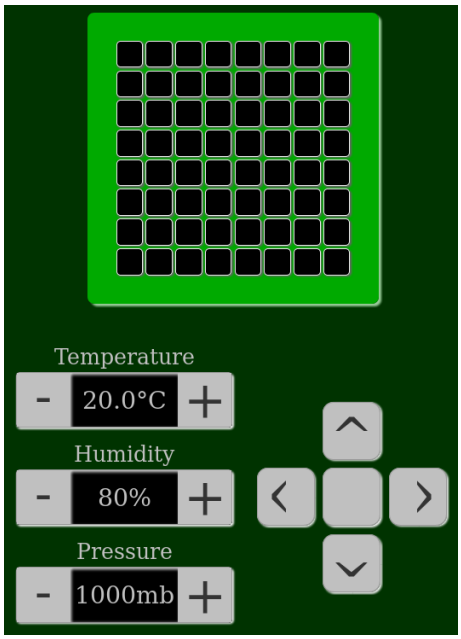

Figura 6.10: Simulador Sense HAT

Módulo `node-red-dashboard`

En este apartado vamos a implementar una página web que muestre la información de los sensores del simulador del Sense HAT. Para ello utilizamos el módulo `node-red-dashboard`, el cual permite generar distintos tipos de gráficas, así como poner botones y *sliders* para hacer paneles de control.

Para instalar el módulo utilizamos el comando `npm i node-red-dashboard`. Una vez ejecutado, realizamos el mismo proceso que con el módulo anterior.

Una vez instalado, podemos empezar a implementar la web. El simulador del Sense HAT cuenta con tres sensores distintos: temperatura, humedad y presión. Lo que hacemos es crear tres gráficas (una para cada magnitud), por lo que nuestro *flow chart* tiene tres "caminos". El *flow chart* global es el siguiente:

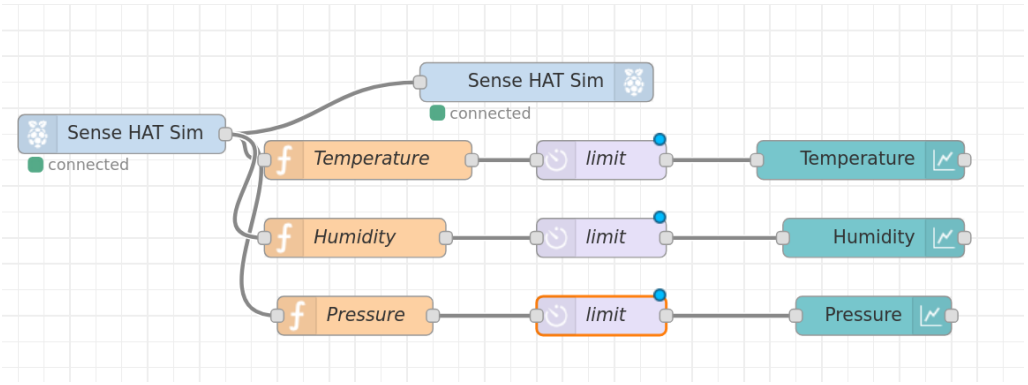

Figura 6.11: *Flow chart* global

Los bloques de color naranja son funciones. En este caso, las utilizamos para extraer la información que queremos de los mensajes que genera el bloque del Sense HAT. Su contenido es el siguiente:

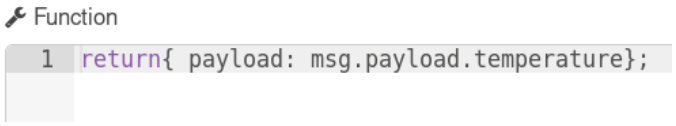

Figura 6.12: Contenido función

Los bloques violeta se utilizan para introducir retardos o limitar el número de mensajes que llegan a un bloque. En nuestro caso, hemos de utilizarlos para limitar los mensajes que llegan a los bloques que general las gráficas, ya que si conectamos la función anterior directamente con el bloque, llegarán más mensajes de los que el navegador puede procesar al *renderizar* las gráficas. Configurándolo de la siguiente manera conseguimos que entre 1 mensaje cada 0.1 segundos, y que todos los que se envíen en medio se eliminen.

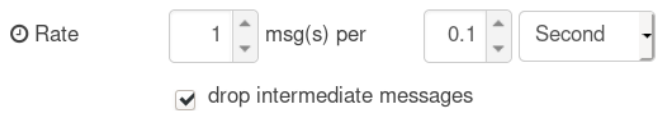

Figura 6.13: Configuración bloque *limit*

Por último, para que las gráficas estén una al lado de otra, creamos el siguiente *layout*:

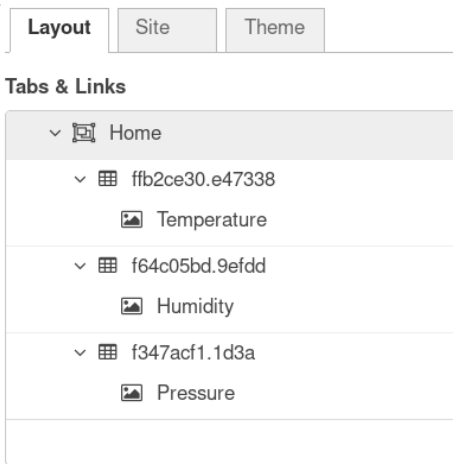

Figura 6.14: *Layout*

Para ver el resultado accedemos a la URL `http://<IPRaspberryPi>:1880/ui`. El resultado es el siguiente:

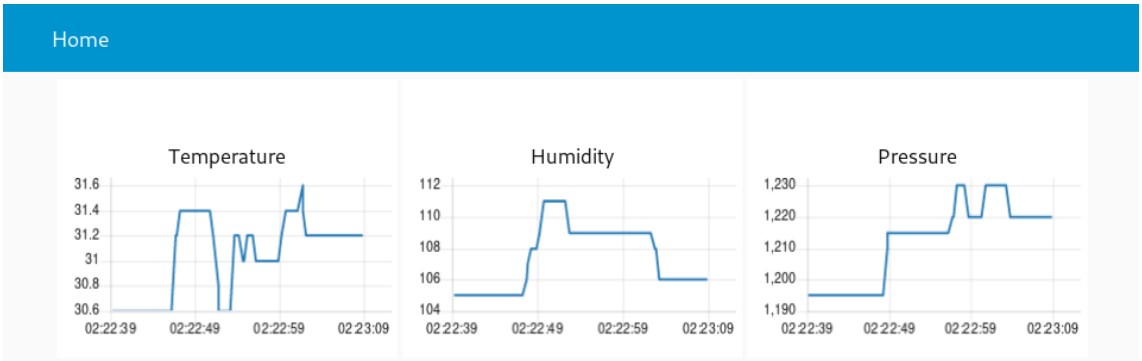

Figura 6.15: Graficas sensores

Si vamos modificando el valor de los sensores en el Sense HAT, veremos como el valor de las gráficas va variando.

6.7 Interfaz gráfica en Python

En este apartado vamos a implementar una sencilla interfaz gráfica que nos permitirá apagar y encender diferentes servicios en la Raspberry Pi. La implementaremos en `python`, concretamente utilizando `kivy`, un *framework* para el desarrollo rápido de aplicaciones gráficas. *kivy* tiene la ventaja de dar soporte a pantallas táctiles, lo cual nos permitiría portar nuestra aplicación a *tablets* o *smartphones* de forma sencilla.

6.7.1. Diseño de la interfaz gráfica

La interfaz gráfica es la siguiente:

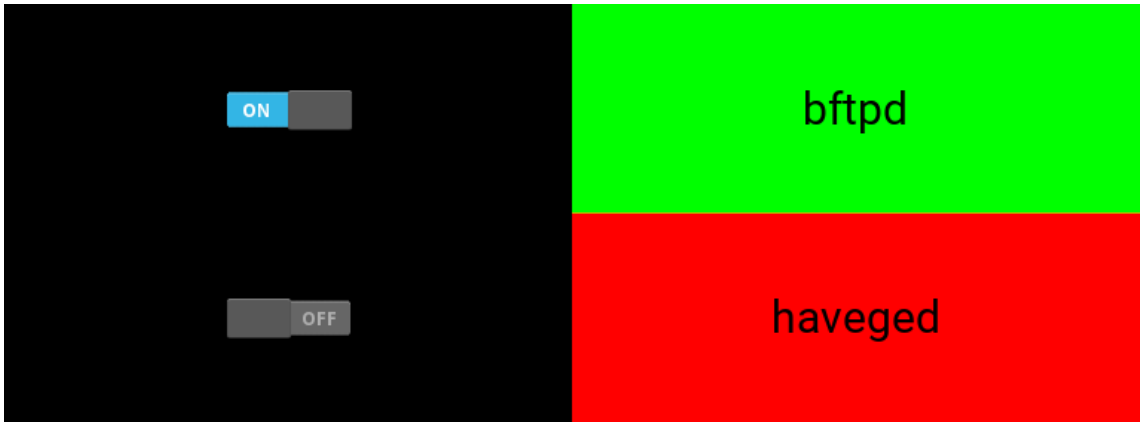

Figura 6.16: Interfaz gráfica

En la parte izquierda tenemos interruptores con los que podemos apagar y encender servicios. En el ejemplo solamente ponemos dos servicios, pero como veremos, es muy sencillo añadir más. En la parte derecha tenemos el nombre del servicio que apagamos y encendemos. Cambia de color de verde a rojo cuando lo apagamos.

6.7.2. Implementación

A continuación se explica las parte más importantes del programa.

Lenguaje `KV`

Kviy cuenta con un lenguaje propio llamado lenguaje `KV`, el cual se utiliza para construir interfaces gráficas de forma declarativa y explícita. Esto permite separar la interfaz gráfica de la lógica de nuestro programa.

kivyApp.py

```
<MyLabel@Label>:
    bcolor: 1, 0, 0
    color: 0,0,0,1
    font_size: '30dp'
    canvas.before:
        Color:
            rgb: self.bcolor
        Rectangle:
            pos: self.pos
            size: self.size
```

Definimos `MyLabel`, una etiqueta que hereda las propiedades de `Label`. Necesitamos definirla porque queremos poder cambiar el color del fondo de la etiqueta, lo cual no se puede en `Label`. Una vez definida, podemos utilizarla donde nosotros queramos.

kivyApp.py

```
<MyBoxLayout@BoxLayout>
    orientation:'horizontal'
    text:''
    service:''
    MySwitch:
        label:myswitch1
```

```
7           service:root.service
8           active:True
9       MyLabel:
10          id:myswitch1
11          text:root.text
```

Definimos `MyBoxLayout` a partir del `BoxLayout`, el cual consiste para poner elementos otros elementos dentro de él de forma horizontal o vertical. Cada `MyBoxLayout` estará dedicado a un servicio, por lo que le añadimos el campo `service`. El campo `text` tendrá el texto que se mostrará en la etiqueta.

Añadimos un `MySwitch` (el cual definiremos en el siguiente apartado) y un `MyLabel` y configuramos sus campos adecuadamente.

kivyApp.py

```
1  <GUI>:
2      BoxLayout:
3          orientation:'vertical'
4          MyBoxLayout:
5              text:'bftpd'
6              service:'bftpd'
7          MyBoxLayout
8              text:'haveged'
9              service:'haveged'
```

`GUI` será nuestra raíz en la jerarquía de elementos de la interfaz gráfica. Simplemente tiene un `BoxLayout` vertical, y dentro de éste, añadimos `MyBoxLayout`. En caso de querer añadir más servicios a la aplicación, simplemente tenemos que añadir más `MyBoxLayout` y modificar sus campos.

Lógica

kivyApp.py

```
1  class MySwitch(Switch):
2      label=ObjectProperty(None)
3      service=StringProperty()
4      def __init__(self,**kwargs):
5          super(MySwitch,self).__init__(**kwargs)
6
7      def on_active(self,instance,value):
8          if value is True:
9              print(self.service)
10             proc=subprocess.Popen("ssh raspi sudo  systemctl restart "+self
                   .service,stdout=subprocess.PIPE,shell=True)
11             self.label.bcolor=(0,1,0)
12         else:
13             proc=subprocess.Popen("ssh raspi sudo systemctl stop "+self.
                   service,stdout=subprocess.PIPE,shell=True)
14             self.label.bcolor=(1,0,0)
```

Definimos el objeto `MySwitch` a partir del `Switch`. La parte más importante es el método `on_active`, que se ejecuta cada vez que cambiamos el `MySwitch` de posición.

Utilizamos un subproceso para realizar una conexión SSH y ejecutar el comando `systemctl restart` o `systemctl stop` para encender y apagar el servicio que hayamos definido en el campo `service`.

kivyApp.py

```
class GUI(BoxLayout):
    pass

class MainApp(App):
    def build(self):
        return GUI()

MainApp().run()
```

Por último, definimos los objetos más altos de la jerarquía (en el caso de GUI no ponemos nada porque ya está definido en la parte de `KV`) y ejecutamos la aplicación.

CAPÍTULO 7
Conclusiones

Durante el desarrollo de este Trabajo de Fin de Grado, el lenguaje de comandos que hemos utilizado para escribir la mayoría de los *scripts* ha sido `bash` y hemos podido comprobar su gran utilidad en la automatización de diferentes tareas en sistemas operativos Unix-like. La elección del sistema operativo también ha sido acertada, ya que GNU/Linux está diseñado para que sea extensible y fácilmente automatizable.

Hemos cumplido los objetivos planteados inicialmente. En primer lugar, hemos automatizado la instalación de Arch Linux ARM en una Raspberry Pi y la conexión con ella a través de Ethernet de forma que los programas que realizan las distintas tareas sean fácilmente utilizables por terceras personas. Esta ha sido sin duda la parte más costosa del proyecto, sobre todo porque ha habido que probar diferentes ideas y implementarlas hasta llegar finalmente a la presentada en el trabajo. Además, es la parte donde más problemas se han encontrado y resuelto.

Respecto a la parte despliegue y automatización de servicios, hemos realizado unos *scripts* que cumplen con su cometido y que permiten que cualquier usuario pueda adaptarlos a sus necesidades

Por último, el programa que implementa la interfaz gráfica para encender y apagar servicios es un ejemplo sencillo que ilustra la potencia de Python. Una mejora que se podría implementar sería que para poder comprobar que el servicio se ha activado correctamente desde la interfaz, el color que indica el estado del servicio fuera obtenido preguntando periódicamente a la Raspberry Pi a través de SSH, en vez de depender del estado del objeto `MySwitch`.

Bibliografía

[1] Setup armv7h chroot under x86_64 host (Archlinux/Archlinuxarm biased)
Consultado en
https://gist.github.com/mikkeloscar/a85b08881c437795c1b9.

[2] Super User: How to create and format a partition using a bash script?
Consultado en
https://superuser.com/questions/332252.

[3] Manpage ARCHLINUX(7)
Consultado en
https://jlk.fjfi.cvut.cz/arch/manpages/man/archlinux.7.

[4] ArchWiki: compartir Internet
Consultado en
https://wiki.archlinux.org/index.php/Internet_sharing.

[5] ArchWiki: SFTP chroot
Consultado en
https://wiki.archlinux.org/index.php/SFTP_chroot

[6] ArchWiki
Consultado en
https://wiki.archlinux.org

[7] Arch Linux ARM
Consultado en
https://archlinuxarm.org

[8] Documentación de Kivy
Consultado en
https://kivy.org/doc/stable/

[9] *Manpages* de los programas utilizados

[10] Dan Mackin, Ben Whaley, Trent R. Hein, Garth Snyder, Evi Nemeth
UNIX and Linux System Administration Handbook.
Addison-Wesley Professional, 5ª edición, 2018.

[11] Cameron Newham, Bill Rosenblatt
Learning the bash Shell.
O'Reilly Media, 3ª edición, 2005.

[12] Mike Loukides, Tim O'Reilly, Jerry Peek, Shelley Powers
Unix Power Tools.
O'Reilly Media, 3ª edición, 2002.

[13] Dusty Phillips
Creating Apps in Kivy.
O'Reilly Media, 1ª edición, 2014.

[14] Piotr J Kula
Raspberry Pi 2 Server Essentials.
Packt Publishing, 1ª edición, 2016.

APÉNDICE A
Código

Todos los *scripts* desarrollados en este trabajo pueden encontrarse en https://github.com/joseprrm/TFG, así como otros *scripts* que se han escrito pero no se han explicado en la memoria.