UNIVERSIDAD DE SALAMANCA

Diseño e implementación de un sistema de computación distribuida con Raspberry Pi, y estudio comparativo del mismo frente a otras soluciones

by

Diego Martín Arroyo

Trabajo de Fin de Grado en el marco de los estudios de
Graduado en Ingeniería Informática

en la
Facultad de Ciencias
Departamento de Informática y Automática

11 de mayo de 2015

Declaración de Autoría

Yo, Diego Martín Arroyo, declaro que la autoría de este Trabajo de Fin de Grado titulado, 'Diseño e implementación de un sistema de computación distribuida con Raspberry Pi, y estudio comparativo del mismo frente a otras soluciones' y el trabajo presentado en el mismo corresponde a mi persona. Confirmo que:

- Este trabajo fue realizado completamente durante mis estudios del Grado en Ingeniería Informática en la Universidad de Salamanca.
- En aquellas partes de este Trabajo que han sido previamente presentadas como Trabajo de Fin de Grado o cualquier otro tipo de disertación en esta Universidad u cualquier otra institución, esto ha sido claramente indicado.
- Que todo el trabajo de terceros que ha sido consultado ha sido apropiadamente atribuido.
- Donde haya citado el trabajo de otros, la fuente ha sido siempre dada. A excepción de dichas citas, todo el conjunto del Trabajo ha sido realizado por mí.
- He reconocido todas aquellas fuentes de ayuda.
- Donde mi Trabajo ha sido parte de una colaboración con otras personas, he indicado claramente la extensión de mi trabajo y el de dichos terceros.

Firmado: _____

Fecha: _____

"Write a funny quote here."

If the quote is taken from someone, their name goes here

UNIVERSIDAD DE SALAMANCA

Abstract

Facultad de Ciencias
Departamento de Informática y Automática

Doctor of Philosophy

by Diego Martín Arroyo

The Thesis Abstract is written here (and usually kept to just this page). The page is kept centered vertically so can expand into the blank space above the title too...

Acknowledgements

The acknowledgements and the people to thank go here, don't forget to include your project advisor...

Índice general

Índice de figuras

Índice de cuadros

Abbreviations

LAH **L**ist **A**bbreviations **H**ere

Physical Constants

Speed of Light $c = 2{,}997\ 924\ 58 \times 10^8\ \mathrm{ms}^{-\mathrm{S}}$ (exact)

Symbols

a	distance	m
P	power	W (Js^{-1})
ω	angular frequency	rads^{-1}

For/Dedicated to/To my...

Capítulo 1

Introducción

La presente memoria recoge el proceso de instalación de un sistema distribuido formado por dispositivos Raspberry Pi y la creación de un conjunto de protocolos, herramientas y programas para la utilización del mismo como herramienta de investigación en el campo de la computación distribuida y como herramienta didáctica para disciplinas relacionadas con dicho área.

El sistema se compone de un conjunto de dispositivos físicos compuesto por los nodos de computación y una serie de módulos accesorios, así como los diferentes mecanismos de alimentación y refrigeración, un conjunto de herramientas software que permiten la coordinación y comunicación entre los diferentes procesos y una serie de herramientas que facilitan el trabajo con el sistema.

1.1. Contenidos de la memoria

- Definición del dominio del problema y motivación
- Evaluación de alternativas y propuesta de solución
- Plataforma física
- Herramientas creadas

    MarcoPolo

    MarcoTools

    MarcoStatusMonitor

    Deployer

    Material didáctico

Ricard Agrawala

- Aplicaciones distribuidas

  MPI

  Python

  Tomcat

- Evaluación
- Consulta a los estudiantes
- Evaluación de las prácticas en MPI
- Evaluación de las prácticas en Sistemas Distribuidos

Capítulo 2

Motivación

2.1. Introducción

Los límites físicos de los que adolecen los computadores en la época actual hacen de la computación distribuida y paralela un método para incrementar de forma sencilla y rentable el rendimiento total de un sistema, rendimiento que aumenta significativamente en problemas *ridículamente paralelos*[Citation needed].

Sin embargo, dichos beneficios conllevan una serie de inconvenientes, o el aumento de la complejidad de diversas tareas. En general, un sistema distribuido requiere de un conjunto de máquinas independientes, que en conjunto constituyen un coste superior al de un único nodo. Además, aparecen nuevos problemas de índole técnica: problemas de comunicación, depuración de aplicaciones, etcétera. Con el objetivo de solventar estos inconvenientes se han propuesto varias soluciones.

2.2. Situación actual

2.2.1. Computadores de placa única

Los computadores de placa única (*Single-Board Computer*) consisten en computadores de generalmente bajas prestaciones que aglutinan todos los componentes necesarios para su funcionamiento en un único circuito integrado. Dichas placas suelen tener un coste bajo y una relación rendimiento/coste muy elevada.

Durante los últimos años se han popularizado como una herramienta para el estudio y creación de sistemas distribuidos con un gran rango de propósitos diferentes.

2.2.1.1. RPiCluster (Joshua Kiepert)

Joshua Kiepert, estudiante de doctorado en la universidad Boise State, crea este sistema utilizando 33 computadores **Raspberry Pi B**, con el objetivo de utilizarlo como herramienta de pruebas que sirva de alternativa al supercomputador situado en su universidad[1], sobre el que trabaja de forma rutinaria, a fin de poder continuar su trabajo en periodos de mantenimiento, cierre del centro, etcétera. El sistema está diseñado para utilizar la *Message Passing Interface* y además poder utilizar los diferentes puertos de las placas (GPIO, I^2C, SPI, UART), puertos generalmente ausentes en computadores como clústeres. Utiliza además un servidor **NFS** para compartir datos entre todos los nodos, y un *router* dedicado para la interconexión. El sistema se complementa con un ordenador **Chromeboox** con el mismo sistema operativo (**Arch Linux**), que actúa como nodo coordinador.

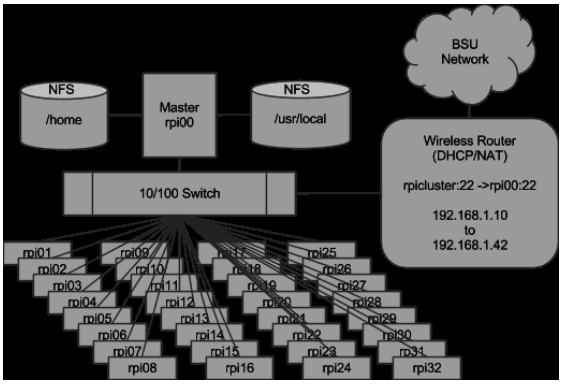

FIGURA 2.1: Estructura del sistema (Fuente: Joshua Kiepert).

Coste: 1967.21 dólares.

2.2.1.2. Dramble (Jeff Geerling)

El clúster *Dramble* es un conjunto de 6 equipos **Raspberry Pi** capaces de ejecutar en conjunto el gestor de contenidos **Drupal**[1]. El sistema es utilizado como servidor de pruebas para la ejecución de instancias de **Drupal** de forma experimental o durante demonstraciones en público[2].

[1] drupal.org

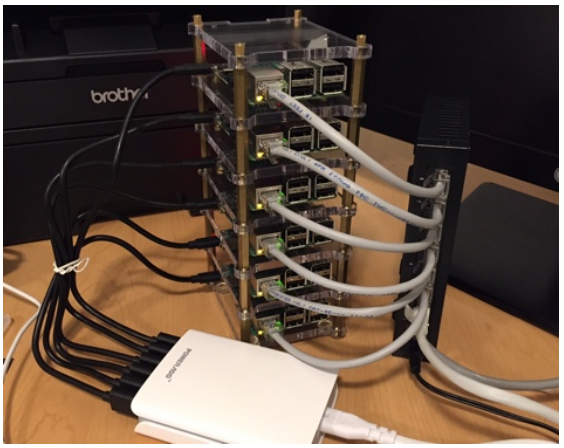

FIGURA 2.2: El *Dramble* en ejecución

Coste (estimado): 35 dólares por cada Raspberry Pi[Citation needed]

2.2.1.3. Bramble (GCHQ)

El organismo gubernamental *Government Communication Headquarters*, agencia de inteligencia del Gobierno Británico presentó en la *Big Bang Fair* de 2015 un proyecto educativo que combina 66 *Raspberry Pi* en un clúster jerárquico con 8 grupos de 8 nodos, cada uno de ellos con un coordinador, y dos nodos coordinadores. El cableado se reduce gracias al uso de la tecnología **PoE** (*Power over Ethernet*), y cada **Raspberr** cuenta con un conjunto de elementos adicionales, como un reloj de tiempo real, disco duro externo, cámara, punto de acceso WiFi, etcétera[3].

FIGURA 2.3: Vistazo general de la estructura del sistema

Coste (estimado)[Citation needed]

2.2.1.4. Clúster Iridis (Simon Cox, University of Southampton)

Con el objetivo de atraer a jóvenes estudiantes al mundo de la Computación, el profesor Simon Cox crea este clúster con 64 **Raspberry Pi B** sobre una estructura de LEGO[4]

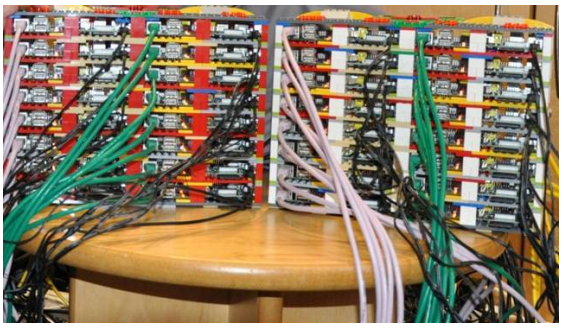

FIGURA 2.4: Sistema

2.2.1.5. Paralella

Capítulo 3

Dominio del problema

La utilización de algoritmos distribuidos implica mejoras sustanciales en una gran cantidad de aplicaciones, incrementando la capacidad global de computación de un sistema mediante la unión de varios dispositivos de cómputo que trabajan de como una única unidad indivisible a la vez que mantienen un alto grado de independencia y una tolerancia global a fallos muy alta. Sin embargo, el desarrollo de aplicaciones distribuidas implica el uso de un conjunto de nodos cuyo coste y mantenimiento es costoso.

Dicho aumento de la potencia implica una mayor complejidad en el desarrollo de algoritmos que puedan aprovechar de forma óptima este tipo de sistemas. Varios factores como la sincronización y la comunicación entre partes, o errores tales como condiciones de carrera son mucho más comunes que en otro tipo de aplicaciones. Dichas dircunstancias no solo dificultan el desarrollo de este sistema, sino también la comprensión de los fundamentos básicos de la Computación Distribuida.

Si bien la mayoría de las aplicaciones en las que el paradigma de computación distribuida introduce mejoras suelen exigir una gran capacidad de cálculo, su desarrollo únicamente requiere un conjunto de instancias independientes de un software (sistema operativo, contenedor de servicios...) con las que trabajar. Dicha característica implica que la utilización de nodos de precio reducido (o incluso reutilizados) para el diseño, análisis y evaluación de este tipo de algoritmos constituye una alternativa válida frente a sistemas de precio superior.

Sumada a dicha motivación existe el potencial aprovechamiento de este sistema como herramienta didáctica que facilite el aprendizaje de conceptos como el reparto de procesos, balance de carga o la compartición de recursos en asignaturas centradas en este tipo de conceptos dentro de los planes de estudio de Ingeniería Informática o titulaciones afines.

Con el presente proyecto se plantea la creación de un sistema con estas características aprovechando el bajo coste de los componentes del mismo que permita en primer lugar su utilización como herramienta de análisis y diseño (o incluso su utilización como plataforma definitiva) de aplicaciones distribuidas y en segundo lugar la posibilidad de uso como herramienta educativa.

A la hora de crear el sistema se realiza un análisis de las diferentes alternativas, a fin de escoger la alternativa que mejor satisfaga los objetivos definidos.

Figura: Tabla de alternativas

3.1. Objetivos del proyecto

Este proyecto cuenta con varios objetivos muy diferentes entre sí, que se agrupan en tres categorías:

- Arquitectura subyacente
  Definición de los componentes hardware a utilizar en el sistema, interconexión de los mismos, soluciones de alimentación eléctrica, almacenamiento....
- Servicios a proveer
  Conjunto de servicios que podrán ser aprovechados por diferentes clientes para explotar la capacidad de cálculo de las máquinas.
- Componente didáctico
  Creación de aplicaciones, herramientas y documentación como alternativa a las instalaciones típicas utilizadas actualmente.

3.2. Definiciones

3.2.1. Definición del dominio del problema

El sistema se ubica en una Facultad universitaria con aproximadamente 600 alumnos (**cita requerida**) con varias asignaturas en las que se imparten áreas de conocimiento relacionados con la Computación Distribuida, en particular las asignaturas **Arquitectura de Computadores** y **Sistemas Distribuidos** [5].

3.2.2. Modelado del sistema actual

La Facultad cuenta con varias aulas y laboratorios informáticos donde los alumnos disponen de la intraestructura necesaria para realizar los ejercicios y prácticas asignadas. Dichos espacios permiten utilizar cualquier equipo como nodo, pues se integran en la misma red, siendo incluso factible la comunicación directa entre equipos situados en diferentes aulas o incluso edificios. La conexión es relativamente rápida, contando con un cableado capaz de soportar teóricamente transferencias de hasta 100Mb/s de forma bidireccional (*full-duplex*) (**Cita requerida**). La gestión de un sistema de autenticación se realiza mediante el protocolo LDAP (*Lightweight Directory Access Protocol*) [6], contando con un sistema de ficheros centralizado que permite acceder a la información de un usuario desde cualquier equipo, facilitando las tareas de replicación de la información entre nodos.

La mayoría de las prácticas asignadas a los alumnos son desarrolladas en el lenguaje **Java**, ya conocido por la totalidad de los estudiantes gracias a asignaturas previamente cursadas (**Cita requerida**) y que facilita el despliegue y la compatibilidad entre diferentes equipos de trabajo sustancialmente. En ocasiones es necesario el uso de lenguajes como C y se plantean alternativas a *Java* como Python o C#.

Problemas conocidos Si bien la infraestructura existente es capaz de proveer a los estudiantes de los recursos necesarios, identificamos una serie de problemas inicialmente:

- Cada grupo de alumnos necesita tres estaciones de trabajo para poder realizar algunos de los ejercicios propuestos.
- El servidor LDAP constituye un "cuello de botella", pues todos los alumnos acceden a él de forma intensiva, provocando el fallo por exceso de peticiones del mismo.
- Las técnicas de programación utilizadas hasta la fecha tienen un rendimiento bajo y son en ocasiones relativamente complejas.

3.2.3. Identificación de usuarios participantes

- Estudiantes de tercero y cuarto curso del Grado en Ingeniería Informática.
- Doctentes de las asignaturas Arquitectura de Computadores y Sistemas Distribuidos.
- Administradores del Sistema.

3.3. Identificación de las necesidades de cada parte

3.3.1. Necesidades de los alumnos

- Entorno de trabajo sencillo que agilice el desarrollo de sus prácticas.
- Posibilidad de observar los resultados de las ejecuciones de forma sencilla.
- Facilidades para el despliegue de los diferentes ejecutables en todas las máquinas, así como el consumo de los servicios que estos implementen.

3.3.2. Necesidades de los docentes

- Entorno versátil sobre el cual puedan llevarse a cabo la totalidad de las prácticas y ejercicios propuestos, aportando si es posible algún tipo de ventaja sobre el sistema en uso.

3.3.3. Administrador

- Sistema integrable en la intraestructura actual cuyo mantenimiento sea sencillo y cuyo enfoque garantice la escalabilidad y su durabilidad.

3.4. Propuestas para la búsqueda de necesidades

- Encuestas o entrevistas a todas las partes.
- Evaluación de la experiencia de uso en las diferentes etapas de desarrollo del sistema.

3.5. Identificación de requisitos

3.5.1. Requisitos de almacenamiento de la información

- Gestión de usuarios (credenciales de autenticación)
- Gestión de los datos de cada usuario
- *Logs* del sistema

3.5.2. Identificación de requisitos funcionales

3.5.3. Identificación de requisitos no funcionales

- El *software* debe ser mantenible y robusto[1].
- Reducción de los costes de desarrollo.
- Definición de los protocolos de comunicación.
- Definición de los protocolos de seguridad y confidencialidad.
- Definición de la interacción con el usuario.
- Integridad del sistema y fiabilidad (*uptime*, recuperación frente a fallos).
- Productos a crear.
- Compatibilidad con las prácticas y ejercicios.

3.6. Evaluación de alternativas

A la hora de evaluar las diferentes opciones que satisfagan los requisitos descritos, se consideran los siguientes aspectos:

- Coste económico.
- Prestaciones técnicas (potencia de procesamiento, entrada/salida, capacidad de almacenamiento,facilidad de interconexión con otros materiales...).
- Facilidad de trabajo y de aprendizaje (documentación disponible, proyectos similares ya realizados, conocimiento sobre la plataforma en cuestión...).
- Escalabilidad del sistema.
- Necesidades de mantenimiento del sistema.
- Consumo del sistema (consumo eléctrico).
- Obsolescencia del sistema (número de años en los que el sistema podrá ser actualizado (tanto en hardware como software) y será capaz de seguir siendo una herramienta adecuada para el propósito planteado).

[1] Siendo dicha robustez garantizada mediante el uso de *software* utilizado por una base de usuarios significativa, una arquitectura conocida, pruebas realizadas sobre él o un equipo de desarrollo en activo, entre otras

3.6.1. Propuesta de solución: Virtualización de entornos de trabajo

Crear un conjunto de nodos virtuales dentro de una máquina que simulen un sistema distribuido

Ventajas intrínsecas de la solución Simplificación del sistema (reduce las necesidades de adquisición y mantenimiento de hardware). Gestión de varias partes del sistema (sistema de ficheros centralizado, gestión de usuarios...) de forma mas sencilla.

Inconvenientes intrínsecos del sistema No se exploran apenas las posibilidades de un sistema distribuido formado por varios equipos físicamente independientes.

Facilidad de trabajo y curva de aprendizaje Requiere una etapa de formación en materia de virtualización

Prestaciones técnicas

Coste econonómico

Escalabilidad del sistema Dependiente de las capacidades de virtualización del equipo disponible, y el número de nodos y usuarios a gestionar (previsiblemente alto)

Necesidades de mantenimiento Las necesidades propias de un sistema GNU/Linux junto a las específicas de la virtualización de los equipos.

Consumo energético del sistema

Obsolescencia del sistema

Material con el que se cuenta actualmente Se plantea el aprovechamiento de equipos pertenecientes a la Universidad, por lo que se estima un coste muy pequeño a la hora de adquirir material.

Otras características

3.6.2. Propuesta de solución: Clúster con equipos de escritorio

Se plantea la reutilización de equipos de escritorio pertenecientes a la Universidad que ya no se encuentran en uso (debido a su renovación, falta de potencia como PC...) para la creación de este sistema.

Ventajas intrínsecas de la solución La potencia del sistema es mucho mayor que la de cualquier otra solucion. Reduce dramáticamente el coste de adquisición de material y permite dar nueva vida a material universitario. La arquitectura es conocida

Inconvenientes intrínsecos del sistema No se exploran las características únicas de otros sistemas menos "convencionales", tales como la utilización de sistemas embebidos. Consumo mayor. Mayor demanda de espacio. Debido a el mayor tamaño de los equipos, puede llegar a ser complicado implementar las soluciones de visualizacion planteadas (LEDs, etc)

Facilidad de trabajo y curva de aprendizaje Soporte completo de casi la totalidad de las distribuciones de GNU/Linux. Las necesidades de manipulación de hardware se minimizan.

Prestaciones técnicas Arquitectura x86/x64 Entre 2 y 4 GB de memoria Conectividad Ethernet, USB Almacenamiento en disco duro

Coste econonómico

Escalabilidad del sistema Dependiente únicamente del coste económico de la adquisición de nuevos equipos

Necesidades de mantenimiento Las necesarias en cualquier sistema GNU/Linux y las específicas del montaje dado (en materia de refrigeración, gestión de cableado, etc)

Consumo energético del sistema

Obsolescencia del sistema

Material con el que se cuenta actualmente La Facultad de Ciencias ya dispone de los equipos, pues se plantea la reutilización de los mismos

Otras características

3.6.3. Clúster con equipos embebidos multimedia

Utilización de equipos embebidos diseñados para aplicaciones multimedia en el sistema (ejemplos son Chromecast, Apple TV, Amazon Fire TV...)

Ventajas intrínsecas de la solución Relacion potencia/precio presumiblemente superior a soluciones de coste similar como las placas Raspberry Pi.

Inconvenientes intrínsecos del sistema Dificultad de conexión (generalmente la conexión a red se realiza de forma inalámbrica, ausencia casi absoluta de cualquier conexión cuya finalidad no sea la emisión de vídeo o conexión con sistemas de almacenamiento mediante USB), falta de puertos GPIO, I2C...

Facilidad de trabajo y curva de aprendizaje Es difícil determinar la viabilidad de esta solución, pues no se cuenta con experiencia previa ni una documentación amplia al respecto. Es probable que sea necesaria la manipulación del sistema a muy bajo nivel. Lo cual incrementa el grado de complejidad de la solución.

Prestaciones técnicas Arquitectura ARM 2 núcleos a 1.2 GHz 512 MB de RAM Almacenamiento: 2 GB no expandibles Alimentación por microUSB

Coste econonómico

Escalabilidad del sistema Dependiente del coste de adquisición de nuevos equipos y las facilidades de interconexión de la plataforma (previsiblemente compleja, debido a la ausencia de sistemas de interconexión más allá de WiFi)

Necesidades de mantenimiento Dependiente del número de modificaciones que se realicen a las capas más bajas. En el peor de los casos puede que el administrador del sistema tenga que someterse a una etapa de formación para realizar un mantenimiento adecuado del sistema sin depender de desarrolladores previos. Las derivadas del mantenimiento de un sistema Linux sumadas a posibles problemas de interconexión si se utiliza una red inalámbrica (conexión a la LAN de la infraestructura local, interferencias...).

Consumo energético del sistema

Obsolescencia del sistema Difícil de determinar: no se cuenta con una gran cantidad de software para este tipo de sistemas más allá de las aplicaciones multimedia. No obstante, el sistema subyacente es conocido (Linux)

Material con el que se cuenta actualmente No se dispone de material de estas o similares características

Otras características

3.6.4. Clúster con equipos embebidos multimedia

Utilización de equipos embebidos diseñados para aplicaciones multimedia en el sistema (ejemplos son Chromecast, Apple TV, Amazon Fire TV...)

Ventajas intrínsecas de la solución Relacion potencia/precio presumiblemente superior a soluciones de coste similar como las placas Raspberry Pi.

Inconvenientes intrínsecos del sistema Dificultad de conexión (generalmente la conexión a red se realiza de forma inalámbrica, ausencia casi absoluta de cualquier conexión cuya finalidad no sea la emisión de vídeo o conexión con sistemas de almacenamiento mediante USB), falta de puertos GPIO, I2C...

Facilidad de trabajo y curva de aprendizaje Es difícil determinar la viabilidad de esta solución, pues no se cuenta con experiencia previa ni una documentación amplia al respecto. Es probable que sea necesaria la manipulación del sistema a muy bajo nivel. Lo cual incrementa el grado de complejidad de la solución.

Prestaciones técnicas Arquitectura ARM 2 núcleos a 1.2 GHz 512 MB de RAM Almacenamiento: 2 GB no expandibles Alimentación por microUSB

Coste econonómico

Escalabilidad del sistema Dependiente del coste de adquisición de nuevos equipos y las facilidades de interconexión de la plataforma (previsiblemente compleja, debido a la ausencia de sistemas de interconexión más allá de WiFi)

Necesidades de mantenimiento Dependiente del número de modificaciones que se realicen a las capas más bajas. En el peor de los casos puede que el administrador del sistema tenga que someterse a una etapa de formación para realizar un mantenimiento adecuado del sistema sin depender de desarrolladores previos. Las derivadas del mantenimiento de un sistema Linux sumadas a posibles problemas de interconexión si se utiliza una red inalámbrica (conexión a la LAN de la infraestructura local, interferencias...).

Consumo energético del sistema

Obsolescencia del sistema Difícil de determinar: no se cuenta con una gran cantidad de software para este tipo de sistemas más allá de las aplicaciones multimedia. No obstante, el sistema subyacente es conocido (Linux)

Material con el que se cuenta actualmente No se dispone de material de estas o similares características

Otras características

3.6.5. Clúster con Raspberry Pi

Utilizar la plataforma de hardware libre Raspberry Pi para la creación del sistema, disponiendo los diferentes equipos en un pequeño "rack" con un sistema de alimentación propio centralizado y una conexión directa a la infraestructura local.

Ventajas intrínsecas de la solución Existen varias soluciones similares bien documentadas. El hardware es flexible, barato y el consumo es pequeño. Gran comunidad de desarrolladores alrededor de la plataforma.

Inconvenientes intrínsecos del sistema La potencia del sistema es pequeña (ver seccion prestaciones técnicas)

Facilidad de trabajo y curva de aprendizaje Ya se cuenta con experiencia en el manejo de estas placas. Amplia documentación de las prestaciones de la misma. Proyectos similares ya realizados. Soporte completo de varias distribuciones de GNU/Linux

Prestaciones técnicas Arquitectura ARM Entre 512 MB y 1 GB de memoria 1 o 4 Núcleos a 700 o 900 MHz (overclock a 1 GHz de forma segura) Conectividad Ethernet, I2C, GPIO Alimentación por microUSB/GPIO Almacenamiento entre 1 GB y 256 GB mediante tarjetas MicroSD/SD

Coste econonómico

Escalabilidad del sistema Dependiente únicamente del coste económico de la adquisición de nuevos equipos

Necesidades de mantenimiento Las mismas que cualquier sistema GNU/Linux de iguales características. Pueden surgir problemas con la fuente de alimentación, dado que es una solución propia.

Consumo energético del sistema Variable según modelo, entre 3 y 4 W, con 5V de tensión y un amperaje variable entre 0.6 y 0.8 A

Obsolescencia del sistema El software de terceros (sistema operativo, bibliotecas, etc) a incluir está respaldado por una comunidad extensa que provee actualizaciones de forma continua, por lo que previsiblemente el sistema podrá estar actualizado durante varios años. Las necesidades que el sistema cubre no demandarán previsiblemente una mayor potencia de cálculo.

Material con el que se cuenta actualmente El Departamento de Informática y Automática cuenta con varios de estos equipos se plantea la reutilización de los mismos

Otras características

3.6.6. Elección de la solución

3.6.7. Raspberry Pi: Elección de las características básicas del sistema

Comparativa de las características relevantes de los diferentes modelos de Raspberry Pi. Quedan descartados los modelos A y A+ por la carencia de puerto Ethernet (amén de otras características necesarias).

	Modelo B	Modelo B+	Modelo B 2
Procesador	ARMv6 1 Núcleo, 700 MHz (safe overclock hasta 1GHz)	ARMv6 1 Núcleo, 700 MHz (safe overclock hasta 1GHz)	ARMv7 4 Núcleos a 900 MHz
Memoria	512 MB compartidos con GPU	512 MB compartidos con GPU	1 GB compartido con GPU
Evaluación de rendimiento con LINPACK [7–9]	40.64	40.64	92.88
Conexiones	2 USB, GPIO de 8 pines. Ethernet 10/100	4 USB, GPIO de 17 pines. Ethernet 10/100	4 USB, GPIO de 17 pines. Ethernet 10/100
Consumo medio [Citation needed]	700 mA, 5 V (3.5 W)	600 mA, 5 V (3 W)	800 mA, 5 V (4 W)
Almacenamiento	SD	MicroSD	MicroSD
Alimentación	Mediante MicroUSB o los pines GPIO	Mediante MicroUSB o los pines GPIO	Mediante MicroUSB o los pines GPIO
Sistemas operativos compatibles	Archlinux ARM, OpenELEC, Puppy Linux, Raspbmc, RISC OS, Raspbian, XBian, openSUSE, Slackware ARM, FreeBSD, Plan 9, Kali Linux, Sailfish OS, Pidora (Fedora Remix), Lista completa en [Citation needed]	Los mismos que para el modelo B	Hasta la fecha, únicamente: Ubuntu Snappy Core, Raspbian, OpenELEC, RISC OS, Según la web de ArchLinux, también soporta este sistema operativo [2]
Otros	Modelo descatalogado, el soporte oficial y proporcionado por la comunidad probablemente será menor que para los modelos más recientes en el futuro.		Lleva poco tiempo en el mercado (apenas un mes). Se conocen pequeños fallos en el hardware (fotosensibilidad de algún componente).

3.6.8. Elección del sistema operativo

Nombre	Enfoque	Características notables	Ventajas	Inconvenientes	Software disponible
ArchLinux ARM	Distribucion ligera centrada en el minimalismo y la disponibilidad de software novedoso. Requiere sin embargo que el usuario conozca el entorno GNU/Linux antes de utilizarlo	Muy optimizado con un ciclo de desarrollo que permite contar con software puntero en poco tiempo	Eficiente, gran comunidad alrededor, relativamente sencillo de utilizar	En ocasiones puede ser complejo su uso. Ya no se incluye en las distribuciones por defecto de la Fundacion Raspberry Pi, lo cual puede suponer falta de soporte oficial	8700 paquetes disponibles en los repositorios oficiales, más pequeño que para otras distribuciones, si bien no se ha encontrado aun software no compatible
Ubuntu Snappy Core	Centrado en la facilidad de uso	Es la distribución más popular (en equipos de escritorio) con gran cantidad de paquetes disponible	Fácil de configurar, gran cantidad de soporte	Aún no ha sido probado en la Raspberry de forma intensiva.El rendimiento de ubuntu suele ser menor al de otros sistemas operativos debido a la gran cantidad de paquetes incluidos por defecto.	
Raspbian					

Nombre	Enfoque	Características notables	Ventajas	Inconvenientes	Software disponible
RISC OS	Diseñado específicamente para la arquitectura ARM, aprovechando las posibilidades de dicha arquitectura	Eficiente, basado en el RISCOS original, incluyendo características del mismo. Sistema monousuario con multitarea cooperativa (en contraste con multihilo o multitarea apropriativa)	Muy eficiente	No esta basado en un sistema conocido previamente. Relativamente desfasado en cuanto a la arquitectura del Sistema Operativo. El software suele ser programado en BBC BASIC	
Gentoo	Diseñado para permitir la personalización del sistema al máximo nivel posible	Enfocado en la personalizacion, siendo el sistema compilado en la maquina sobre la que se va a utilizar en vez de ser descargado como archivo binario	Permite ser modificado de forma sencilla	Poco soportado en Raspberry Pi	
Windows 10	Diseñado para el paradigma IoT	Sencillo de utilizar, con soporte (previsiblemente) del *framework* .NET	Aún no se encuentra disponible[10]. Esta diseñado para un proposito especifico. No compatible con software para Linux de forma nativa		

Capítulo 4

Herramientas

La complejidad que acarrea el uso de aplicaciones distribuidas hace necesario el uso de herramientas que permitan el desarrollo de forma cómoda del propio sistema, su uso posterior como herramienta de prueba de aplicaciones distribuidas y por último, facilitar el aprendizaje de algoritmos y herramientas distribuidas.

Muchas de las aplicaciones distribuidas utilizadas incluyen varias herramientas para facilitar su uso. Sin embargo estas soluciones suelen ser diseñadas para el propósito específico de dicha aplicación, y son difíciles de adaptar a otros contextos. Debido a esta carencia, se han creado varias herramientas propias que permiten aprovechar al máximo este sistema.

4.1. MarcoPolo, el protocolo de descubrimiento de servicios

Uno de los problemas típicos a la hora de crear un sistema distribuido es la localización de cada uno de los nodos que lo conforman. Soluciones como servicios de nombres (DNS) permiten crear estructuras jerárquicas donde cada nodo está identificado por un nombre previamente conocido. También existen protocolos inspirados en este como **mDNS** (*Multicast Domain Name Service*) donde la necesidad de un servidor de nombres desaparece, y los nodos son capaces de encontrarse entre ellos mediante multicast[11]. Otras alternativas como Bonjour, Avahi o AppleTalk (ya descontinuado) también han sido evaluadas.

Sin embargo, estas y otras soluciones similares no responden a una de las necesidades básicas del sistema a construir: la condición de que la información que conoce cada nodo sobre el resto en el arranque del sistema es nula. Si bien con **mDNS** evitamos contar con un servidor de nombres, debemos conocer el nombre de cada máquina o

esta debe anunciarse en la red antes de poder estar disponible (mDNS Probing). Dicho inconveniente se suma al hecho de que **DNS** y protocolos similares son creados con el único propósito de resolver la correspondencia nombre - dirección de red de un equipo, y son difícilmente extensibles a otro tipo de aplicaciones. Además, la mayoría de los protocolos asumen que la información de un nodo presente de una red local es de interés para el resto de nodos de la red, lo cual dificulta la independencia de un conjunto de equipos frente al resto.

Una de las piezas clave del sistema consiste en la escalabilidad del mismo en tiempo real: no es necesario conocer qué nodos participan en el sistema hasta que no se vayan a utilizar. Además, se pretende optimizar al máximo cada uno de los nodos del sistema por separado, por lo que dedicar uno de ellos como "autoridad" frente a la que el resto de nodos se registren y esta actúe posteriormente como nodo coordinador y "resolver" supone una dedicación de recursos innecesaria y que dificulta la escalabilidad del sistema. Además, la gestión del espacio de direcciones de la red en la que se integra el sistema no es gestionado por este y además es compartido con una gran cantidad de equipos adicionales. Esto implica que las direcciones de cada nodo son asignadas por un servidor DHCP (*Dynamic Host Configuration Protocol*) sobre el que no se tiene control, y cuyas direcciones son asignadas para intervalos de tiempo pequeños[1]. Por otro lado, la clave de este sistema no la constituye la disponibilidad de un nodo, sino las aplicaciones distribuidas que pueden utilizarse en el mismo (de ahora en adelante denominaremos a estas "servicios"). Un nodo puede contar con un conjunto de servicios diferente al de sus vecinos, y por tanto colaborará en unas tareas y en otras no en virtud de dicha disponibilidad. Este requisito no es satisfecho por la mayoría de los sistemas anteriormente mencionados.

Motivada por esta serie de características surge la necesidad de crear un pequeño protocolo de descubrimiento de nodos basado principalmente en los servicios que dichos nodos pueden (y desean) ofrecer al conjunto de la malla. Además, siendo uno de los objetivos funcionales del sistema el aprovechamiento del mismo como herramienta didáctica, surge la necesidad de que dos conjuntos de nodos puedan trabajar en la misma red de forma independiente. Como aproximación para satisfacer estas necesidades surge el procolo de descubrimiento de servicios **MarcoPolo**

4.1.1. MarcoPolo: Introducción

MarcoPolo es un protocolo de descubrimiento de servicios cuya dinámica y nombre se inspiran en el juego homónimo[Citation needed], en el cual uno de los integrantes debe encontrar

[1] Durante el desarrollo del sistema se observa que las direcciones son asignadas por periodos de tiempo pequeños y no suelen repetirse a menos que dicha dirección no haya sido asignada anteriormente, fenómeno que suele darse con bastante frecuencia.

al resto privado de visión mediante ecolocalización (gritando la palabra clave "Marco", cuya respuesta por parte del resto de jugadores es "Polo"). El protocolo se compone de dos roles claramente diferenciados (y prácticamente independientes aún siendo ejecutados en el mismo nodo): **Marco**, encargado de enviar consultas a la red y **Polo**, que emite una respuesta a dichos comandos y gestiona la información de cada nodo.[Citation needed]

Con el objetivo de posibilitar la coexistencia de varias "mallas" de nodos independientes (donde los servicios ofrecidos por un nodo únicamente sean conocidos y consecuentemente aprovechables por el resto de sus vecinos) a la vez que las consultas son realizadas a todos los integrantes sin necesidad de conocer su identificador en la red (dirección a nivel de red o enlace, nombre *DNS*) se utilizan mensajes uno-a-muchos, conocidos generalmente con el nombre *multicast*, donde cada una de las *mallas* se comunicará con el resto de integrantes de la misma a través de un grupo preestablecido (o consensuado por dichos nodos).

4.1.1.1. Objetivos

- Independencia El protocolo debe ser compatible con el mayor número de aplicaciones posible, adaptándose. Dicho objetivo se consigue delegando una gran parte de la funcionalidad a aplicaciones que se apoyan sobre el protocolo, en vez de implementar dicha funcionalidad en este. Dicho desacoplamiento permite, gracias a la mayor simplicidad del protocolo, poder ser compatible con un mayor número de casos de uso.
- *Zeroconf*. El protocolo funciona en una red sin requerir ningún tipo de configuración por parte del usuario, y en una gran cantidad de casos, sin gran esfuerzo por parte del administrador.
- Segmentación Varias instancias del protocolo pueden ejecutarse en una misma red de forma independiente, permitiendo la creación de varias "mallas" de equipos. Dicha segmentación no debe alterar en absoluto el esquema de la red preexistente.
- Conectable Las diferentes aplicaciones presentes en los diferentes nodos deben poder aprovechar la funcionalidad del protocolo mediante una serie de elementos conectores (interfaces).
- Seguro En aquellos casos en los que la información compartida por los nodos sea confidencial, el protocolo debe implementar las medidas oportunas para la protección de la misma.
- Independiente de plataforma e implementación Toda la comunicación entre elementos del protocolo se realizará a través de tecnologías que no dependan de una

implementación concreta, tales como un lenguaje de programación o un sistema operativo dado. El único elemento que puede presentar tal dependencia es el conector final con otro código fuente, así como cualquier otro punto final (comandos, ficheros de configuración, etc).

- Independiente del espacio de direcciones, nombres de red y cualquier otro elemento El protocolo debe funcionar en cualquier espacio de direcciones dado, sin considerar en cualquier caso la dependencia con protocolos como **DHCP** o **DNS**.
- Simplicidad Los comandos del protocolo deben ser simples y, en caso de que sea posible, deben ser similares a otros ya conocidos por los usuarios del sistema, a fin de que estos ya estén familiarizados con los mismos.
- Descentralización El protocolo no debe en ningún momento generar un "cuello de botella", a menos que sea la opción más adecuada para la una tarea dada[2].
- Visibilidad
- Optimización de la red
- Sin conexión
- Extensible
- Extensible a diferente hardware

4.1.2. Comandos

El protocolo consiste en una serie de mensajes (a partir de ahora denominados *comandos*) que contienen las consultas sobre la información de uno o varios servicios, nodos o información sobre la propia *malla* que un nodo desee conocer, así como la respuesta a dichas consultas. Dichos mensajes son enviados como cadenas de texto que almacenan la información en estructuras de datos JSON (*JavaScript Object Notation*) debido a la gran legibilidad de estas por humanos y la gran cantidad de herramientas disponibles para su creación y procesado.

Los comandos de MarcoPolo constituyen las primitivas del protocolo. Actualmente se cuenta con las siguientes primitivas y las correspondientes respuestas:

[2]Como se detalla más adelante, dichas situaciones se han desplazado a las aplicaciones que utilizan MarcoPolo

Nombre	Emisor	Función	Información	Respuesta esperada	Protocolo y puerto
Marco	Marco	Descubrir todos los nodos presentes en la *malla*	Únicamente se incluye el nombre del comando	Un comando *Polo* por cada nodo disponible en la red, incluyendo como parámetros opcionales información sobre el nodo o *ninguna* si no existe ningún nodo disponible.	UDP *multicast* al puerto 1338.
Polo	Polo	Informar a un nodo de la existencia del emisor	Información sobre el nodo opcional (servicios disponibles, información sobre el nodo o la instancia de Polo...)	*Ninguna*	UDP *unicast* al puerto efímero del mensaje de pregunta.
Request-For	Marco	Conocer todos los nodos que ofrecen un servicio identificado por su nombre único en el sistema	Identificador unívoco del servicio a descubrir	**OK** con información opcional sobre el nodo o el servicio	UDP *multicast* al puerto 1338.
OK	Polo	Comando utilizado para emitir una respuesta a una petición, siendo la información de interés contenida en los parámetros de respuesta.	Respuesta a un comando con la información solicitada	*Ninguna*	UDP *unicast* al puerto efímero de la pregunta.
Services	Marco	Descubrir todos los servicios ofrecidos por un nodo	No se envía información adicional con el comando	**OK** con una lista de los identificadores del servicio o *ninguna* si el nodo no está en la red.	UDP *unicast* al puerto 1338.

4.1.3. Esquemas de comunicación

4.1.3.1. Comando Marco

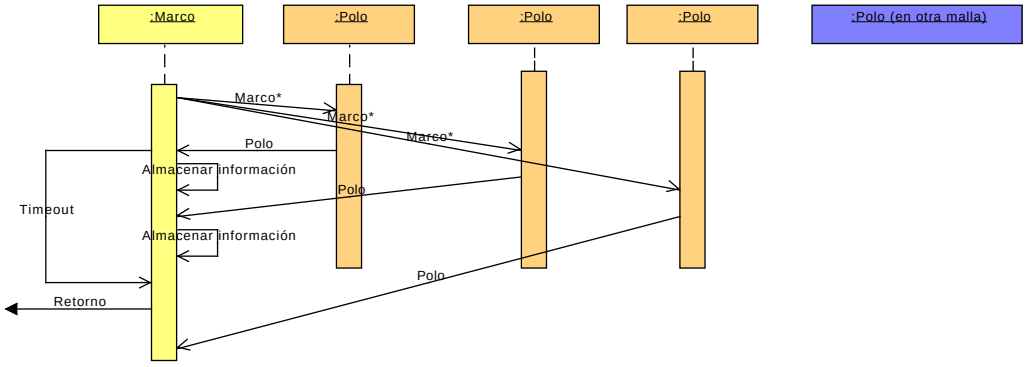

FIGURA 4.1: Interacción al enviar el comando **Marco**. Los mensajes a grupos *multicast* se indican con "*"

El comando Marco se envía al grupo *multicast* definido en la configuración de la instancia local de **Marco**. Los nodos suscritos a dicho grupo (aquellos que pertenecen a la "malla") reciben el mensaje y emiten una respuesta **Polo**. Debido a la falta de una conexión entre los nodos (debido a que todos los mensajes son intercambiados utilizando el protocolo UDP) se fija un tiempo de espera de respuesta, durante el cual se reciben y acumulan todas las respuestas. Al final dicho tiempo de espera, se retornan los resultados y el resto de respuestas son ignoradas.

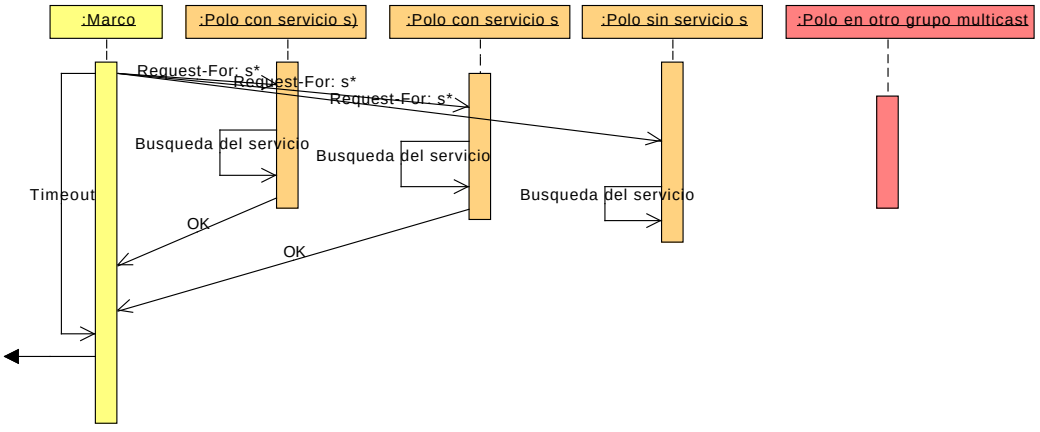

FIGURA 4.2: Diagrama de interacción al enviar el comando **Request-For**. Los nodos comprueban si deben ofrecer el servicio identificado por la clave *s*. En caso de que la búsqueda sea exitosa se retorna un mensaje indicando la disponibilidad de dicho nodo. En caso contrario no habrá respuesta alguna. Los mensajes enviados a grupos *multicast* se indican con "*"

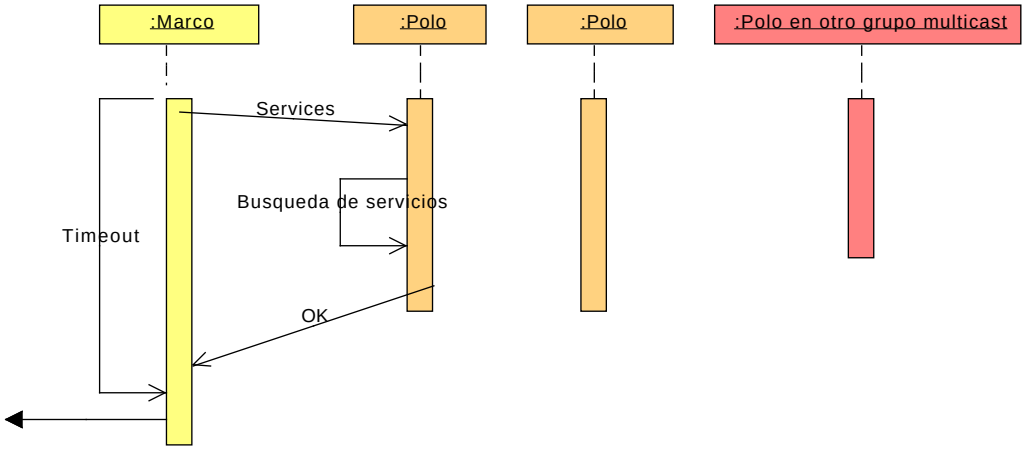

FIGURA 4.3: Diagrama de interacción al enviar el comando **services**. El nodo al que se le envía el comando consulta la información sobre los servicios que posee y posteriormente envía una respuesta a la instancia de Marco que ha realizado la consulta. Obsérvese toda la información es enviada en modo *unicast*

4.1.4. Arquitectura en detalle

La funcionalidad del protocolo se segmenta en dos roles claramente definidos e identificados: **Marco** y **Polo**. Dicha funcionalidad se implementa en dos ejecutables completamente independientes, que pueden por tanto coexistir o ser ejecutados sin presencia del otro elemento.

Dichos ejecutables son iniciados al arranque el equipo, aprovechando para ello las herramientas que el sistema operativo provee[3], y se ejecutan en segundo plano de forma continua (es por ello pueden ser categorizados como procesos *daemon*).

Toda la funcionalidad se ejecuta en un único proceso que se encarga de la creación de los diferentes canales de comunicación (utilizando la API de *sockets* de Berkeley). Dichos canales de comunicación son gestionados por la utilidad **Twisted**, que simplifica el trabajo con la API, en particular a la hora de crear *sockets* asíncronos.

4.1.4.1. Configuración

Todos los aspectos modificables de cada rol, tales como el grupo *multicast* al que suscribirse o el tiempo de espera predeterminado se definen en un archivo de configuración alojado en el directorio `/etc/marcopolo` (siguiendo la estructura definida en el *Filesystem Hierarchy Standard* [12]).

[3]Los ejecutables han sido configurados para ser compatibles con el inicializador **init** y el más reciente **systemd**.

data.txt

```
.
 marco
ãã  marco.conf
 polo
     polo.conf
     services
         deployer
         statusmonitor
         tomcat-carrera
         tomcat-isis
         tomcat-ntp

3 directories, 7 files
```

FIGURA 4.4: Árbol de directorios dentro del directorio de configuración

Los archivos de configuración de cada uno de los *daemons* sigue la típica estructura clave-valor presente en archivos de configuración de servicios del sistema. Por el contrario, la información de todos los servicios a ofrecer sigue la sintaxis de un fichero **JSON**[4]. Todos estos ficheros son leídos al arrancar el ejecutable, y su modificación no tendrá efectos hasta la próxima vez que se inicie el servicio (salvo excepciones que veremos a continuación).

```
{
    "id": "statusmonitor",
    "version": "1"
}
```

LISTING 4.1: Un archivo que describe el servicio status monitor

4.1.4.2. Archivos auxiliares

Log Toda la información sobre la ejecución de los *daemons* se refleja en los archivos de *log* presentes en el directorio `/var/log/marcopolo`. El nivel de log se configura en el parámetro `LOGLEVEL` de cada uno de los *daemons* y puede tomar uno de los siguientes valores:

- **Error** Errores internos durante la ejecución.
- **Warn** Advertencias sobre posibles situaciones atípicas.

[4]La razón de esta decisión de diseño es la facilidad de interpretación de dicho formato y la legibilidad que ofrecen.

- **Info** Información de interés sobre el funcionamiento del sistema.
- **Debug** Información de depuración.

Registro de ejecución En ocasiones es necesario conocer el identificador del proceso **PID** del *daemon*. Para ello se almacena en el directorio `/var/run/marcopolo/(marco.pid|polo.pid)` dicho identificador, que puede ser aprovechado por el gestor de arranque del proceso.

4.1.5. Integración de los *daemons* en el sistema operativo

Los *daemons* se integran en el arranque del sistema a través de los ficheros de configuración de `init`[Citation needed] o `systemd` dependiendo del gestor disponible en el sistema operativo sobre el que se ejecuten los procesos.

Por defecto los *daemons* se ejecutan durante todo el ciclo de vida del computador, pero pueden ser reiniciados o detenidos arbitrariamente por voluntad del administrador:

```
systemctl start (marco|polo)
systemctl stop (marco|polo)
systemctl restart (marco|polo)
systemctl reload polo #Orden exclusiva de Polo
```

El comando `reload` permite actualizar la lista de servicios que **Polo** ofrece sin tener que detener todo el proceso para ello. Dicho comportamiento se consigue de forma similar al comando `reload` de Apache, enviando la señal `SIGUSR1` al proceso.

4.1.6. Conexiones con MarcoPolo (*Bindings*)

La funcionalidad de **MarcoPolo** no se limita al descubrimiento de los servicios del sistema, por lo que es necesario proveer a los usuarios del clúster de herramientas que permitan integrar sus aplicaciones distribuidas con estos servicios. Dichas herramientas, conocidas generalmente como *bindings*, permiten exponer públicamente la funcionalidad de **MarcoPolo** para que pueda ser aprovechada por otros usuarios.

Se han creado *bindings* para los lenguajes de programación **Python** y **Java** y se plantea crear uno para el lenguaje **C**. Todos ellos son consistentes entre sí, y utilizan la misma sintaxis para realizar el mismo tipo de operación a la vez que aprovechan las características propias de cada lenguaje. Dicha filosofía está inspirada en el funcionamiento de las primitivas de la API de resolución de nombres en red (`netdb.h`)[13], por lo que los `bindings` se comunican con la instancia local de Marco o Polo a través de *sockets* vinculados a la dirección IP local (127.0.1.1).

Todos los *bindings* deben implementar el mismo conjunto de primitivas, a saber:

Primitivas en el *binding* de Marco

- `request_for(service, timeout=None)` Retorna una lista de nodos que ofrecen el servicio indicado en `service`. Esta función bloquea la ejecución del proceso hasta que el tiempo de espera de nuevas respuestas se cumple (si bien esto no constituye un problema para la mayoría de aplicaciones, es importante que sea conocido por el programador). Si se especifica un `timeout`, este se utiliza en lugar del determinado por defecto en los parámetros de configuración de **MarcoPolo**. Se lanza una excepción o un código de error en caso de que la comunicación con la instancia de **Marco** sea infructuosa (generalmente este tipo de problemas se originan debido a un fallo en el arranque del servicio). Toda la información es transferida en cadenas JSON codificadas en UTF-8.

- `getOneNode(criteria=None, timeout=None)`

  Retorna un nodo elegido aleatoriamente entre las respuestas (en concreto, el nodo cuya respuesta llegue primero). Si se especifica un criterio en la variable `criteria` se elegirá el nodo que mejor satisfaga dicho criterio.

- `getAllNodes(timeout=None)` Retorna todos los nodos disponibles en la *malla* sin considerar los servicios ofertados. Se lanza una excepción o un código de error en caso de que la comunicación con la instancia de **Marco** sea infructuosa.

- `getNodeInfo(ip)` Obtiene la información de un nodo identificado por su `ip` si este está disponible en la red.

Primitivas en el *binding* de Polo

- `register_service(service, params=None)`

  Añade un nuevo servicio al conjunto de servicios ofertados. El servicio únicamente será ofertado durante el ciclo de vida de la instancia local de Polo. Si esta es detenida o reiniciada se procederá a la eliminación del registro. Para registrar un servicio de forma permanente es necesario definirlo en el directorio `/etc/marcopolo`

- `remove_service(service)` Elimina un servicio de la lista de ofertados. Para poder realizar este proceso es necesario ser el "propietario" del servicio. Esto es, el único proceso que puede eliminar un servicio es aquel que lo creó o en su defecto la instancia de **Polo**. En caso de que esta restricción sea quebrantada, una excepción o código de error será retornado.

- `have_service(service)` Indica si el servicio está ofertado o no.

Como se puede observar, la mayoría de primitivas tienen como objetivo el descubrimiento y publicación de servicios. Sin embargo, varias de ellas permiten realizar consultas sobre la información del propio nodo y se plantea la creación de más primitivas que sigan dicha filosofía.

4.2. Aplicaciones construidas sobre MarcoPolo

4.2.1. Utilidades

A fin de simplificar al máximo el funcionamiento de los *daemons* varias utilidades que podrían tener cabida dentro del propio protocolo han sido creadas como utilidades independientes que aprovechan la funcionalidad de **MarcoPolo** para realizar su cometido, pero cuya interdependencia se limita a dichos canales de comunicación.

4.2.1.1. `marcodiscover`

Esta utilidad consiste en un comando que permite ejecutar consultas al sistema a través de un intérprete de órdenes. El comando posibilita realizar la mayoría de consultas de interés y cuenta con varias opciones para dar diferentes formatos a la salida por pantalla, algo que, como veremos posteriormente, es de gran utilidad para la ejecución de un conjunto particular de programas.

Las opciones del comando son las siguientes:

```
usage: marcodiscover.py [-h] [-d [ADDRESS]] [-s [SERVICE]] [-S [SERVICES]]
                        [-n [NODE]] [--sh [SHELL]]

Discovery of MarcoPolo nodes in the subnet

optional arguments:
  -h, --help            show this help message and exit
  -d [ADDRESS], --discover [ADDRESS]
                        Multicast group where to discover
  -s [SERVICE], --service [SERVICE]
                        Name of the service to look for
  -S [SERVICES], --services [SERVICES]
                        Discover all services in a node
  -n [NODE], --node [NODE]
                        Perform the discovery on only one node, identified by
                        its ip/dns name
  --sh [SHELL], --shell [SHELL]
                        Print output so it can be used as an interable list in
                        a shell
```

FIGURA 4.5: Diferentes opciones de configuración de `marcodiscover`

4.2.2. Aplicaciones del sistema

A fin de aprovechar la funcionalidad de **MarcoPolo** dentro del sistema, se crean las siguientes utilidades

4.2.2.1. Status Monitor

El monitor de estado consiste en una aplicación con interfaz web que permite observar las estadísticas de uso del *hardware* y de diversos procesos. Utiliza para la detección de los diferentes nodos el *binding* de **Marco** en Python que realiza una consulta para descubrir que nodos están dispuestos a ofrecer el servicio `statusmonitor`. La respuesta de dicho comando es enviada al cliente, que establece conexiones directas a cada uno de los nodos a través de *Websockets*[Citation needed]. Esto es posible debido a que según la especificación del estándard de websockets, la *Same-Origin Policy*[14] no es utilizada de la misma forma que en peticiones HTTP,[15].

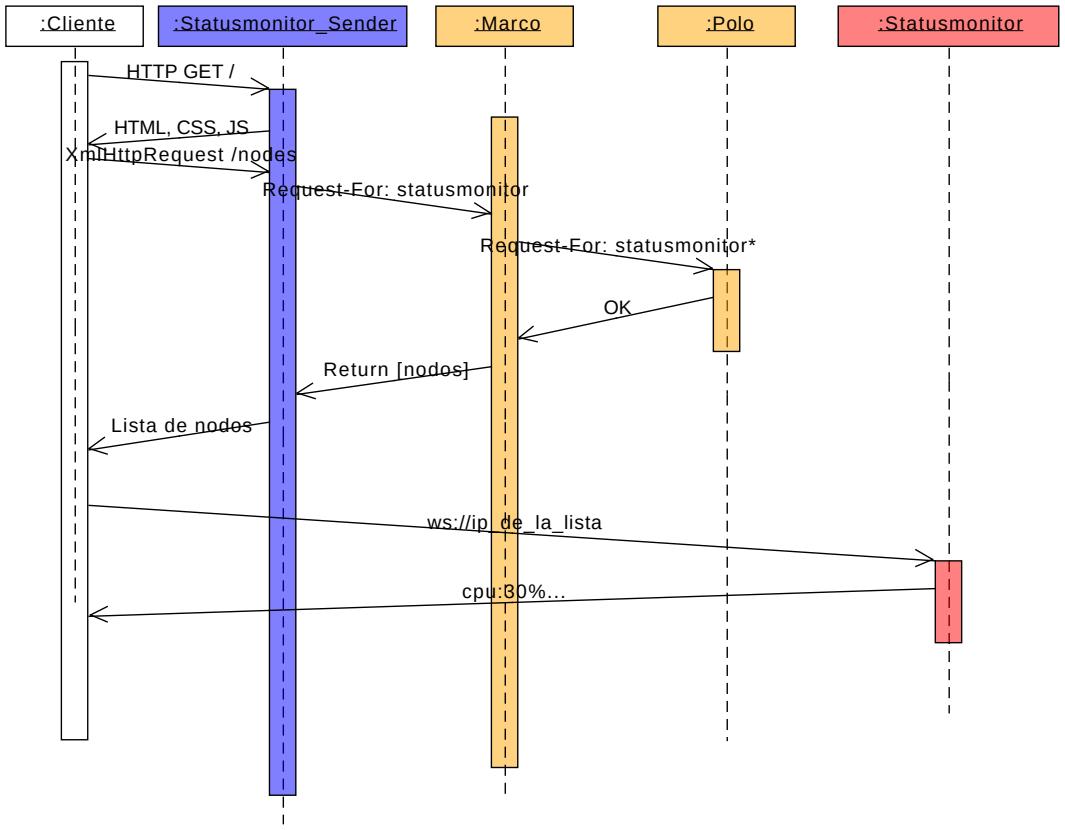

FIGURA 4.6: Interacción completa del usuario con **statusmonitor**. Los mensajes a grupos *multicast* se indican con "*"

. El usuario se conecta a la página web, que en respuesta envía un código *JavaScript* (además del código HTML y CSS) que solicita la lista de nodos disponibles. Una vez recibida la petición de los nodos disponibles, el servidor solicita dicha información a través de su instancia local de **Marco** (utilizando para ello un *binding*. Cuando la instancia de **Marco** termina de recoger las respuestas, retorna la información al servidor, que a su vez retorna dicha información al cliente. Al recibir dicha información, el nodo crea una conexión *Websocket* con el servicio statusmonitor que se encarga de enviar por dicha conexión la información local a intervalos de tiempo definidos.)

FIGURA 4.7: Vista de la interfaz web una vez obtenidos los nodos y establecida la conexión a los mismos. Se observa el porcentaje de memoria y principal y de intercambio utilizadas, la temperatura del procesador, los procesos con más consumo de CPU

Para conocer la información sobre el sistema el proceso servidor utiliza varios comandos y ficheros auxiliares, destacando:

- `top` Para conocer la información sobre los procesos más activos
- El directorio `/proc` para conocer estadísticas del sistema como la memoria total, libre y en caché
- El directorio `/sys` para conocer características del hardware como la temperatura
- Comandos como `uptime` o `hostname` para conocer diversos parámetros del sistema.
- Herramientas como `awk`, `grep` o `cut` para obtener las cadenas de interés dentro del comando de respuesta.

Dichos comandos son ejecutados periódicamente mediante el gestor de eventos `ioloop` de **Tornado**.

La implementación del servicio está realizada íntegramente en Tornado[5], un servidor web ligero asíncrono implementado íntegramente en Python y mantenido por Facebook.

4.2.2.2. Deployer

El **Deployer** es una herramienta concebida a partir de la necesidad observada entre los estudiantes de las asignaturas Sistemas Distribuidos y Arquitectura de Computadores (como se refleja en las diferentes evaluaciones[Citation needed] realizadas), de replicar de una forma sencilla un ejecutable entre los diferentes nodos que conformarán el sistema distribuido.

Actualmente la infraestructura cuenta con un servidor NFS que posibilita la disponibilidad de la información en varios nodos de forma sencilla, mediante la copia a uno de los directorios alojados en el servidor. Sin embargo, este enfoque presenta varios inconvenientes: en el aspecto técnico supone una gran cantidad de ancho de banda consumido de forma continua (debido a que todos los estudiantes utilizan la misma infraestructura y realizan un gran número de operaciones de lectura y escritura a estos directorios, ralentizando el funcionamiento general del sistema enormemente) y en el aspecto didáctico, fomenta un mal hábito, pues los estudiantes no conocen otra forma de realizar despliegues más allá de la copia utilizando una interfaz gráfica y accediendo físicamente al nodo (si bien esta situación se mitiga en la asignatura Sistemas Distribuidos, donde deben automatizar los despliegues). Además, es necesario disponer de acceso físico a cada uno de los nodos, o en su defecto, conocer sus direcciones de red para realizar un acceso remoto.

Con el objetivo de proporcionar una alternativa adecuada a las necesidades y problemas descritos, surge esta herramienta, que aprovecha la funcionalidad de **MarcoPolo** para realizar su cometido.

La herramienta permite realizar las siguientes tareas de forma sencilla:

- Conocer todos los nodos disponibles sobre los que se podrá realizar el despliegue y seleccionar sobre cuáles de ellos trabajar.
- Permitir la copia a dichos nodos.
- Posibilitar la ejecución de comandos de forma remota una vez que el despliegue ha sido realizado.
- Facilitar la integración con contenedores de servicios, tales como **Apache Tomcat**.

[5] Más información sobre el proyecto puede encontrarse en tornadoweb.org/en/stable

La aplicación es accesible a través de un panel web . La interfaz web permite además conocer el estado de cada nodo en tiempo real, funcionalidad que a través de la línea de órdenes está disponible a través de los comandos

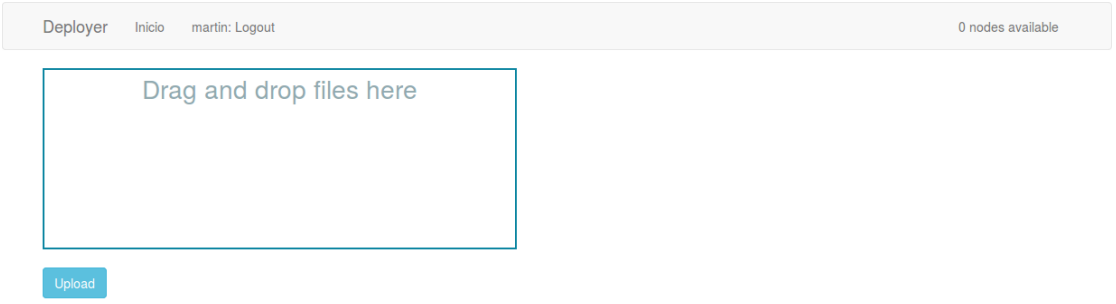

FIGURA 4.8: Interfaz web del deployer. A la izquierda figuran los controles y a la derecha la lista de nodos sobre los que se puede realizar el despliegue

Al igual que en el caso de la aplicación **statusmonitor** el **deployer** está creado utilizando el servidor web **Tornado** y todo el contenido enviado al usuario se reduce a archivos HTML, CSS y JavaScript. La comunicación entre el cliente y el servidor se realiza a través de peticiones *AJAX* y *Websockets*. Todo el control de la interfaz se delega a hojas de estilo CSS y JavaScript utilizando la biblioteca jQuery[6].

Autenticación La autenticación de los usuarios se realiza mediante el módulo **PAM** presente en cada nodo[Citation needed], utilizando **python-pam** para el acceso al mismo desde **Python**[16].

4.2.3. Pruebas de concepto

[6] jquery.com/

Capítulo 5

Arquitectura

Sumada a las herramientas creadas en el sistema, es necesario llevar a cabo una serie de operaciones que posibiliten el acceso a servicios más básicos tales como la autenticación de los usuarios del sistema,

Con el objetivo de mejorar la situación actual en la infraestructura a analizar, se tratan los siguientes problemas.

5.1. Instalación del sistema

El sistema a crear requiere de la instalación de diferentes componentes, en particular el sistema operativo, antes de poder ser utilizado. Dicha instalación, si es realizada en cada nodo secuencialmente, implica una gran carga de trabajo y aumenta la propensión a errores durante dicho proceso (en particular si en el mismo existe una gran carga de trabajo que debe ser supervisado por un administrador humano). Una solución a este problema es la autoinstalación del sistema operativo partiendo de una imagen definida y probada por el administrador, que se cargará e instalará en cada nodo sin supervisión.

Una de las herramientas ya existentes para solucionar este problema es el **PXE** (*Preboox eXecution Environment*)[17], un estándar *de facto*[18] para la carga de un sistema operativo desde un servidor. El estándar se apoya en protocolos presentes en la práctica totalidad de sistemas, tales como **DHCP**, **TFTP** y **TCP/IP**. El descubrimiento de servicios se realiza mediante una extensión en el mensaje **DHCPDISCOVER** que envía el servicio **DHCP** en su secuencia de arranque[19]. El servidor **DHCP**, si implementa esta extensión del protocolo, enviará la información sobre la localización de cada uno de los servidores de arranque al cliente, que procederá a la descarga utilizando el protocolo **TFTP** y posterior instalación[20].

Sin embargo, el uso de este protocolo requiere un controlador de interfaz de red (**NIC**) en el cliente que soporte el protocolo **PXE**. Generalmente dicho controlador se incluye como extensión de la **BIOS** o en equipos más modernos como código **UEFI**. La **Raspberry Pi** carece de este tipo de *software*, pues delega todo el arranque del sistema a los datos presentes en la tarjeta SD, y por tanto no es posible realizar ningún tipo de arranque en red sin la previa instalación de un conjunto de aplicaciones que realicen la descarga del sistema operativo. Es por ello que el uso de **PXE** como herramienta de arranque debe ser desestimado.

5.1.1. marco-netinst

Debido a la falta de soporte para **PXE** u otra alternativa similar, es necesario crear una herramienta que se encargue de la detección de un servidor que aloje la imagen del sistema operativo, la descarga del mismo y su instalación. Con este objetivo se crea la herramienta **marco-netinst**.

marco-netinst es una ramificación del proyecto **rasbpian-ua-netinst**[21]. Esta utilidad permite instalar un conjunto mínimo de utilidades que posibilitan la descarga de un sistema operativo desde los repositorios de **Debian** y su instalación. La ramificación incluye las siguientes modificaciones:

- Instalación de **ArchLinux ARM** en lugar de **Raspbian**.
- Instalación del sistema operativo completo a partir de un archivo **.tar.gz** en lugar de la descarga de paquetes[1].
- Nuevo *script* de carga del *software* en la tarjeta SD (en el paquete original se delega a utilidades de terceros).
- Detección del servidor sin configuración previa utilizando **MarcoPolo**.

La especificación en detalle del funcionamiento de la herramienta se detalla en

5.2. Autenticación de los usuarios

Los usuarios del sistema deben ser capaces de acceder al sistema mediante un sistema de credenciales que posibilite el uso de cualquier nodo del sistema con el mismo conjunto

[1] **raspbian-ua-netinst** utiliza el paquete cdebootstrap-static para la descarga e instalación de todos los archivos. Existe una herramienta para ArchLinux similar, denominada **Archbootstrap**
https://wiki.archlinux.org/index.php/Archbootstrap
https://packages.debian.org/sid/cdebootstrap-static

de claves. Dicho enfoque es el propio de la infraestructura actual del sistema, que en concreto sigue un enfoque centralizado.

Un primer intento de posibilitar la "universalización" del acceso ha sido la creación de los mismos usuarios en cada uno de los nodos, utilizando el mismo par usuario-contraseña en cada uno de ellos. Sin embargo, este enfoque impide una escalabilidad sencilla y requiere un mantenimiento continuo (suponiendo que se añaden usuarios periódicamente). Por ello únicamente las pruebas iniciales de las plataformas que requieren acceso a la funcionalidad de autenticación han sido realizadas siguiendo este enfoque, pero siempre desacoplando al máximo el sistema de acceso del resto de la lógica del programa, con el objetivo de facilitar su reemplazo.

Habiendo descartado dicha estrategia, queda como alternativa más adecuada a las necesidades del sistema el uso de la infraestructura presente en el centro académico.

La infraestructura del centro comprende varios servicios que interactúan entre sí, siendo el pilar clave el servidor LDAP (*Lightweight Directory Access Protocol*)[Citation needed]. Dicho servidor almacena la información de todos los usuarios de la infraestructura y da acceso a cualquier equipo de varias de las aulas de la Facultad.

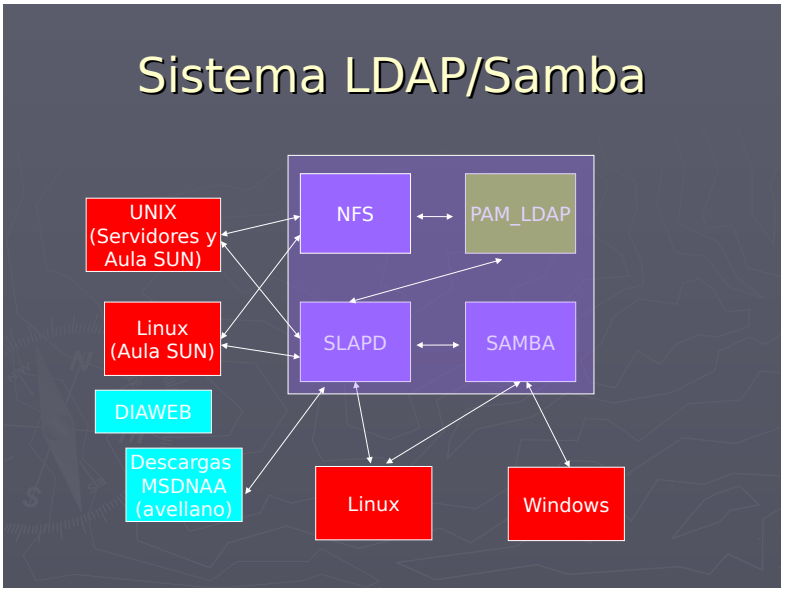

FIGURA 5.1: Esquema de los diferentes componentes del sistema de autenticación y gestión de archivos, así como de una serie de componentes adicionales. Obsérvese la interacción entre los componentes situados en el rectángulo interior

5.2.1. Características en detalle

Debido a la heterogeneidad de los diferentes equipos presentes en la infraestructura, el sistema debe posibilitar el acceso a todos los equipos utilizando el mismo conjunto de

credenciales. Esto implica que el sistema debe ser compatible con al menos los sistemas operativos GNU/Linux, Microsoft Windows y Solaris. Por ello se interconecta el servidor LDAP con Samba, así como el PAM (*Pluggable Authentication Module*) tanto en el cliente como el servidor.

Sin embargo el sistema permite también que los usuarios puedan almacenar información en un espacio centralizado al que es posible acceder desde cualquier equipo, facilitando la copia de ficheros entre nodos, uniformidad de los diferentes equipos. Esto se consigue utilizando un servidor NFS (*Network File Storage*).

5.2.2. Utilización en el sistema

En el sistema se aprovechará principalmente la funcionalidad de autenticación provista por el servidor LDAP, debido a que uno de los objetivos principales del sistema es evitar "cuellos de botella" debido al uso de un servidor de almacenamiento central. Herramientas como el **deployer** facilitarán la replicación de servicios en su lugar. En cualquier caso, se plantea permitir el acceso al NFS desde el sistema como complemento, pero no como espacio principal de almacenamiento.

El sistema aprovecha el módulo PAM para realizar el proceso de autenticación.

5.3. Compilación

Si bien el sistema Raspberry Pi es capaz de compilar el *software* que después utilizará, en ocasiones es beneficioso delegar dicha tarea a otro componente que realice el proceso por el nodo en cuestión y posteriormente añadir los archivos ejecutables al sistema. Este enfoque reduce el tiempo de trabajo de forma significativa, como observaremos posteriormente.

5.3.1. Creación de un compilador cruzado

Un compilador cruzado (*cross-compiler*)es una herramienta capaz de generar código para una architectura utilizando un equipo con otra arquitectura diferente.El uso de compiladores cruzados

Capítulo 6

Herramientas de terceros

Capítulo 7

Evaluación de usuarios

Bibliografía

[1] J. Kiepert, "Creating a Raspberry Pi-Based Beowulf Cluster," tech. rep., Boise State University, May 2013.

[2] J. Geerling, " Introducing the Dramble - Raspberry Pi 2 cluster running Drupal 8." http://www.midwesternmac.com/blogs/jeff-geerling/introducing-dramble-raspberry, Feb. 2015.

[3] GCHQ, "GCHQ's Raspberry Pi 'Bramble' - exploring the future of computing," *Big Bang Fair*, Feb. 2015.

[4] S. Cox, "Southampton engineers a Raspberry Pi Supercomputer ," Feb. 2011.

[5] U. de Salamanca, "Titulación y Programa Formativo - Grado en Ingeniería Informática." http://http://www.usal.es/webusal/files/Grado_en_Ingenieria_Informatica_2014_1%C2%AA%20parte-actualizado%202-10-14.pdf, 10 2014.

[6] N. W. Group, "Comment on RFC 4516 - Lightweight Directory Access Protocol (LDAP)," *RFC*, June 2006.

[7] B. Benchoff, " Benchmarking The Raspberry Pi 2." http://hackaday.com/2015/02/05/benchmarking-the-raspberry-pi-2/, Feb. 2015.

[8] T. Nishinaga, "Raspberry Pi 2 Linpack Benchmark ," Feb. 2015.

[9] ELinux.org, "RPi Performance," Aug. 2012.

[10] W. A. Team, "Windows 10 Coming to Raspberry Pi 2." Press Release, February 2015. http://blogs.windows.com/buildingapps/2015/02/02/windows-10-coming-to-raspberry-pi-2/.

[11] S. Cheshire and M. Krochmal", "Multicast DNS." RFC 6762 (Proposed Standard), Feb. 2013.

[12] F. H. S. Group, "Filesystem Hierarchy Standard," 2004.

[13] "netdb.h - definitions for network database operations." http://pubs.opengroup.org/onlinepubs/7908799/xns/netdb.h.html, 1997.

[14] I. Fette and A. Melnikov, "The WebSocket Protocol." RFC 6455 (Proposed Standard), Dec. 2011.

[15] A. Barth, "The Web Origin Concept." RFC 6454 (Proposed Standard), Dec. 2011.

[16] D. Ford, "python-pam 1.8.1," *Python Package Index*, Aug. 2014.

[17] I. Corporation and Systemsoft, "Preboot Execution Environment (PXE) Specification," *Preboot Execution Environment (PXE) Specification*, Sept. 1999.

[18] L. Avramov, *The Policy Driven Data Center with ACI: Architecture, Concepts, and Methodology*. Cisco Press, Dec. 2014.

[19] M. Johnston and S. Venaas, "Dynamic Host Configuration Protocol (DHCP) Options for the Intel Preboot eXecution Environment (PXE)." RFC 4578 (Informational), Nov. 2006.

[20] I. Corporation and Systemsoft, "Preboot Execution Environment (PXE) Specification," in *Preboot Execution Environment (PXE) Specification* [17].

[21] Rasbpian, "raspbian-ua-netinst." https://github.com/debian-pi/raspbian-ua-netinst, May 2015.

[22] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems: Concepts and Design*. USA: Addison-Wesley Publishing Company, 5th ed., 2011.

[23] G. S. Sidhu, R. F. Andrews, and A. B. Oppenheimer, *Inside AppleTalk*. Addison-Wesley Publishing Company, Inc., 2 ed., 1990.

There are 13 undefined references