UNIVERSIDAD DE SALAMANCA

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

por

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

12 de junio 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

por 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

Lista de abreviaturas

IEEE **I**nstitute of **E**lectrical and **E**lectronic **E**ngineers

NTP **N**etwork **T**ime **P**rotocol

Constantes físicas

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

Símbolos

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

For/Dedicated to/To my...

Capítulo 1

Introducción

Los límites físicos de los que adolecen los computadores en la actualidad[Citation needed: None] hacen de la computación distribuida un recurso para incrementar de forma sencilla y económica el rendimiento total de un sistema. El auge de sistemas como teléfonos inteligentes o aplicaciones web[Citation needed: https://vsis-www.informatik.uni-hamburg.de/getDoc.php/publications/432/activecomponents.pdf

Sin embargo, estas ganancias conllevan una serie de inconvenientes entre los que figura el aumento de la complejidad del sistema. En general, un sistema distribuido requiere un conjunto de entidades independientes, más difíciles de configurar y mantener que una única entidad. Además, aparecen nuevos problemas: comunicación, integridad y sincronización, dificultad en el desarrollo y depuración de aplicaciones, etcétera.

En el apartado didáctico, el estudio del paradigma distribuido suele requerir un gran esfuerzo por parte de los estudiantes, en particular a la hora de comprender los fundamentos básicos de cualquier aplicación distribuida así como a la hora de realizar tareas de análisis y depuración.

La presente memoria recoge el proceso de evaluación de diferentes alternativas para la creación de un sistema distribuido que satisfaga un conjunto de necesidades previamente establecidas, así como las diferentes etapas de diseño y desarrollo de un sistema formado por dispositivos Raspberry Pi como propuesta de solución y la creación de un conjunto de protocolos, herramientas y servicios para la utilización del mismo como utilidad 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.

Además, se incluyen las definiciones de los diferentes conceptos teóricos necesarios para la creación del sistema, así como las diferentes etapas de aprendizaje, evaluación de alternativas y diferentes procesos de evaluación llevados a cabo durante las diferentes etapas desarrollo, así como las metodologías de trabajo utilizadas, sin olvidar la documentación de todas las herramientas creadas.

Capítulo 2

Motivación

2.1. Objetivos

El sistema creado se inspira en proyectos similares y se diseña con el objetivo de dar solución a diferentes necesidades identificadas como estudiante de varias asignaturas del currículo del Grado en Ingeniería Informática. El sistema cuenta con cuatro objetivos a alto nivel independientes:

- Como síntesis de los conocimientos adquiridos en la carrera, se busca la creación de un sistema completo desde sus cimientos hasta los componentes de más alto nivel, gestionando las tareas de mantenimiento, instalación y manejo del mismo, así como los protocolos de trabajo, tanto en cada uno de los componentes del sistema como en la comunicación entre los mismos. Con un enfoque más teórico, se pretende crear un sistema capaz de poder ser utilizado como herramienta de diseño y prueba de algoritmos que resuelvan problemas aprovechando la distribución de tareas, así como el análisis de dichos algoritmos utilizando versiones finales del sistema.
- Potenciar su uso como herramienta de aprendizaje en las áreas de conocimiento Sistemas Operativos, Algoritmia, Redes de Computadores, Sistemas Distribuidos, Administración de Sistemas y Sistemas Embebidos.
- Constituir una herramienta didáctica para varias asignaturas del currículo del Grado en Ingeniería Informática de la Universidad de Salamanca, analizando aquellas relevantes y proponiendo soluciones a las diferentes necesidades propuestas por el Profesorado, Estudiantes y Administradores del sistema, en colaboración con dichas partes.

- Intentar elevar el *state of the art* en el mundo de los sistemas distribuidos con plataformas embebidas mediante la creación de un sistema multipropósito en lugar de soluciones con un fin determinado, que constituyen la tendencia actual.

2.2. Objetivos del proyecto

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

Partiendo de la premisa de las potenciales ventajas del uso de este tipo de computadores en detrimento de otras soluciones se plantea el sistema definitivo (en 4.5 se detalla el proceso de decisión), no sin antes realizar una etapa de evaluación de las diferentes alternativas.

2.3. Objetivos del sistema

Durante las fases de definición del proyecto, se plantean los siguientes objetivos concretos para la propuesta de solución elegida que se deben cumplir:

2.3.1. Diseño y construcción de la arquitectura física del sistema

Se deberán definir las interconexiones entre los diferentes componentes del sistema, solucionar los diferentes problemas físicos tales como la alimentación eléctrica, conexiones de red o la refrigeración, entre otros, analizando los diferentes enfoques y valorando la mejor solución en función del resto de objetivos a cumplir.

2.3.2. Arquitectura orientada a servicios

Conjunto de servicios que podrán ser aprovechados por diferentes clientes para explotar la capacidad de cálculo de las máquinas.

2.3.3. Gestión del sistema

El sistema debe contar con un conjunto de herramientas que mantengan los principios de transparencia propios de un sistema distribuido (ver 3.1), y su gestión debe ser sencilla para los responsables del mismo (personal de administración).

2.3.4. Integración

El sistema debe integrarse en una infraestructura preexistente, la presente en la Facultad de Ciencias de la Universidad de Salamanca, sin que dicha integración comprometa el diseño básico del sistema a fin de facilitar su adaptabilidad a otros entornos (ver 4.1.2). Es necesario por tanto realizar pruebas que evalúen el rendimiento del sistema creado en la misma.

2.3.5. Uso como herramienta didáctica

El sistema debe ofrecer una serie de ventajas a las herramientas didácticas utilizadas en aquellas asignaturas donde se impartan conocimientos relacionados con la computación paralela y distribuida, ofreciendo herramientas que faciliten la comprensión de dichos paradigmas o el desarrollo, prueba y aplicación de programas basados en los mismos.

2.3.6. Evaluación

A fin de probar los objetivos definidos anteriormente, la viabilidad de sistema como herramienta didáctica y su integración en la organización deberán ser determinados por los diferentes usuarios de la misma y la realización de pruebas de integración.

Durante el desarrollo del proyecto se añaden los siguientes objetivos funcionales:

2.3.7. Simplicidad de MarcoPolo

MarcoPolo debe conseguir un alto grado de versatilidad y aplicabilidad en un gran rango de aplicaciones. A fin de conseguir este objetivo, la simplicidad del sistema construido es clave. Esto conlleva el desacoplamiento y delegación de gran parte de la funcionalidad a otras capas superiores, independientes del protocolo, pero que aprovechan su funcionalidad, en lugar de ser integradas en el mismo (ver 8.1).

2.3.8. Test-Driven Development

El desarrollo de las diferentes herramientas *software* se deberá realizar bajo los principios del desarrollo conducido por pruebas (ver 3.7.1) como mecanismo para la detección temprana de errores.

2.4. Situación actual (*state of the art*)

En esta sección se definen diferentes enfoques ya aplicados a soluciones a problemas similares al planteado anteriormente.

2.4.1. Computadores de placa única

El uso de computadores de prestaciones reducidas como componentes de un sistema distribuido ha experimentado un gran crecimiento en los últimos años debido a la popularización y el abaratamiento de este tipo de dispositivos, existiendo gran cantidad de fabricantes y proveedores de *software* para los mismos.

Los computadores de placa única (*Single-Board Computers*) son máquinas de generalmente bajas prestaciones que aglutinan todos los componentes necesarios para su funcionamiento en un único circuito impreso. Suelen tener un coste bajo y una relación rendimiento/coste elevada. Su versatilidad y precio reducido han propiciado su uso como herramienta para el estudio y creación de sistemas distribuidos con un gran rango de propósitos diferentes.

2.4.1.1. 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 con el que su universidad cuenta[1] y sobre el que trabaja de forma rutinaria, con el objetivo de poder continuar su trabajo en periodos de mantenimiento, cierre del centro, etcétera. El sistema está diseñado para utilizar la *Message Passing Interface* como mecanismo de comunicación y coordinación (siguiendo un esquema maestro-esclavo) y además utilizar los diferentes puertos de las placas (GPIO, I^2C, SPI, UART), puertos generalmente ausentes en computadores convencionales. Utiliza además un sistema **NFS** (*Network File Storage*) para compartir datos entre todos los nodos, y un *router* dedicado para la interconexión. El sistema se completa con un ordenador portátil **Chromebook** con el mismo sistema operativo que los nodos del sistema, (**Arch Linux**), que actúa como nodo coordinador. La estructura incluye el conjunto de nodos esclavos y coordinador, dos fuentes de alimentación y un mecanismo de refrigeración, así como un mecanismo de distribución de la energía (diseñado por Kiepert) y de gestión de los diodos LED que incluye cada nodo y que son utilizados como elemento estético y mecanismo de análisis visual del comportamiento de los algoritmos ejecutados[1].

[1] Vídeo del sistema en ejecución: youtube.com/watch?v=i_r3z1jYHAc

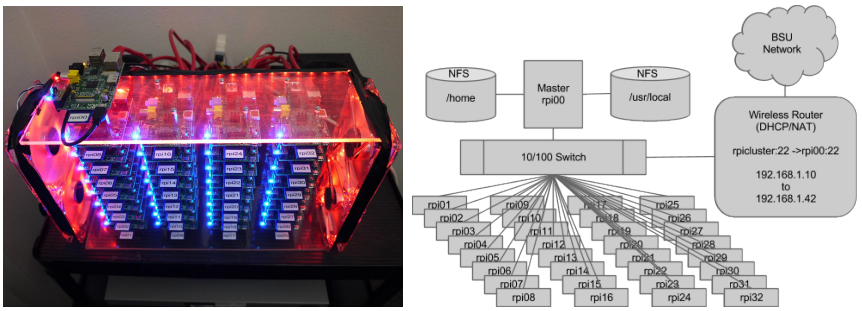


FIGURA 2.1: Vista general y estructura del sistema (Fuente: Joshua Kiepert)

El coste total del proyecto según Kiepert es de 1967.21 dólares.

2.4.1.2. Dramble (Jeff Geerling)

El clúster *Dramble* está formado por 6 equipos **Raspberry Pi** capaces de ejecutar en conjunto el gestor de contenidos **Drupal**[2]. Es utilizado como servidor de pruebas para la ejecución de instancias de este *software* de forma experimental o durante demostraciones en público[2]. Se compone del conjunto de nodos *Rasbperry Pi* y los mecanismos de red y alimentación que interconectan y proveen de energía a los mismos.

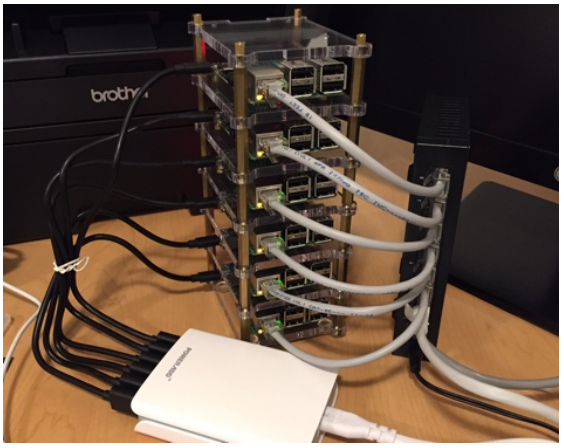


FIGURA 2.2: El *Dramble* en ejecución

El coste estimado es de 35 dólares por cada Raspberry Pi mas el coste añadido de la red y el cableado de alimentación, totalizando aproximadamente 300 dólares.

2.4.1.3. Bramble (GCHQ)

El organismo gubernamental *Government Communication Headquarters*, agencia de inteligencia del Gobierno Británico presentó en la *Big Bang Fair* de 2015 un proyecto

[2] drupal.org

educativo que combina 66 *Raspberry Pi* en un clúster jerárquico con 8 grupos de 8 nodos, cada uno de ellos con un coordinador. El cableado se reduce gracias al uso de la tecnología **PoE** (*Power over Ethernet*), y cada **Raspberry** cuenta con un conjunto de elementos adicionales, como un reloj de tiempo real, disco duro externo, cámara, o punto de acceso WiFi[3].

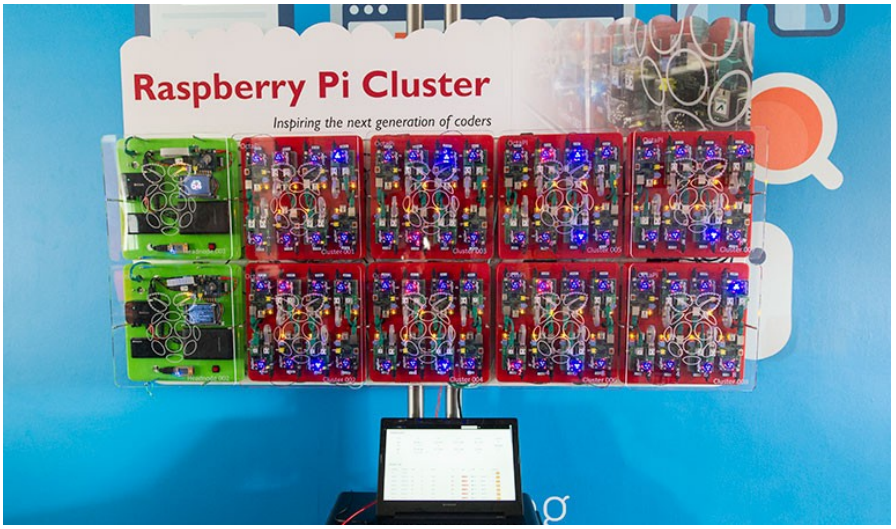


FIGURA 2.3: Vistazo general de la estructura del sistema Bramble

Se desconocen datos sobre el coste total del sistema.

2.4.1.4. 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 construida con LEGO[4]. El sistema está diseñado para ejecutar aplicaciones sobre *MPI*. Se desconocen datos sobre el coste total del sistema.

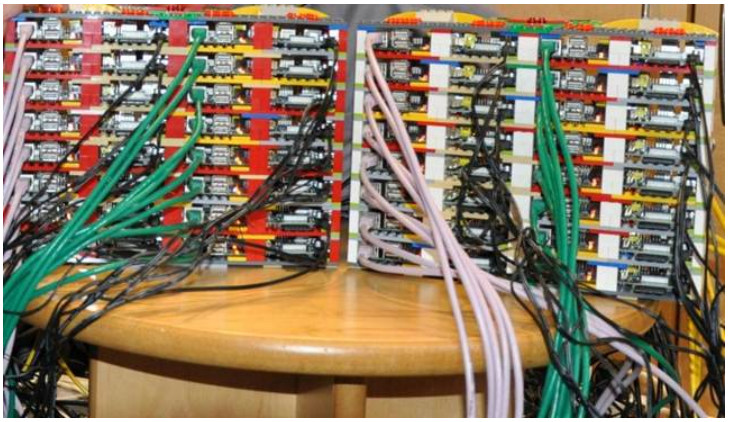


FIGURA 2.4: Clúster Iridis

2.4.1.5. Paralella

Paralella es un proyecto de la compañía Adapteva que integra en un único chip un conjunto elevado de procesadores independientes con el objetivo de incrementar la capacidad de procesamiento total del sistema a un coste muy reducido[5]. El coste es de 99 dólares por unidad.

2.4.2. Virtualización

Uno de los mecanismos para crear sistemas distribuidos en auge es la utilización de mecanismos de virtualización (ver 3.8.1, que evitan el uso de diferentes unidades de *hardware*. Estas soluciones se han popularizado en los últimos años principalmente en entornos empresariales, existiendo gran cantidad de proveedores de servicios y herramientas para la creación de un sistema propio (ver 3.8.1). Ejemplos de este tipo de proveedores son **Amazon Web Services**[3], **Google App Engine**[4], **Microsoft Azure**[5] o **Digital Ocean**[6], entre muchos otros. Su éxito reside en su gran versatilidad: es sencillo crear y destruir nuevas réplicas de un sistema bajo demanda, ahorrando costes de forma significativa.

2.4.3. *Commercial Off-The-Shelf hardware*

Este tipo de hardware está constituido por equipos disponibles al público de forma inmediata (*off the shelf*) y generalmente son máquinas de propósito general, las cuales son interconectadas para crear un sistema distribuido que sirva de alternativa a utilidades más potentes, pero de coste superior (como un *mainframe* o un ordenador de mayor potencia). Su coste económico (que se reduce si existe la posibilidad de aprovechar *hardware* existente en la organización, como equipos de escritorio que no están en uso) es su mayor atractivo.

Un ejemplo de este tipo de sistemas son los clústeres *Beowulf*[6], que se construyen sobre una red de área local y un sistema de intercomunicación como **MPI**, **PVM** u **OpenMP**. Existe gran cantidad de documentación para la creación de un clúster de este tipo[7], así como una serie de recursos (sistemas operativos, herramientas...) diseñadas con el propósito específico de crear este tipo de sistemas.

[3] https://aws.amazon.com
[4] https://cloud.google.com/appengine/
[5] http://azure.microsoft.com/en-us/
[6] http://digitalocean.com
[7] tldp.org/HOWTO/Beowulf-HOWTO/

Capítulo 3

Conceptos teóricos

Es necesario conocer una serie de conceptos clave antes de pasar a los diferentes aspectos de análisis y desarrollo del sistema. Se incluye además en cada apartado una sección que indica la aplicación final de cada uno de los conceptos teóricos en el sistema final.

3.1. Computación distribuida

Un sistema distribuido es aquel conformado por un conjunto de nodos independientes que son percibidos como una entidad única y coherente por el usuario final. Dicha definición implica dos conceptos de importancia:

- **Autonomía**: los diferentes integrantes cuentan con un alto grado de autonomía entre sí, y por tanto se deben diseñar e implementar mecanismos de comunicación entre los mismos que formarán parte del núcleo del sistema, y cuyo diseño tendrá consecuencias directas en el funcionamiento final del mismo.
- **Transparencia**: Las diferencias entre diferentes nodos deben ser invisibles para los usuarios finales, así como la gestión de fallos y la posterior recuperación, así como la uniformidad a la hora de interactuar con el sistema. Según [Citation needed: None]las siguientes propiedades deben contar con un grado de transparencia alto:[1]:

  Acceso: La forma de almacenamiento y gestión de los recursos e información presentes en el sistema debe ser completamente transparente.

  Localización: La localización física del sistema no debe ser de relevancia para el uso del mismo.

[1] En numerosas ocasiones las propiedades de un sistema hacen desfavorable el cumplimiento de todos los requisitos de transparencia. Un ejemplo sería la transparencia de traslado en el sistema DNS.

Migración: El cambio de plataforma debe ser transparente para el usuario (ejemplo: cambio del sistema operativo).

Traslado: El hecho de que un sistema se esté trasladando de un lugar a otro no debe afectar al usuario final.

Replicación: El numero de elementos redundantes en un sistema es desconocido para el usuario final.

Concurrencia: Un usuario no debe percibir la presencia de otros agentes interactuando con el sistema.

Gestión de errores: En caso de fallo, el usuario final no debe percibir el mismo, ni el proceso de recuperación consecuente (ver 3.1.10.

Generalmente los sistemas distribuidos son fácilmente escalables gracias a la autonomía de cada nodo, y los mecanismos de transparencia permiten crear sistemas heterogéneos de forma sencilla, en ocasiones apoyados en capas *middleware* (ver 3.1.3) que posibilitan dicha transparencia.

Otro concepto importante en el desarrollo de sistemas distribuidos es la "franqueza" (*openness*) de los mismos. Un sistema ofrece una serie de servicios gracias al uso de un conjunto de reglas conocidas por todos los participantes, generalmente recogidas en estándares de acceso público que definen la sintaxis y semántica de los servicios, conocidos como lenguajes de especificación de interfaz (*Interface Definition Language*), tales como CORBA[7] o el *IDL specification language*[8] (ver 3.1.3). Si dicho lenguaje es definido de forma apropiada, es posible crear diferentes implementaciones del mismo que sean capaces de comunicarse entre sí, incluso ejecutándose sobre máquinas completamente diferentes.

3.1.1. Escalabilidad y flexibilidad

La escalabilidad del sistema se ve afectada por las decisiones de diseño llevadas a cabo. En arquitecturas centralizadas el punto principal del sistema constituye un "cuello de botella" evidente que define un límite en el crecimiento del sistema. La disposición geográfica exige una serie de consideraciones adicionales, entre las que se encuentra la latencia del sistema. En arquitecturas constituidas por componentes relativamente pequeños y fácilmente adaptables la flexibilidad del sistema se incrementa significativamente, facilitando la escalabilidad del mismo. A fin de conseguir dicha flexibilidad es necesario desarrollar interfaces definidas para los componentes de bajo nivel del sistema, así como una descripción de las interacciones entre los diferentes componentes.

3.1.2. Algoritmos distribuidos

Un algoritmo distribuido es aquel que realiza una tarea de forma distribuida, cumpliendo el siguiente conjunto de propiedades:

- Ningún componente conoce el total de la información sobre el estado del sistema (**principio de autonomía**).
- Un componente únicamente puede tomar decisiones basadas en su conocimiento local (**principio de autonomía**).
- El fallo de un nodo no provoca el fallo del sistema (**principio de transparencia**).
- No hay una asunción implícita de que existe un reloj global (**principio de autonomía**).

3.1.3. *Middleware*

Un *middleware* es una capa *software* que permite abstraer las peculiaridades de un estrato subyacente (como una interfaz de comunicación en red, funcionalidad del sistema operativo, mecanismos de acceso a bases de datos...) en una interfaz común e independiente. Ejemplos de middleware son la *Common Object Request Broker Architecture* (**CORBA**), llamadas a procedimientos y métodos remotos (**RPC**, **RMI**) o herramientas de serialización. Sin embargo, el término también se aplica a cualquier otro tipo de capa intermedia que proporciona una abstracción entre componentes.

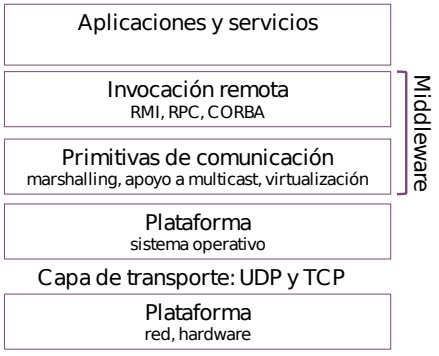


FIGURA 3.1: Ejemplo de una arquitectura que utiliza varias capas *middleware*.

3.1.3.1. Utilización en el sistema final

Los diferentes *bindings* de MarcoPolo (ver **??**) pueden ser considerados middleware, pues abstraen los diferentes procesos de comunicación que ocurren al utilizar el sistema, proporcionando una interfaz independiente del lenguaje de programación y el sistema operativo. La serialización de las peticiones puede ser considerada *middleware*.

Si **nsswitch** (ver **??**) es considerado un sistema que abstrae los diferentes mecanismos de autenticación y resolución de nombres de la implementación de los mismos puede ser considerado un *middleware*. **Nsswitch** se utiliza como método de abstracción de los diferentes proveedores de información de usuarios (archivos del sistema y **LDAP**) en el sistema final.

3.1.4. Modelos arquitectónicos

Existe un gran rango de diferentes modelos de construcción de sistemas distribuidos en función de las necesidades a cubrir por el sistema.

3.1.4.1. Clúster

En general se conoce como clúster al conjunto de nodos homogéneos y dispuestos físicamente en la misma localización, conectados entre sí mediante mecanismos fiables como redes de área local y que cuentan con el mismo conjunto de herramientas (en particular el sistema operativo). Generalmente estos sistemas se componen de nodos de bajo coste, como equipos **COTS** y son utilizados para la realización de una única tarea con un coste computacional alto en paralelo.

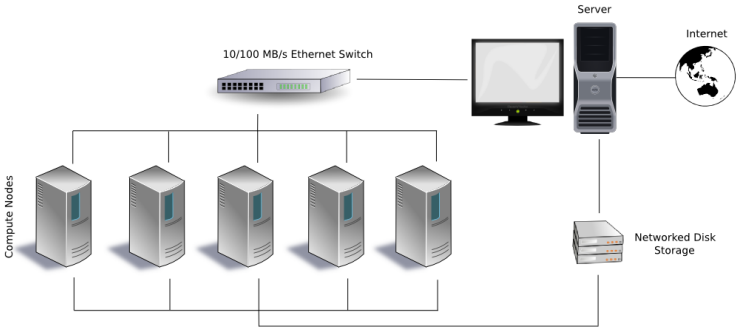


FIGURA 3.2: Esquema de un clúster Beowulf. Este tipo de clústeres permiten obtener una gran capacidad de cómputo paralelo con equipos de bajo coste, y generalmente se comportan como una única unidad.

3.1.4.2. *Grid*

Una "rejilla" (grid) es un sistema distribuido en el que los nodos del sistema no son homogéneos. Los mecanismos de transparencia descritos anteriormente son clave para el correcto funcionamiento del sistema y las interconexión entre los diferentes elementos.

3.1.4.3. Sistemas de información distribuida

[Citation needed: None]

3.1.4.4. Sistemas descentralizados

Uno de los principales problemas de las arquitecturas propuestas es la dependencia de un nodo que actúe de coordinador del resto. Dicha dependencia dificulta o incluso impide el funcionamiento del conjunto de nodos en caso de que el coordinador no cumpla adecuadamente su cometido (debido a fallos en el mismo, dificultades en la comunicación, etcétera). Las arquitecturas *peer-to-peer* (de igual a igual) proponen una solución a dicho problema. Este tipo de enfoques flexibilizan la escalabilidad y tolerancia a fallos del sistema, si bien incorpora una serie de desafíos adicionales. La falta de coordinador implica que los diferentes nodos deben conocerse unos a otros previamente, o de algún modo descubrir la presencia del resto antes de poder cooperar. La centralización de dicho

proceso es una de las soluciones propuestas para facilitar la construcción de la red de *peers*[Citation needed: None].

Una de las ventajas de este tipo de sistemas es la facilidad para el establecimiento de redundancias, tanto de datos como de servicios, así como la alta tolerancia a fallos (el fallo de un nodo no impide que el resto pueda continuar su cometido a menos que cuente con una serie de recursos exclusivos).

3.1.4.5. Utilización en el sistema final

El sistema final es de tipo descentralizado, únicamente existiendo un coordinador en aquellos servicios en los que sea necesaria la presencia de uno (todos los nodos son capaces de actuar en ambos roles, y la elección del coordinador se realiza de forma distribuida, por lo que el fallo de un coordinador no supone un fallo total del sistema, y la recuperación es relativamente sencilla). Se utiliza el protocolo **MarcoPolo** (ver 8.1) para el descubrimiento de nodos y servicios, por lo que la configuración necesaria en el sistema es mínima.

3.1.5. Seguridad

Las medidas de seguridad a tomar dependen de las propiedades y objetivos propios del sistema: en general sistemas utilizados dentro de una organización y una infraestructura sin interacción con entornos no controlados suelen contar con un número menor de medidas de seguridad que aquellos utilizados en entornos "hostiles" (generalmente abiertos al público), y la seguridad depende de la confianza depositada en la administración del sistema. Sin embargo, un sistema de este tipo es vulnerable a ataques internos por parte de usuarios o intrusiones en la infraestructura.

Un sistema integrado en un entorno hostil debe además vigilar cualquier tipo de potencial ataque malicioso y controlar el acceso al sistema de forma más minuciosa.

3.1.6. Protocolo

Un protocolo consiste en un conjunto de formatos y reglas bien definidas y conocidas que son utilizadas para posibilitar la comunicación entre un conjunto de entidades. Un protocolo consta de dos especificaciones: la secuencia de mensajes que deben ser intercambiados en cada una de las diferentes acciones y la especificación del contenido y formato de dichos mensajes.

Si un protocolo es definido de forma correcta posibilitará la comunicación entre entidades sin importar la implementación del mismo, ofreciendo un alto grado de transferencia. De esta forma es posible, por ejemplo, ofrecer un servicio web que procese una serie de datos numéricos sin importar el tipo de *endianness* del cliente o el servidor o el lenguaje de programación en el que el servicio se implementa. Únicamente es necesario conocer el tipo de dato que requiere el servicio y el valor esperado de retorno, así como los mensajes a intercambiar.

3.1.6.1. Utilización en el sistema final

En el sistema se utilizan gran cantidad de protocolos conocidos (tales como HTTP, JSON, Websockets...) y se ha definido el protocolo de descubrimiento de servicios **MarcoPolo**.

3.1.7. Integridad

3.1.8. Comunicación

3.1.9. Distribución

Uno de los modelos típicos en el desarrollo de sistemas distribuidos es el "divide y vencerás": la división de un problema en múltiples tareas y la distribución de las mismas entre los diferentes componentes del sistema, reagrupando los resultados posteriormente. Dicho paradigma no se aplica únicamente a tarea a realizar, sino también al conjunto de datos sobre el que realizarla, paralelizando una misma operación en diferentes nodos sobre fragmentos del conjunto de datos sobre el que operar, recopilando los datos devueltos y conformando la respuesta final.

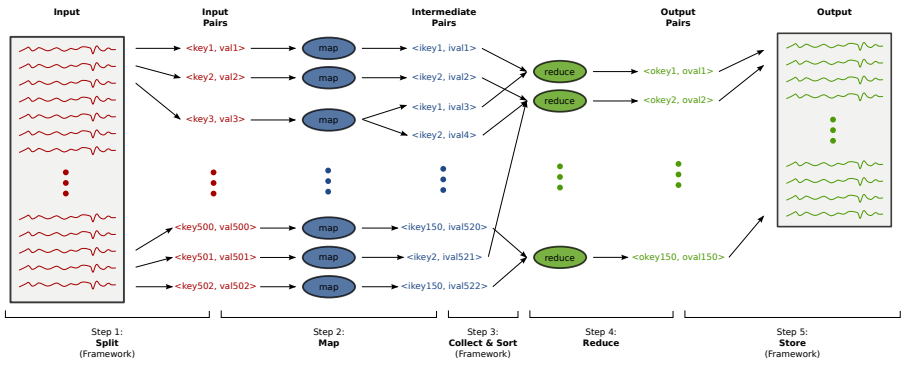


FIGURA 3.3: **MapReduce** se basa en el principio de la distribución de los datos sobre diferentes nodos (*Map*) y la recopilación de los datos procesados (*Reduce*)

3.1.10. Autoadministración

Un sistema distribuido debe proporcionar mecanismos que resuelvan de forma transparente todos los problemas asociados con la gestión de los mecanismos de interconexión entre los diferentes integrantes. Dichos mecanismos están diseñados para ser adaptativos y si bien el diseño suele ser genérico, es habitual encontrar soluciones *ad-hoc*.

3.1.10.1. Utilización en el sistema final

3.2. Paradigmas de programación

3.2.1. Programación orientada a eventos

La programación orientada a eventos (*event-driven programming*) constituye un patrón de programación en el cual el flujo del programa no se define íntegramente de forma secuencial, sino por las diversas interacciones (eventos) que el *software* recibe durante su ejecución. Dichos eventos, generalmente impredecibles, son causados por cualquier otra aplicación o entidad externa, y su naturaleza es muy variada. Interacciones de ratón o teclado, conexiones de red o estímulos de sensores son claros ejemplos, pero también son eventos notificaciones de finalización de un trabajo o llamadas periódicas.

La programación orientada a eventos es un patrón exitoso en varios tipos de aplicaciones, en particular aquellas con interfaz gráfica de usuario, donde la interacción es completamente imprevisible o aplicaciones de un único hilo (*single-threaded applications*), que utilizan los eventos para establecer mecanismos de coordinación.

Si bien existen una gran cantidad de herramientas para el desarrollo de *software* siguiendo este patrón, el funcionamiento de todas ellas se reduce a un conjunto de manejadores y disparadores de eventos. También reciben nombres similares, como señales y ranuras.

3.2.1.1. Suscriptor de eventos

Un suscriptor de eventos permite vincular un evento a un manejador (*handler*), que determina la acción a realizar al recibir un evento. En una gran cantidad de *frameworks* se definen como funciones. Por ejemplo:

```
function manejadorMouseMove(evento){
  console.log(evento.pageX);
}
document.onmousemove=manejadorMouseMove
```

FIGURA 3.4: El evento "onmousemove", que se dispara en el momento en el que el ratón se desplaza, es vinculado a la función `manejadorMouseMove` en una página web.

Ejemplos de herramientas que utilicen este paradigma son **Node.js**, **Tornado web**, **Twisted** o la gran mayoría de los *frameworks* para creación de interfaces gráficas, como **Cocoa**, **Windows Presentation Framework** o **Qt**, entre otros.

En el proyecto se utiliza este patrón para la creación de la mayoría de las herramientas, en particular en combinación con el modelo de aplicaciones de un único hilo (ver 3.2.2).

3.2.2. Paralelización, hilos y procesos

Uno de los mecanismos para conseguir paralelizar tareas es su distribución en procesos independientes. Cada proceso cuenta con un espacio de memoria y asignación de recursos por parte del sistema operativo (CPU, descriptores de fichero abiertos...) completamente independiente. Sin embargo, dicha independencia implica la reserva de un segmento de memoria y el almacenamiento de los valores de ejecución (contador de programa, valores de registros, puntero de pila, etcétera) cuando la CPU debe realizar un cambio de contexto para permitir la ejecución de otro proceso. Dicha alternancia y la reserva de estos recursos implican un coste computacional en ocasiones alto, en particular cuando el número de cambios de contexto es elevado. Como solución se plantea el uso de hilos (*threads*), ejecuciones de fragmentos de código independientes del resto de hilos pero con un nivel de transparencia menor si el mantenimiento de la misma afecta al rendimiento del conjunto de hilos.

Un tercer enfoque es desarrollo de aplicaciones dirigidas por eventos (*event-driven*) en un único hilo (*single-threaded application*). Para conseguir paralelizar las diferentes tareas se deben ejecutar de forma no bloqueante, cediendo la CPU a otra tarea en caso de que se deba esperar a que otra sea realizada (generalmente, operaciones de entrada-salida, que presentan un gran tiempo de espera sin consumo de CPU, como la descarga de información en red o el acceso al almacenamiento secundario).

Mediante rellamadas (*callbacks*) una tarea que termina una operación de este tipo indica al hilo gestor que requiere de nuevo tiempo de computación, y este añadirá la tarea de nuevo a la cola de acceso a la CPU.

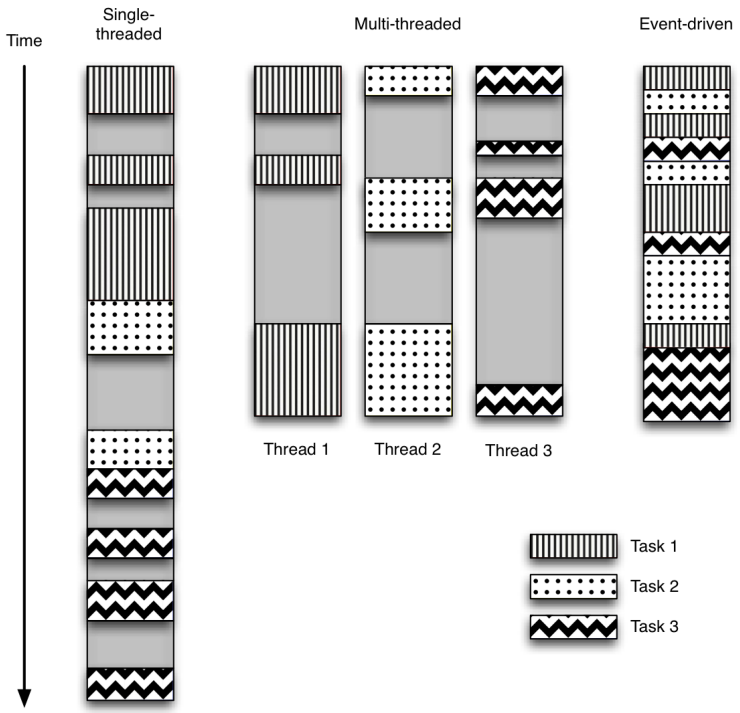


FIGURA 3.5: Comparación de los diferentes modelos de paralelización[Citation needed: Twisted book]

3.2.2.1. Patrón Reactor

El patrón de diseño reactor[9] consiste en un sistema de gestión de peticiones de servicio distribuidas de forma concurrente en un esquema cliente-servidor. En el servidor existe un manejador de eventos que distribuye (demultiplexa) los diferentes eventos a manejadores de los mismos, distribuyendo el tiempo de proceso entre ellos de forma síncrona mediante un bucle ejecutado de forma continua.

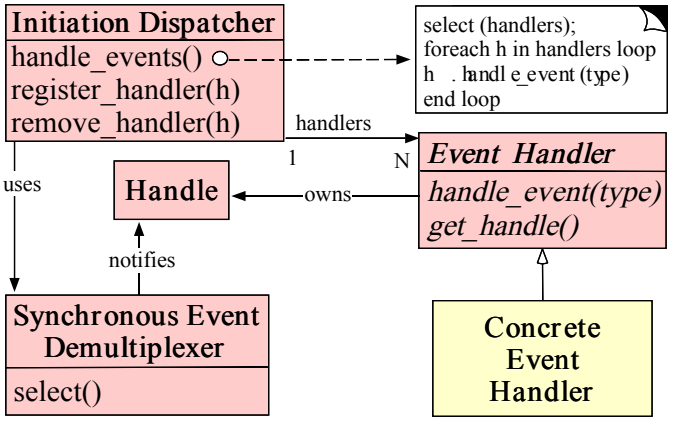


FIGURA 3.6: Diagrama UML del patrón reactor.

3.3. Sincronización

3.3.1. Mecanismos de coordinación

En determinadas ocasiones es necesaria la presencia de un coordinador que lleve a cabo una serie de tareas de gobierno. La elección de dicho coordinador puede atender a una serie de criterios muy variados y los mecanismos de designación siguen varios enfoques:

3.3.1.1. Coordinación central

Un nodo, o instancia de un programa es designado como coordinador, cuya autoridad es obedecida por el resto de integrantes del sistema. Es un enfoque sencillo de implementar que sin embargo crea un punto de fallo en el sistema, pues la caída del coordinador central impedirá la realización de cualquier tarea en la que sea necesaria su intervención hasta que vuelva a estar activo.

El coordinador puede designarse mediante diferentes mecanismos. En algoritmos como OSPF[10] la política de elección del *router* designado es aquel que cuente con el mayor valor de prioridad.

3.3.1.2. Coordinación distribuida

Con objeto de solventar los problemas que la coordinación centralizada conlleva se proponen algoritmos que permiten designar un coordinador y reemplazarlo en caso de fallo del mismo.

Algoritmo del abusón El algoritmo del abusón (descrito en [11]) posibilita la elección de un coordinador en función del cumplimiento de un criterio cuantitativo dado, siendo elegido coordinador aquel que mejor satisfaga dicho criterio. La secuencia de pasos es la siguiente:

Sea un proceso, P que detecta la ausencia de un coordinador (en caso de que ya se haya ejecutado el algoritmo, se detecta la ausencia del anterior coordinador) y X un valor numérico que determina el criterio para la elección del coordinador (el proceso con el valor más alto te X será el coordinador).

1. P envía un mensaje de elección a todos los procesos con X más alto que X^P.
2. Si ningún proceso responde en un tiempo dado, P envía un mensaje indicando que es el coordinador.

3. En caso de que algún proceso responda al mensaje, el proceso se repetirá con el conjunto de nodos con X mayor que X^P.

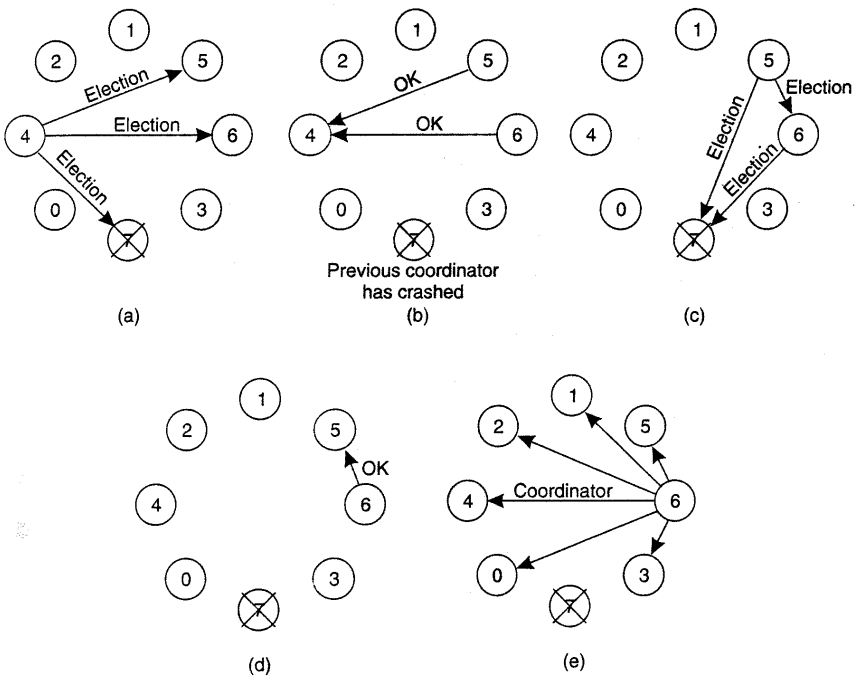


FIGURA 3.7: Secuencia de elección de un coordinador

Algoritmo en anillo El algoritmo funciona bajo la premisa de que todos los procesos cuentan con un orden preestablecido, conociendo a su antecesor y sucesor en la cadena[12]. El funcionamiento del algoritmo es el siguiente:

1. Cuando un proceso P detecta la caída de un coordinador, envía un mensaje de elección al proceso sucesor, o en su defecto, al proceso sucesor más próximo, en caso de que se encuentre inactivo, hasta que encuentre un proceso activo.

2. Cuando un proceso recibe el mensaje, añade al mismo su número de proceso.

3. Una vez que el mensaje llega al creador del mismo, este construye un mensaje con el identificador del proceso que ha sido elegido como coordinador (el proceso con el identificador más alto, o aquel que satisfaga otro criterio similar) y lo envía al siguiente proceso de manera similar al mensaje inicial, si bien el contenido del mensaje no es alterado en este caso.

4. Una vez que el mensaje ha circulado por todos los procesos, retorna a su creador y en este momento el trabajo se reanuda.

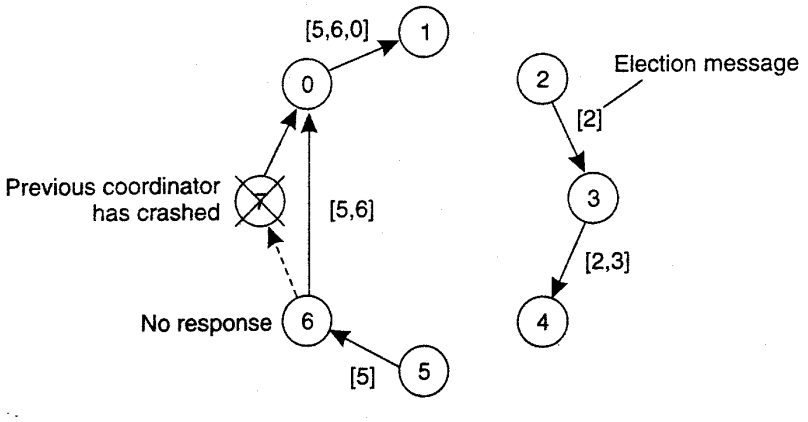


FIGURA 3.8: Funcionamiento del algoritmo en anillo

3.4. Protocolos criptográficos

3.4.1. Criptografía asimétrica

La criptografía asimétrica, o criptografía de clave pública se basa en algoritmos que requieren un par de claves en cada entidad, siendo una de ellas conocida únicamente por dicha entidad (clave privada) y la otra públicamente disponible. Dichos algoritmos se basan en problemas sin solución en un tiempo aceptable, como por ejemplo la factorización de enteros.

Ambas claves consisten en un valor numérico y están relacionadas matemáticamente entre sí. La clave pública es utilizada para realizar operaciones de cifrado de datos o verificación de una firma digital, mientras que la clave privada es utilizada para los procesos complementarios.

3.4.2. Infraestructura de clave pública

Una infraestructura de clave pública está formada por el conjunto de *hardware*, *software*, políticas y procedimientos necesario para la creación y gestión de certificados digitales, documentos electrónicos utilizados para probar la identidad de una determinada entidad. Dichas infraestructuras se componen de una autoridad (**CA**, *Certificate Authority*) que garantiza la identidad del solicitante de un certificado, y por tanto, es el responsable de verificar dicha identidad en el momento de creación del mismo. El resto de entidades confiarán en dicha **CA**. Estos certificados se componen de un par de claves pública y privada[Citation needed: None].

3.4.2.1. X.509

El estándar X.509 define una realización de una infraestructura de clave pública en la cual los certificados están vinculados a un nombre, sea este un nombre distinguido (siguiendo la estructura del protocolo LDAP, ver 3.5.2) o un nombre alternativo, como una dirección de correo electrónico o un nombre **DNS**. Según el estándar se define una cadena de validación que culmina en los denominados certificados raíz, a partir de los cuales son creados el resto de autoridades de certificación y finalmente, los certificados dados a los diferentes componentes que los utilizan.

La estructura de un certificado es la siguiente:

- Certificado.
  - Versión.
  - Número de serie.
  - Tipo de algoritmo.
  - Distribuidor.
  - Validez.
    - No válido hasta.
    - No válido después de.
  - Información de clave pública: Algoritmo y clave.
  - Identificador único del distribuidor (opcional).
  - Identificador único del solicitante (opcional).
  - Extensiones (opcional).
- Algoritmo de firma del certificado.
- Firma del certificado.

3.4.3. TLS/SSL

El protocolo *Transport Layer Security*[13] y su predecesor, *Sockets Security Layer* permiten establecer conexiones seguras sobre un canal inseguro (proclive a ataques pasivos como el uso de *sniffing*[Citation needed: None] o activos, como técnicas de *spoofing* o *man-in-the-middle*[Citation needed: None]) mediante el uso de una infraestructura de clave pública. Este protocolo se integra en los niveles de sesión y presentación del modelo OSI, y por lo tanto utilizan los mismos protocolos de transporte y red que una conexión tradicional,

por lo que es sencillo construir aplicaciones sobre canales seguros utilizando las mismas técnicas que las usadas en este tipo de comunicaciones, o incluso reutilizar código. El protocolo funciona según la siguiente secuencia:

1. El proceso con el rol de cliente comienza el proceso conocido como "apretón de manos" (*handshake*, solicitando una conexión segura junto con las diferentes funciones de cifrado admitidas).
2. El servidor elige una de las funciones propuestas y notifica la decisión al cliente.
3. El servidor envía su identificación en forma de certificado digital, que incluye campos como su *common name*, la autoridad de certificación y la clave pública.
4. El cliente valida la autoridad de certificación según la cadena de validación presente en el mismo, o en caso necesario, contactando a esta.
5. El cliente cifra un número aleatorio con la clave pública del servidor y se lo envía.
6. Mediante este número se genera la clave simétrica secreta ("secreto maestro") y negocian una clave de sesión para realizar los procesos de cifrado.
7. Comienza la transmisión de información.

En caso de que alguno de los pasos sea infructuoso, la conexión terminará.

3.4.4. HTTPS

El protocolo HTTPS[14] utiliza TLS para cifrar las conexiones realizadas mediante el protocolo HTTP. La combinación de ambos protocolos se realiza superponiendo TLS al protocolo HTTP, por lo que las cabeceras y datos transmitidos no son alterados, siendo por tanto compatibles entre sí. Este protocolo (o combinación de protocolos, al consistir en la simple combinación de varios) es utilizado en recursos web para garantizar la identidad del servidor que envía los datos y establecer un canal seguro sobre una red no segura.

3.4.5. Autenticación mutua

El uso más común de los protocolos previamente definidos es verificar la identidad del servidor al que se está conectando un cliente. Sin embargo, en ocasiones es necesario que el servidor confíe en el cliente, como en situaciones donde el cliente no dispone de otro mecanismo de autenticación, como puede ser un par usuario-contraseña. El *handshake*

es similar al utilizado tradicionalmente, únicamente se añade el paso de validación del certificado que el cliente debe enviar al servidor.

En el sistema este proceso de autenticación se utiliza en aquellas aplicaciones sin interfaz de usuario o que realizan operaciones que puedan comprometer la integridad del sistema de forma significativa (por ejemplo, solicitar la realización de operaciones con privilegios de administrador en **pam_mkpolohomedir** o la descarga de un sistema operativo que contiene todas las claves privadas, que deben ser confidenciales).

3.5. Autenticación

3.5.1. PAM

El *Linux Pluggable Authentication Module*[15] es un componente integrable en el mecanismo de autenticación del sistema operativo para combinar diferentes mecanismos de identificación que puedan ser aprovechados por aplicaciones de alto nivel, abstrayendo las características del sistema de autenticación presente en la máquina, proporcionando una interfaz única. **PAM** se apoya en diferentes fuentes de datos configurables, tales como los ficheros `/etc/passwd` y **/etc/shadow**, directorios **LDAP** (ver 3.5.2), autenticación mediante clave pública, (ver 3.5.2).

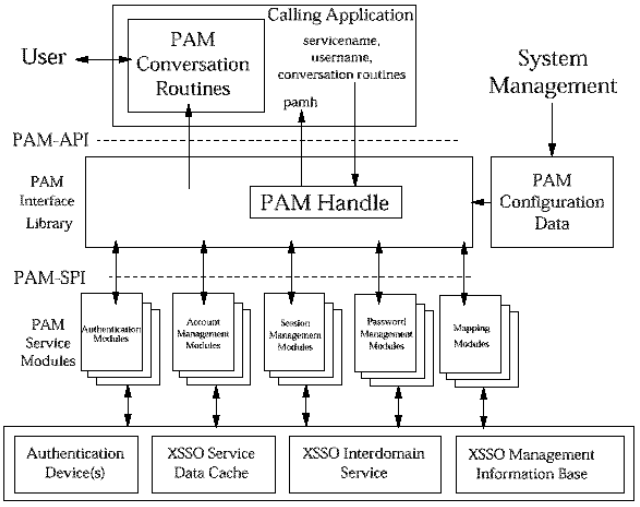


Figure 4-1 PAM Framework

FIGURA 3.9: Esquema de la arquitectura de PAM (fuente [15])

PAM está diseñado para definir su comportamiento mediante el uso de módulos, pequeñas piezas de código que definen las acciones a llevar a cabo dadas un conjunto de circunstancias, como el cambio de una contraseña, la modificación de un parámetro en

la información del usuario o el acceso al sistema de un usuario. Mediante los parámetros de configuración de **PAM** se definen aquellas circunstancias que requieren del uso del módulo definido.

```
service  module_type  control_flag  module_path          options
-------  -----------  ------------  -----------          -------
login    auth         required      pam_unix_auth.so     nowarn
login    session      required      pam_unix_session.so
login    account      required      pam_unix_account.so
ftp      auth         required      pam_skey_auth.so     debug
ftp      session      required      pam_unix_session.so
telnet   session      required      pam_unix_session.so
login    password     required      pam_unix_passwd.so
passwd   password     required      pam_unix_passwd.so
OTHER    auth         required      pam_unix_auth.so
OTHER    session      required      pam_unix_session.so
OTHER    account      required      pam_unix_account.so
```

FIGURA 3.10: Un fichero de configuración de PAM con diferentes módulos definidos para el acceso al sistema mediante un terminal (*login*), FTP, Kerberos o telnet, así como acciones a realizar cuando una entrada del fichero `/etc/passwd` es modificada (*passwd*)

.

3.5.2. LDAP

El protocolo *Lightweight Directory Access Protocol* posibilita el acceso a directorios de información en un entorno distribuido siguiendo un esquema cliente-servidor.

3.5.2.1. Utilización en el sistema final

La gestión de usuarios está delegada al módulo PAM, utilizando los mecanismos de autenticación que provee en las diferentes aplicaciones del sistema, como en **Deployer**.

Con el objetivo de garantizar diversas propiedades de transparencia del sistema, se ha creado un módulo de PAM que se integra con **MarcoPolo** para la realización de tareas de gestión en el conjunto de nodos del sistema (ver 8.2.3.1).

3.6. Comunicación

La comunicación entre procesos dentro de un sistema distribuido constituye un aspecto clave en el diseño del mismo, y existen una gran cantidad de modelos para posibilitarlo.

3.6.1. Multicasting

Si bien las comunicaciones punto a punto consituyen un mecanismo efectivo para la comunicación entre diferentes nodos, presentan una serie de ineficiencias a la hora de distribuir el mismo contenido entre un número significativo de nodos, al tener que enviar un mensaje diferente a cada uno de los destinatarios. Utilizar difusión (*broadcast*) para este tipo de aplicaciones supone una alternativa efectiva, si bien obliga a las entidades no interesadas en el mensaje a procesar el mismo en ocasiones hasta el nivel de red de la pila OSI [Citation needed: None].

Como alternativa surge la comunicación *multicast* sobre el protocolo IP. Este tipo de mensajes son enviados a una dirección en un rango reservado (en IPv4, 224.0.0.0-239.255.255.255, las direcciones con el valor 1110 en el primer octeto, conocidas como Clase D[16]), conocido como *grupo*. Los nodos interesados en los mensajes publicados en dicho grupo pueden suscribirse al grupo multicast y cancelar su membresía en cualquier momento dado. Todos los mensajes son de tipo UDP, si bien existen extensiones al protocolo TCP para posibilitar su uso en aplicaciones multicast, si bien de forma experimental[17–21] o soluciones que permiten realizar multicast fiable (con diferentes grados de fiabilidad) sobre UDP[22].

3.6.2. Serialización

Con el objetivo de posibilitar la intercomunicación entre entidades es necesario determinar mecanismos para la representación de estructuras de datos complejas de forma homogénea, definiendo una serie de reglas para la representación de datos sin importar la plataforma de cada una de las diferentes entidades partícipes.

Existen una serie de formatos definidos que posibilitan este tipo de interconexión. Uno de ellos es **JSON** (*JavaScript Object Notation*)[23]. **JSON** define estructuras de datos complejas, tales como objetos, listas o diccionarios clave-valor, mediante un subconjunto de la sintaxis de **JavaScript** que permite representar cualquier tipo de dato de forma legible por personas y fácilmente comprensible por máquinas. Existen implementaciones de procesadores de JSON en la mayoría de lenguajes de programación, lo cual permite el intercambio de información entre diferentes programas de forma sencilla.

3.6.2.1. Utilización en el sistema final

Toda comunicación entre dos entidades se realiza gracias a cadenas con formato **JSON** (salvo casos concretos, como los programas ejecutados sobre MPI, donde el entorno proporciona un mecanismo claramente más efectivo de comunicación) utilizando bibliotecas de terceros para el procesamiento de las mismas.

3.6.3. WebSockets

3.7. Modelos de desarrollo

3.7.1. Test-Driven Development

El desarrollo dirigido por pruebas es una estrategia de desarrollo de software basada en el desarrollo de tests de partes concisas y fácilmente desacoplables del código de una aplicación que realizan una tarea muy concreta (conocidas como pruebas unitarias). Dichas pruebas se realizan generalmente con un control exhaustivo de las condiciones que pueden influir el desarrollo de la prueba. Mediante la modificación de dichas condiciones y el análisis de la respuesta frente a un conjunto de datos de entrada se puede analizar si el comportamiento de dicho fragmento de código es el adecuado.

Las pruebas consisten generalmente en una serie de aserciones sobre el resultado del fragmento de código ejecutado, o sobre un estado intermedio. Una batería de pruebas consistente debe cubrir el mayor número de resultados de salida posible, incluyendo valores que indiquen condiciones de error (verificando que dicho error deba ser devuelto para los parámetros de entrada y condiciones iniciales), excepciones y diferentes tipos de retorno válidos.

Un ciclo de desarrollo basado en pruebas consiste en dos fases: inicialmente la escritura de la batería de pruebas basadas en los diferentes requisitos del *software* y ejecución de dicha batería, que deberá ser completamente infructuosa. Tras esta fase comienza la etapa de refactorización, consistente en la implementación de la funcionalidad que el fragmento de código de cada test debe realizar y la ejecución de la batería de pruebas durante las diferentes etapas de implementación. Cuando un test se ejecute satisfactoriamente se garantizará que el código cumple los requisitos de la prueba definidos en la fase inicial, y por tanto, con los requisitos del *software*.

Esta práctica asiste al programador en el desarrollo de *software* más fiable, pues todas las potenciales condiciones de fallo que se reflejan en la batería de pruebas son cubiertas

durante la segunda fase del proceso (o en caso contrario el resultado test no será satisfactorio). Uno de los efectos secundarios del desarrollo dirigido por pruebas es la escritura de código de grano muy fino y muy desacoplado (al tener que desacoplar la funcionalidad para poder adaptarla a una prueba unitaria). El desarrollo dirigido por pruebas permite también controlar de forma sencilla las interdependencias entre diferentes módulos de código, pues basta con ejecutar la batería de tests tras una modificación de un componente del cual dependan otros para evaluar si dichos cambios afectan al comportamiento del resto de módulos.

Si bien la escritura de código siguiendo este modelo de desarrollo suele ser sencilla, en ocasiones se dan un conjunto de condiciones que dificultan significativamente el diseño de pruebas. Generalmente dichas condiciones están relacionadas con la modificación de entidades externas, como pueden ser bases de datos o recursos en red. Para dichos casos la mayoría de herramientas de creación de tests ofrecen funcionalidad para crear "objetos simulados" (*mock*), objetos que simulan el comportamiento del elemento al que reemplazan pero evitando los efectos de los mismos (como la modificación de una base de datos o el envío o la recepción de un paquete en red). Dichos objetos permiten simular una serie de condiciones como valores de retorno o lanzamiento de excepciones, así como el análisis de los parámetros con los que es invocado (en caso de que el objeto *mock* reemplace a una función o clase) o la inclusión de parámetros únicamente relevantes durante la fase de realización de pruebas.

3.7.1.1. Utilización en el sistema final

Varias de las las herramientas del Trabajo han sido desarrolladas siguiendo este proceso de desarrollo. No ha sido posible la aplicación en la construcción de la totalidad de las herramientas debido a la falta de conocimientos sobre este proceso de desarrollo hasta comenzar con el Trabajo. Sin embargo, se han escrito tests unitarios para casi la totalidad de las herramientas existentes anteriormente, utilizando el desarrollo dirigido por pruebas en sucesiones iteraciones dentro del ciclo de desarrollo de estas aplicaciones.

3.8. Otros

3.8.1. Virtualización

Virtualizar consiste en la abstracción de la plataforma física sobre la que un conjunto de *software* es ejecutado. La virtualización de diferentes capas de un sistema facilita la compatibilidad entre componentes de características muy variadas, y posibilita la

creación de diferentes unidades independientes sobre un único equipo físico, la abstracción de diferentes capas (hardware, sistema operativo, bibliotecas) de la implementación de la aplicación final.

3.8.2. Código móvil

Se conoce como código móvil al *software* que se transfiere entre diferentes unidades (distribuido en una red, generalmente). Este tipo de aplicaciones ofrecen una serie de ventajas (principalmente la sencilla distribución del mismo, en ocasiones transparente al usuario final) muy atractivas a la hora de crear sistemas distribuidos.

La mayoría de este tipo de *software* está creado sobre herramientas multiplataforma, garantizando la abstracción del sistema sobre el que se está ejecutando. Ejemplos de código móvil son los contenidos dinámicos de una web (creados mediante **applets** Java, código **JavaScript** o controles **ActiveX**), código embebido en documentos PDF o en general cualquier tipo de *software* descargado de una red.

Sin embargo, este tipo de aplicaciones implican una serie de consideraciones adicionales de seguridad, en particular la verificación de la identidad del nodo que proporciona el paquete, la integridad del mismo durante la transferencia y [Citation needed: None]y el comportamiento del código en ejecución.

3.8.2.1. Compartimentación

Extendiendo los conceptos del código móvil a un contexto más amplio, como el de un sistema operativo, surge la idea de la compartimentación, consistente en la creación de entornos (compartimentos) capaces de ser migrados incluso en tiempo de ejecución entre máquinas, evitando la interrupción de un trabajo en caso de que el nodo sobre el que se está ejecutando deba ser interrumpido, entre otros muchos beneficios.

3.8.3. *Cross-compiling*

La compilación cruzada posibilita el desarrollo de *software* en plataformas diferentes a la plataforma objetivo, aquella sobre la que el programa final deberá ejecutarse. El compilador genera código máquina para la arquitectura objetivo a partir de los archivos de código fuente, en lugar del proceso común de generación de código para la arquitectura sobre la que se ejecuta el software de desarrollo (proceso conocido como compilación nativa).

El uso de compiladores cruzados facilita el desarrollo para diferentes plataformas y en ocasiones es la única forma de crear aplicaciones para sistemas tales como dispositivos embebidos que no cuentan con herramientas de desarrollo. También es útil para facilitar la generación de código en entornos como granjas de servidores, generación de código para emuladores o realizar procesos de *bootstrapping* (creación de las herramientas básicas de una nueva plataforma, como un sistema operativo o un enlazador).

3.8.3.1. *Toolchain*

Una cadena de herramientas (*toolchain*) se conforma del conjunto de utilidades necesarias para la construcción de un determinado producto. Generalmente estas utilidades son ejecutadas secuencialmente, utilizando la salida de una de ellas como la entrada de la siguiente. Este conjunto de herramientas comprende normalmente un compilador, un enlazador y un editor de texto, así como varias bibliotecas y un depurador, si bien puede contar con cualquier herramienta requerida para las necesidades propias del producto a crear.

3.8.4. Utilización en el sistema final

Con el objetivo de posibilitar la realización de trabajos de compilación distribuida utilizando un equipo basado en la arquitectura i686 se ha construido una cadena de herramientas que realiza procesos de compilación cruzada para la arquitectura ARMv7hf utilizando la herramienta *crosstool-ng*[Citation needed: None], consiguiendo optimizar de forma significativa el tiempo de compilación.

[Citation needed: None]http://www.nongnu.org/avr-libc/user-manual/overview.html [Citation needed: None]http://elinu

Capítulo 4

Dominio del problema

La utilización de algoritmos distribuidos implica mejoras sustanciales en una gran cantidad de aplicaciones, incrementando la capacidad global de cómputo de un sistema mediante la unión de varios dispositivos que trabajan como una única unidad manteniendo simultáneamente un alto grado de independencia y una tolerancia global a fallos muy alta. Sin embargo, el coste de la adquisición instalación y mantenimiento de dicho conjunto de nodos suele ser elevado. Además, los beneficios citados implican 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 circunstancias no solo dificultan el desarrollo de este sistema, sino también la comprensión de los fundamentos básicos de la Computación Distribuida, aspecto de relevancia para estudiantes de Ciencias de la Computación.

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 equipos ya presentes en una infraestructura) 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 similares.

En el presente proyecto se realiza un análisis de las diferentes alternativas que permitan satisfacer los objetivos definidos previamente.

4.1. Definiciones

4.1.1. Definición del dominio del problema

4.1.2. Sistema actual: Infraestructura de la Facultad de Ciencias

El sistema se ubica en una Facultad universitaria con aproximadamente 600 alumnos[Citation needed: None]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** [24]. La Facultad cuenta con varias aulas y laboratorios de informática donde los alumnos disponen de la infraestructura 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. Todos los edificios cuentan con un cableado capaz de soportar teóricamente transferencias de hasta 100Mb/s de forma bidireccional[Citation needed: None]. La gestión de un sistema de autenticación se realiza mediante el protocolo **LDAP** (*Lightweight Directory Access Protocol*)[25], 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 en las asignaturas de interés son desarrolladas en los lenguajes de programación **C** y **Java**, ya conocidos por la totalidad de los estudiantes gracias a asignaturas previamente cursadas.

Problemas conocidos Si bien la infraestructura existente es capaz de proveer a los estudiantes de los recursos necesarios, se identifican 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.

4.2. Identificación de usuarios participantes

- Estudiantes de tercer y cuarto curso del Grado en Ingeniería Informática.
- Profesores de las asignaturas Arquitectura de Computadores y Sistemas Distribuidos.
- Administradores del sistema.

4.2.1. Identificación de las necesidades de cada parte

4.2.1.1. Alumnos

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

4.2.2. 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.

4.2.3. Administrador

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

4.3. Propuestas para la búsqueda de necesidades

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

4.4. Identificación de requisitos

4.4.1. Requisitos de almacenamiento de la información

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

4.4.2. Identificación de requisitos funcionales

4.4.3. 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.

4.5. 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 elementos...).

[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

- Facilidad de trabajo y de aprendizaje (documentación disponible, proyectos similares ya realizados, conocimiento previo sobre la plataforma en cuestión...).
- Escalabilidad del sistema.
- Necesidades de mantenimiento del sistema.
- Consumo eléctrico.
- Obsolescencia del sistema (periodo de tiempo en el que el *hardware* y *software* del sistema podrán ser actualizados y ser capaces de satisfacer los requisitos para los que fue creado).

4.5.1. 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. El coste se reduce significativamente.

Inconvenientes intrínsecos del sistema

No se exploran apenas las posibilidades de un sistema distribuido formado por varios equipos físicamente independientes e impide aprovechar dicha independencia para los objetivos didácticos del sistema.

Facilidad de trabajo y curva de aprendizaje

Si bien el trabajo con cada una de las instancias es previsiblemente sencillo, debido a la eliminación de la gran parte del mantenimiento de la capa física subyacente, el uso de este tipo de sistema requiere una etapa de formación previa en materia de virtualización.

Prestaciones técnicas

Las prestaciones técnicas con las que se contaría, de llevarse a cabo este proyecto, son las de los equipos ya dispuestos para fines similares a este en el Centro[Citation needed: Andrés].

Coste económico

El coste económico es muy reducido si ya se cuenta con los equipos a utilizar y las licencias del *software* de virtualización necesarias.

Escalabilidad del sistema

Dependiente de las capacidades de virtualización del equipo disponible, y el número de nodos y usuarios a gestionar.

Necesidades de mantenimiento

Las necesidades propias de un sistema operativo multiusuario (previsiblemente **GNU/-Linux**) junto a las específicas de la virtualización de los equipos (monitor de máquinas virtuales).

Consumo energético del sistema

[Citation needed: Andrés]

Obsolescencia del sistema

Se estima una larga vida útil del sistema. Las máquinas virtuales instaladas en un sistema físico son fácilmente trasladables a otro equipo, por lo que la dependencia de la parte física del sistema es muy baja.

Material con el que se cuenta actualmente

Se plantea el aprovechamiento de equipos ya presentes en la infraestructura en la que trabajar, por lo que se estima un coste muy pequeño a la hora de adquirir material.

Prestaciones técnicas

[Citation needed: Andrés]

Análisis coste/beneficio

Si bien el coste de esta solución es muy atractivo, presenta una serie de carencias que dificultan significativamente el desarrollo del sistema en el mismo.

4.5.2. 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 solución considerada. Se reduce dramáticamente el coste de adquisición de material y permite dar un nuevo ciclo de vida a material universitario. La arquitectura es conocida y fiable.

Inconvenientes intrínsecos del sistema

No se exploran las características únicas de otros sistemas menos "convencionales", tales como la utilización de sistemas embebidos. El consumo energético es mayor y existe una mayor demanda de espacio.

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/x86-64 (dependiendo de los equipos a utilizar finalmente).
- Entre 2 y 4 GB de memoria principal.
- Conectividad Ethernet, USB.
- Almacenamiento en disco duro.

Coste económico

El coste económico de estos equipos es prácticamente nulo, pues ya se cuenta con los mismos y su utilización no exige la adquisición de nuevos equipos que los sustituyan. Estos equipos ya han sido retirados y no están empleados actualmente en ninguna tarea.

Escalabilidad del sistema

Dependiente únicamente del coste económico de la adquisición de nuevos equipos, o de la disponibilidad de equipos que no estén utilizados.

Necesidades de mantenimiento

Las propias de cualquier sistema multiusuario y las específicas del montaje dado (en materia de refrigeración, gestión de cableado, etcétera).

Consumo energético del sistema

El típico de cualquier equipo de escritorio.

Obsolescencia del sistema

Estos equipos tienen una antigüedad de aproximadamente 4 años. Dicha edad no impide que sean capaces de utilizar aplicaciones actuales, y en general no se prevé la incompatibilidad con ninguna aplicación. No obstante, son equipos relativamente antiguos que han sido utilizados de forma intensiva, por lo que la probabilidad de fallo en los mismos puede ser elevada.

Material con el que se cuenta actualmente

Se cuenta con un número suficiente de equipos para la creación del sistema final.

Análisis coste/beneficio

Si bien el coste de estos equipos es prácticamente nulo, dicho atractivo contrasta con los potenciales problemas que el uso de estos sistemas puede implicar (obsolescencia, uso de sistemas convencionales en detrimento de soluciones más innovadoras...).

4.5.3. Clúster con equipos embebidos multimedia

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

Ventajas intrínsecas de la solución

Relación potencia/precio presumiblemente similar o 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 contenido multimedia o la conexión con sistemas de almacenamiento), falta de puertos **GPIO**, **I^2C**...

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. Además, 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

Como referencia se utilizan las prestaciones de uno de los equipos más populares, el **Google Chromecast**[Citation needed: https://wikidevi.com/wiki/Google_Chromecast_%28H2G2-42%29]

- Procesador ARM de 2 núcleos a 1.2 GHz.
- 512 MB de memoria principal.
- 2 GB de almacenamiento no extensibles.
- Alimentación por micro-USB.
- Utiliza un sistema operativo basado en **Google TV**, **ChromeOS** y **Android**.

Coste económico

El coste de estos equipos es reducido, generalmente inferior a 30 por unidad.

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 conexiones inalámbricas).

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 los desarrolladores del mismo. Otras necesidades son aquellas derivadas del mantenimiento de un sistema multiusuario 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

El diseño de estos equipos está orientado a la minimización del consumo energético, por lo que se estima reducido.

Obsolescencia del sistema

La obsolescencia del sistema es difícil de determinar: no se cuenta con una gran cantidad de *software* para este tipo de sistemas más allá de las aplicaciones multimedia.

Material con el que se cuenta actualmente

No se dispone de material de estas o similares características.

4.5.4. Clúster con sistemas embebidos

Recientemente han surgido en el mercado sistemas embebidos con capacidad de cómputo elevada y precio muy reducido (en torno a los 40 euros por unidad). Estos equipos destacan además por su versatilidad. La mayoría de ellos son capaces de ejecutar una gran variedad de sistemas operativos (GNU/Linux, RISC OS, BSD, Windows...), incluyen una gran cantidad de mecanismos de interconexión y soportan la mayoría de herramientas presentes en equipos de escritorio y servidores.

Se plantea utilizar este tipo de plataformas 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.

Facilidad de trabajo y curva de aprendizaje

Existe una amplia documentación del *hardware* de este tipo de equipos, así como numerosos proyectos basados en los mismos, entre los que se incluyen sistemas similares a la solución planteada. Se cuenta además con experiencia en el manejo de estas placas.

Prestaciones técnicas

- Generalmente basados en la arquitectura ARM.
- Entre 512 MB y 2 GB de memoria principal.

- Conectividad **Ethernet**,**I²C**, **GPIO**, **USB**.
- Alimentación a través de **USB**/**GPIO**.
- Almacenamiento secundario basado en tarjetas microSD/SD, expansible a través de USB.

Coste económico

Muy reducido, con un coste por nodo de entre 20 y 40 euros, al que se debe añadir los mecanismos de alimentación e interconexión.

Escalabilidad del sistema

Dependiente únicamente del coste económico de la adquisición de nuevos equipos.

Necesidades de mantenimiento

Las mismas que cualquier sistema multiusuario.

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. Se prevé que las necesidades que el sistema cubre no demandarán una mayor potencia de cálculo en el futuro.

Material con el que se cuenta actualmente

El Departamento de Informática y Automática cuenta con varios de estos equipos actualmente que podrían disponerse para el uso en el proyecto.

4.5.5. Elección de la solución

Basándose en las características descritas anteriormente, se elige realizar el sistema utilizando sistemas embebidos de bajo coste en virtud de los siguientes aspectos positivos:

- Compatibilidad con de gran cantidad de *software* y sistemas operativos.
- Versatilidad y facilidad de interconexión.
- Se cuenta con experiencia en el uso de este tipo de dispositivos.
- Bajo coste.

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

Se opta por las placas de la familia **Raspberry Pi** para la realización el sistema debido a la gran cantidad de soporte con el que cuentan, tanto por parte de la fundación **Raspberry Pi** como por diferentes comunidades de desarrolladores. Es el computador de este tipo que más sistemas operativos soporta[Citation needed: http://elinux.org/RPi_Distributions] y existen gran cantidad de proyectos que dotan de mayor funcionalidad al sistema y que generalmente son diseñados para aprovechar las características del *hardware* específico de la máquina.

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
LINPACK [26–28]	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: None]	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 micro-USB o los pines GPIO	Mediante micro-USB o los pines GPIO	Mediante micro-USB o los pines GPIO
Sistemas operativos compatibles	Arch Linux 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: None]	Los mismos que para el modelo B	Hasta la fecha, únicamente: Ubuntu Snappy Core, Raspbian, OpenELEC, RISC OS, Según la web de Arch Linux, 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).

Cuadro 4.1: Comparativa de los diferentes modelos de Raspberry Pi

[Citation needed: http://en.wikipedia.org/wiki/Raspberry_Pi#Software] [Citation needed: https://learn.adafruit.com/embedded-linux-board

[Citation needed: https://learn.adafruit.com/introducing-the-raspberry-pi-2-model-b/performance-improvements]

[Citation needed: http://raspi.tv/2014/how-much-less-power-does-the-raspberry-pi-b-use-than-the-old-model-b]

4.5.7. Elección del sistema operativo

Nombre	Enfoque	Características notables	Ventajas	Inconvenientes	Software disponible
Arch Linux ARM	Distribución ligera centrada en el minimalismo y la disponibilidad de software novedoso. Requiere sin embargo que el usuario esté familiarizado con el sistema 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 Fundación 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 equiparable si se cuenta el **AUR** (*Arch User Repository*)
Ubuntu Snappy Core	Centrado en la facilidad de uso.	Es la distribución más popular (en equipos de escritorio) contando con gran cantidad de *software* disponible	Fácil de configurar, gran cantidad de soporte	Recientemente portado a 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	Centrado en la estabilidad del sistema en detrimento de las últimas versiones de los componentes del sistema.	Es el sistema más utilizado en la plataforma Raspberry Pi. La fundación Raspberry Pi promociona su uso y la mayoría de los desarrolladores de la plataforma crean herramientas para este sistema.	Estable, gran cantidad de *software* disponible, ya conocido.	La instalación básica del sistema incluye una gran cantidad de herramientas que consumen recursos del sistema de forma significativa.	

Cuadro 4.2: Comparativa de sistemas operativos (1)

Nombre	Enfoque	Características notables	Ventajas	Inconvenientes	Software disponible
RISC OS	Diseñado específicamente para la arquitectura ARM, aprovechando las posibilidades de dicha arquitectura al máximo.	Eficiente, basado en el RISC OS original, incluyendo características del mismo. Sistema monousuario con multitarea cooperativa (en contraste con multihilo o multitarea apropiativa).	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 (con el que no se cuenta experiencia).	
Gentoo	Diseñado para permitir la modificación del sistema al máximo nivel posible. Todo el *software* es compilado en la máquina que lo instala, en lugar de utilizar ejecutables precompilados	Enfocado en la personalización.	Permite ser modificado de forma sencilla.	Poco soportado en Raspberry Pi.	
Windows 10	Diseñado para el paradigma del Internet de las Cosas,	Sencillo de utilizar, con soporte (previsiblemente) del *framework* **.NET**.	Soporta con un conjunto de tecnologías conocidas no compatibles con ninguna otra alternativa. Si el soporte de **.NET** es ofrecido constituiría una ventaja clave.	Aún no se encuentra disponible[29]. Esta diseñado para un propósito especifico. No compatible con software para Linux de forma nativa.	

Cuadro 4.3: Comparativa de sistemas operativos (2)

4.6. Propuesta de solución definitiva

En función de la evaluación llevada a cabo se extraen las siguientes decisiones de diseño que conforman la propuesta de solución definitiva:

4.6.1. Hardware

Todo el sistema se construirá sobre placas **Raspberry Pi** debido a su alta versatilidad, gran potencia de cálculo, interfaces de comunicación, soporte por parte de las diferentes comunidades de desarrolladores y consumo eléctrico.

4.6.2. Sistema operativo

El sistema operativo a utilizar será **Arch Linux ARM**, debido a la gran comunidad de soporte con la que cuenta, compatibilidad con la gran mayoría de componentes presentes en un sistema GNU/Linux y modelo arquitectónico que apuesta por la simplicidad, *limpieza* y eficiencia del sistema.

4.6.3. Herramientas de desarrollo a utilizar

Se plantea el uso del lenguaje de programación Python como herramienta principal de desarrollo, debido a su potencia de cálculo y simplicidad, que permite crear aplicaciones que consuman pocos recursos (aspecto vital, máxime cuando se utilizará sobre un sistema con un *hardware* poco potente) de forma sencilla y rápida. También se plantea el uso de los lenguajes de programación C y C++ para el desarrollo de herramientas a bajo nivel, así como herramientas web para el desarrollo de aplicaciones utilizadas por el usuario.

Capítulo 5

Análisis

Capítulo 6

Arquitectura física

Capítulo 7

Arquitectura software

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.

7.1. 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*)[30], un estándar *de facto*[31] 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[32]. 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[33].

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.

7.1.1. 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**[34]. 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 el anexo

7.2. 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: None]. 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.

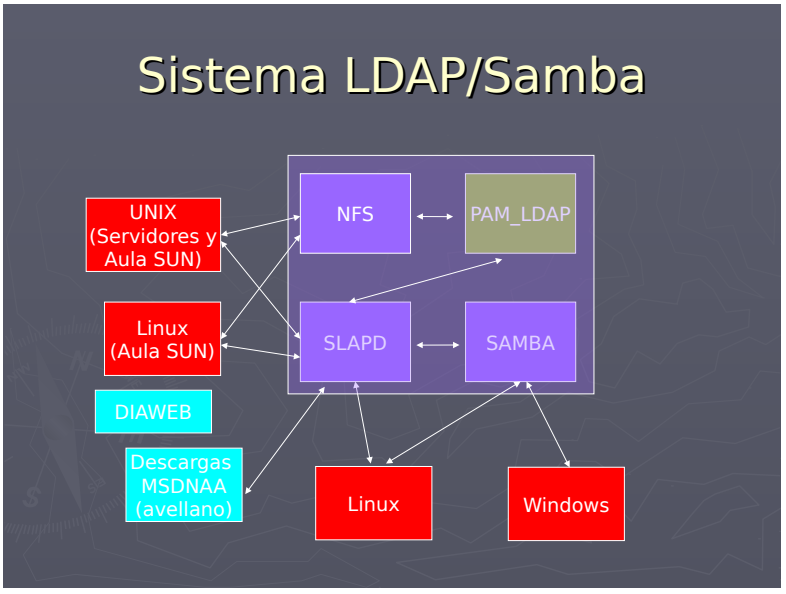


FIGURA 7.1: Esquema de los diferentes componentes del sistema de autenticación y gestión de archivos, así como de una serie de componentes adicionales. Obsérvese la interacción entre los componentes situados en el rectángulo interior

7.2.1. 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*).

7.2.2. 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.

7.3. 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.

7.3.1. 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

7.3.2. Análisis del rendimiento

Para determinar el rendimiento del compilador se ha utilizado el mismo en la compilación de diferentes herramientas a utilizar en el sistema:

7.3.2.1. OpenMPI

Tiempo de compilación en Rasbperry Pi: 4447 seconds (1 h y 14 minutos)

Tiempo de compilación con el compilador cruzado sin paralelización: 2710 seconds (45 minutos)

Tiempo de compilación con 4 trabajos paralelos (make -j4): 1267 seconds (21 minutos)

Capítulo 8

Servicios del sistema operativo

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.

8.1. 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[35]. 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**

[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.

8.1.1. MarcoPolo: Introducción

MarcoPolo es un protocolo de descubrimiento de servicios cuya dinámica y nombre se inspiran en el juego homónimo[Citation needed: None], en el cual uno de los integrantes debe encontrar 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: None]

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).

8.1.1.1. 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

8.1.2. 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.

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

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

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.

8.1.3. Esquemas de comunicación

8.1.3.1. Comando Marco

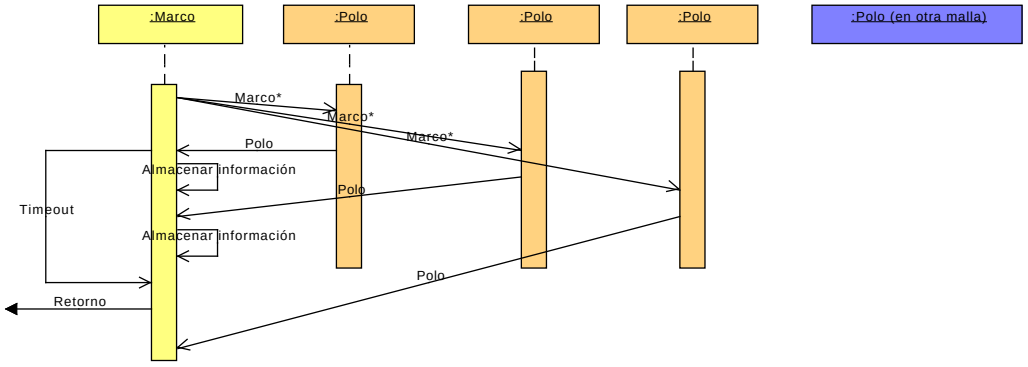


FIGURA 8.1: Interacción al enviar el comando **Marco**. Los mensajes a grupos *multicast* se indican con "*"

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.

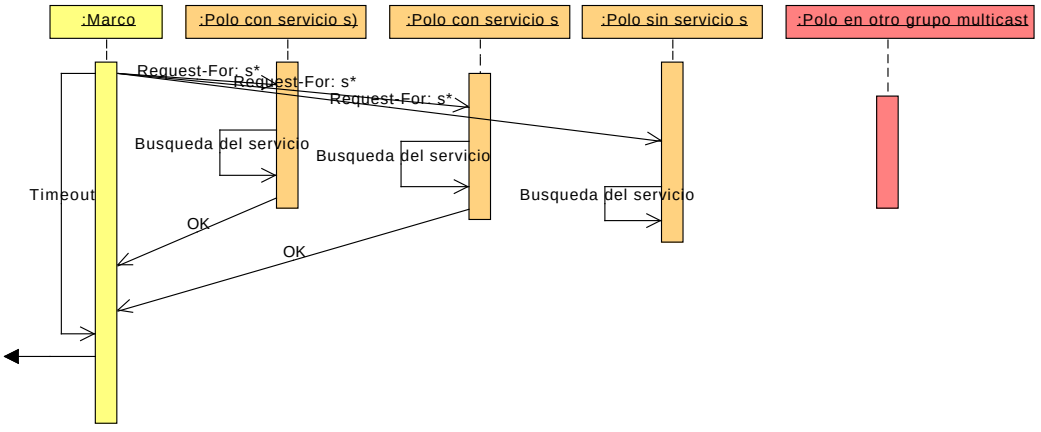


FIGURA 8.2: Diagrama de interacción al enviar el comando **Request-For**. Los nodos comprueban si deben ofrecer el servicio identificado por la clave *s*. En caso de que la búsqueda sea exitosa se retorna un mensaje indicando la disponibilidad de dicho nodo. En caso contrario no habrá respuesta alguna. Los mensajes enviados a grupos *multicast* se indican con "*"

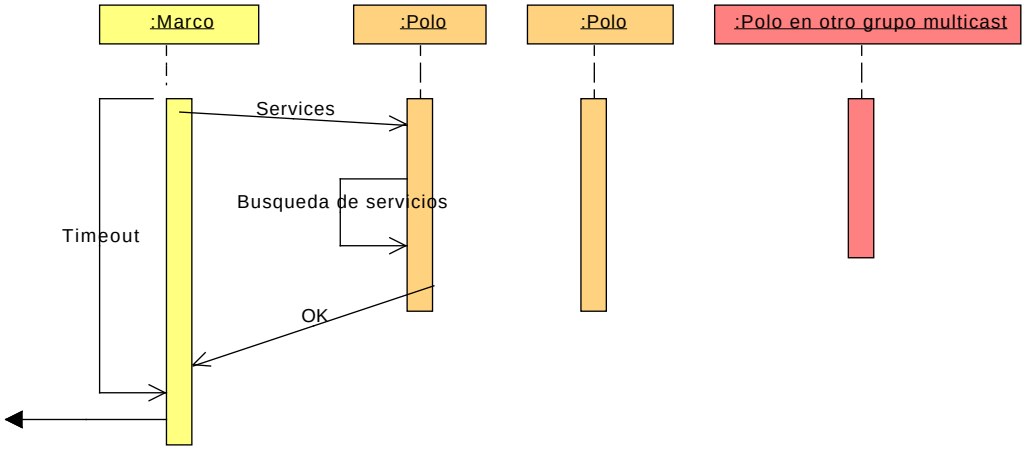


FIGURA 8.3: Diagrama de interacción al enviar el comando **Services**. El nodo al que se le envía el comando consulta la información sobre los servicios que posee y posteriormente envía una respuesta a la instancia de Marco que ha realizado la consulta. Obsérvese toda la información es enviada en modo *unicast*

8.1.4. 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.

8.1.4.1. 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* [36]).

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

data.txt

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

3 directories, 7 files
```

FIGURA 8.4: 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 8.1: Un archivo que describe el servicio status monitor

8.1.4.2. 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.

8.1.5. 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: None] 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.

8.1.6. 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`)[37], 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.

8.2. Aplicaciones construidas sobre MarcoPolo

8.2.1. 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.

8.2.1.1. `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 8.5: Opciones de configuración de `marcodiscover`

8.2.1.2. Marcoinstallkey

8.2.2. Aplicaciones construidas sobre MarcoPolo

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

8.2.2.1. 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: None]. Esto es posible debido a que según la especificación del estándard de websockets, la *Same-Origin Policy*[38] no es utilizada de la misma forma que en peticiones HTTP,[39].

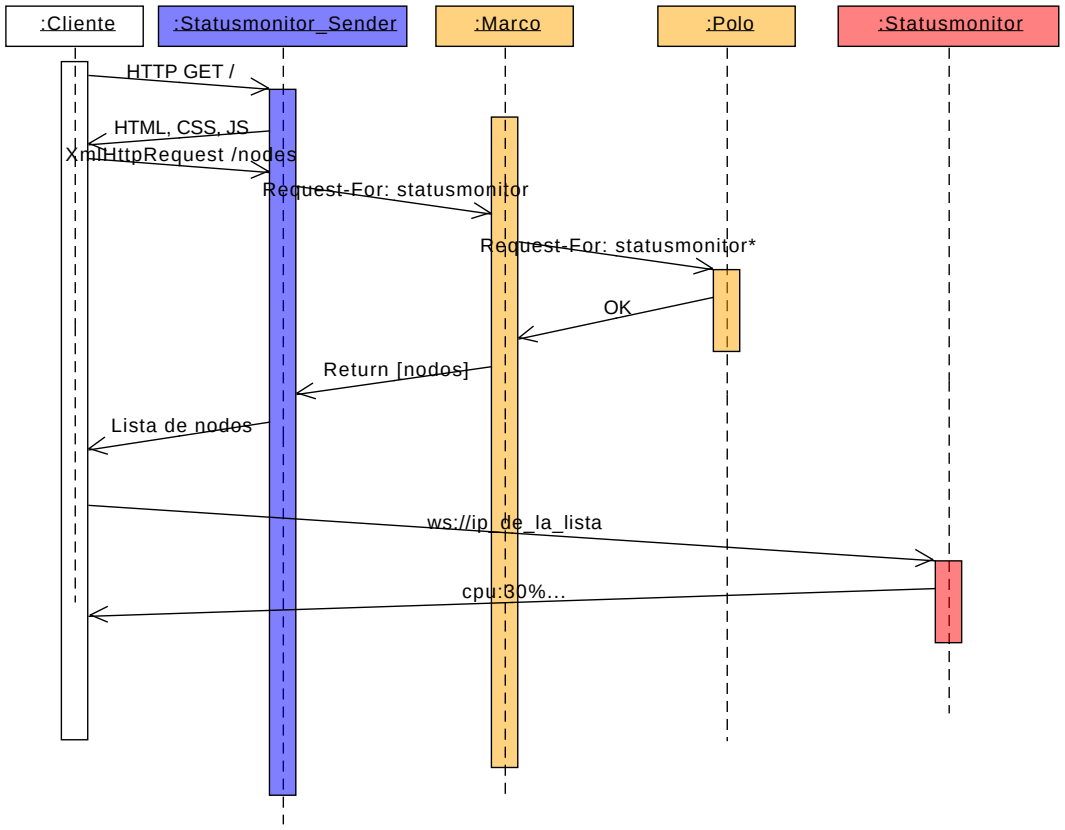


FIGURA 8.6: Interacción completa del usuario con **statusmonitor**. Los mensajes a grupos *multicast* se indican con "*"

. 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 8.7: 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.

8.2.2.2. 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: None]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

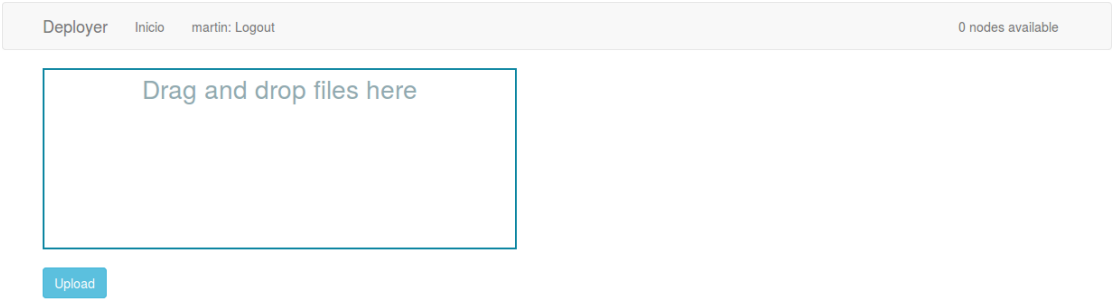


FIGURA 8.8: Interfaz web del deployer. A la izquierda figuran los controles y a la derecha la lista de nodos sobre los que se puede realizar el despliegue

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: None], utilizando **python-pam** para el acceso al mismo desde **Python**[40].

8.2.3. Herramientas de gestión

8.2.3.1. pam_mkhomedir

PAM permite ampliar la funcionalidad que ofrece por defecto mediante la inclusión de *módulos*, pequeñas bibliotecas compartidas de código con las acciones a realizar.

Uno de los módulos incluidos en la instalación por defecto de **PAM** es **mkhomedir**, encargado de la creación de un directorio propio para el usuario en caso de que aún no se haya realizado dicha acción. El proceso consiste en una copia del directorio **skeleton** (generalmente situado en `/etc/skel`) y la correcta fijación de permisos en el mismo.

[6] jquery.com/

Sin embargo, el directorio únicamente es creado en el nodo al que se accede. La filosofía del proyecto se basa en la creación de un sistema que se componga de varias unidades, pero se comporten como una única entidad, por lo que obligar al usuario a acceder a todos los nodos para poder trabajar en el sistema completo rome dicha transparencia. Es por ello necesario contar con un sistema que extienda la funcionalidad de **mkhomedir** para incluir el resto de nodos en la creación del directorio.

Con este objetivo nace **pam_mkpolopohomedir**, un módulo basado en **mkhomedir** que hace uso de **MarcoPolo** para detectar los nodos presentes en la red que ofrezcan el servicio `polousers` y solicitar a los mismos la creación del directorio de inicio.

El módulo se implementa en C/C++ debido a que es el lenguaje con el que los módulos de PAM se construyen por defecto y utiliza por tanto el *binding* de MarcoPolo para este lenguaje. Existen sin embargo herramientas para desarrollar el mismo en Python[7].

Un módulo en PAM debe implementar una serie de funciones que constituirán los puntos de entrada al módulo[Citation needed: None]pam manpages:

- `PAM_EXTERN int pam_sm_open_session(pam_handle_t  pamh, int flags, int argc ,const char **argv)`

  Es la función que **PAM** invoca al incluir el módulo. Incluye en el mismo una estructura `pam_handle_t` con toda la información relevante sobre el usuario que ha iniciado sesión y los parámetros indicados en los ficheros de configuración de `PAM` (en el caso de este módulo, los permisos del directorio de inicio y la localización del directorio *skeleton*).

- `PAM_EXTERN int pam_sm_close_session(pam_handle_t * pamh, int flags, int argc, const char **argv)`

  Es la función que **PAM** utiliza para indicar al módulo que la sesión ha terminado. En el caso del módulo a crear no se debe realizar ninguna acción en este evento, sin embargo es necesario implementarla debido a que PAM la requiere.

- `struct pam_module _pam_mkhomedir_modstruct`

  Define las características del módulo y los puntos de entrada que define.

El módulo realiza la siguiente secuencia de pasos:

[7] [Citation needed: None]python-pam

1. Determina la información de interés a partir de los datos provistos por **PAM** y las acciones a llevar a cabo. En caso de que el directorio ya exista, se omite la creación del mismo y únicamente se solicita la creación en el resto de nodos[8].
2. Realiza si es necesaria la creación del directorio en el nodo actual.
3. Detecta con **MarcoPolo** el resto de nodos dispuestos a colaborar en el sistema (aquellos que oferten el servicio **polousers**).
4. Solicita a cada uno de ellos la creación del directorio.
5. Una vez que todos los nodos han realizado la acción solicitada, se da acceso al sistema.

Todas las operaciones realizan escrituras en ficheros de **log** para su posterior análisis.

Seguridad Al tratarse de acciones llevadas a cabo por usuarios con privilegios elevados y que involucran la gestión de información personal, todas las comunicaciones se realizan utilizando conexiones cifradas mediante *sockets* **TLS** (*Transport Layer Security*) con certificados en ambos lados de la conexión, que son verificados por el contrario.

Extensibilidad El módulo ha sido diseñado con el objetivo de posibilitar la adición de nueva funcionalidad al mismo. Únicamente es necesario definir las acciones a llevar a cabo en el servicio **polousers** y solicitar su realización mediante la sintaxis de comandos de **MarcoPolo**.

8.2.3.2. MarcoBootstrap

Uno de los mayores problemas a la hora de gestionar un sistema con un número de componentes elevado es la instalación de todos los componentes y la actualización de los mismos. **MarcoBootstrap** es una herramienta que posibilita la instalación del sistema operativo de forma autónoma y que únicamente requiere la copia en la tarjeta SD del sistema de un pequeño conjunto de utilidades, cuyo tamaño es 33 *megabytes*.

[8] Este paso siempre es necesario para facilitar la expansibilidad del sistema: añadir un nuevo nodo tras la creación del directorio en el resto haría que este no fuera accesible de no ser por la repetición de este paso

8.2.3.3. Marcologger

8.2.4. Herramientas didácticas

8.2.4.1. The LED API

8.2.4.2. MusicPI

8.2.5. Pruebas de concepto

Capítulo 9

Aplicaciones

Capítulo 10

Herramientas de terceros

10.1. Herramientas utilizadas para la creación del sistema

10.1.1. Arch Linux ARM

Arch Linux es una distribución del sistema operativo GNU/Linux creado y mantenido por una comunidad de usuarios. Entre los objetivos principales del proyecto figuran el minimalismo del sistema, siguiendo el principio de diseño **KISS** (*Keep It Simple, Stupid*)[Citation needed: None]http://people.apache.org/ fhanik/kiss.html. Dicho minimalismo se traduce en una arquitectura basada en paquetes que conforman en su conjunto un sistema fácil de comprender por el usuario, y que cuentan con una gran cantidad de documentación fácilmente accesible. La consecuencia directa de este minimalismo es el rendimiento del sistema. Una instalación de Arch Linux se limita al conjunto de paquetes mínimo para contar con un sistema completamente funcional, delegando al usuario la adición de nuevos paquetes. Este enfoque permite optimizar de forma sencilla el rendimiento del sistema, al no contar con paquetes innecesarios.

Además, Arch Linux apuesta por un modelo de desarrollo de liberación continua (*rolling releases*). El sistema operativo no se distribuye en versiones, sino en imágenes con las últimas versiones de los paquetes disponibles, lo cual posibilita contar con las últimas versiones de las herramientas presentes en el sistema operativo poco tiempo (o de forma inmediata) a su liberación. El modelo de desarrollo sigue este principio de forma tan estricta, que basta una actualización de todos los paquetes del sistema mediante el gestor propio de la distribución (**pacman**) para actualizar el sistema operativo (en contraste con operaciones específicas para este cometido, como ocurre con otros sistemas operativos).

El sistema operativo delega la responsabilidad del mantenimiento de todos los componentes al usuario en un grado mayor que el que pueden ofrecer otras distribuciones. Esto permite a los usuarios contar con completa libertad para modificar componentes del sistema en función de sus intereses a cualquier nivel[Citation needed: None]https://wiki.archlinux.org/index.php/The_Arch_

Arch Linux ARM es un proyecto derivado de Arch Linux que tiene como objetivo portar el sistema operativo a dispositivos basados en la arquitectura ARM (pues este proyecto está enfocado únicamente en las arquitecturas i686 y x68_64[Citation needed: None]), generalmente sistemas embebidos, habiendo conseguido la compatibilidad con las versiones v5te, v6h y v7h de la arquitectura[Citation needed: None]. El proyecto mantiene la misma filosofía de diseño que su progenitor, siendo el buen rendimiento del sistema operativo uno de los aspectos que propician el uso de esta distribución en sistemas con un capacidad de cómputo reducida, siendo por ello una de las principales opciones a la hora de realizar proyectos con este tipo de computadores[Citation needed: None]http://distrowatch.com/search.php?architecture=arm.

Además, la comunidad de Arch Linux es famosa por la gran y exhaustiva cantidad de documentación que mantiene, así como las diferentes opciones de soporte a cualquier nivel que ofrece a través de diferentes canales[Citation needed: None].

El proceso de elección del sistema operativo puede observarse en 4.5.7.

10.1.2. Lenguajes de programación

10.1.2.1. Python

Python es un lenguaje de programación interpretado de propósito general que prioriza la legibilidad del código y la rapidez de desarrollo, manteniendo estas propiedades en proyectos de cualquier escala. Python soporta diferentes paradigmas de programación, entre ellos la orientación a objetos, programación imperativa y la programación funcional. Automatiza la gestión de memoria y utiliza un sistema de tipado dinámico rígido[Citation needed: None].

10.1.2.2. Lenguajes

Python se ha elegido como lenguaje principal en el desarrollo de herramientas para el sistema en detrimento de otras opciones:

Nombre	Características notables	Pros	Contras	Inclusión en el sistema
Python	Orientación a objetos, portable	Portable, buen rendimiento	Necesidad de un intérprete	Se incluye en los componentes de alto nivel del sistema.
C	Imperativo, acceso a características de muy bajo nivel de forma sencilla	El desarrollo en el lenguaje suele ser más complejo que en otros lenguajes de más alto nivel	Se incluye en componentes que trabajan con entornos tediosos donde el rendimiento es crucial, o no se puede contar con un intérprete.	
C++	Orientado a objetos	Gran rendimiento, acceso a todas las características de C	No es portable fácilmente en algunos casos	Se incluye en conjunción con C.
Java	Orientado a objetos	Multiplataforma, popular y sencillo de utilizar	El rendimiento del lenguaje y su *JVM* en el sistema son inferiores al de otras alternativas	Se utiliza en los paquetes *software* que hacen uso de **Tomcat** o **Hadoop**.
Perl				

10.2. Herramientas utilizadas para la creación de *software*

10.2.1. Twisted

Twisted [Citation needed: None]https://twistedmatrix.com/ es un motor dirigido por eventos para la creación de aplicaciones basadas en red. Uno de los principales beneficios de la programación orientada a eventos es la capacidad del sistema de optimizar el tiempo de CPU y evitar cambios de contexto, pues todo el código se ejecuta en un único hilo. Twisted se basa en el patrón de diseño **reactor**[9], que se basa en la gestión de diferentes eventos, su demultiplexación y el envío a los manejadores apropiados de forma síncrona (ver **??**).

Twisted permite crear de forma sencilla sockets asíncronos a bajo nivel en los protocolos UDP y TCP y aplicaciones que utilizan protocolos bien definidos, como HTTP o DNS. Es capaz de trabajar con protocolos como **multicast** o **TLS** e integra funcionalidades para el desarrollo dirigido por pruebas (*test-driven development*).

Twisted se ha utilizado para la creación de la herramienta de descubrimiento de servicios **MarcoPolo** (ver 8.1).

10.2.2. Tornado

Tornado es un *framework* web y una biblioteca para aplicaciones en red que utiliza mecanismos de entrada/salida asíncrona, permitiendo crear herramientas como **WebSockets** de forma sencilla y escalable. Todo el código, a menos que explícitamente se indique lo contrario se ejecuta en un único hilo.

Tornado se utiliza en todas las interfaces web creadas, en ocasiones en conjunción con **Django** y se integra con **MarcoPolo** a través del *binding* para Python.

10.2.3. Websockets

El protocolo WebSocket [38] posibilita el establecimiento de un canal bidireccional en una arquitectura cliente-servidor sobre el protocolo HTTP/HTTPS evitando el uso de peticiones asíncronas (XmlHttpRequest, <iframe>) y *polling*.

La mayoría de las interfaces web creadas utilizan este tipo de comunicación para obtener información desde los diferentes nodos del sistema, optimizando la comunicación al reducirse el intercambio de datos al momento en el que estos son necesarios (al contrario

de otras estrategias) y posibilitando la difusión de eventos en directo, a diferencia de estrategias como el *polling*.

10.2.4. PAM, LDAP

Siguiendo el objetivo funcional[Citation needed: None], la gestión de los usuarios está delegada al sistema de autenticación preexistente en la infraestructura en la que se integra el sistema. En ella se cuenta con un servidor **LDAP** con la información de los usuarios. Mediante la configuración del paquete LDAP es posible accedeer a los mismos, y gracias a **PAM** (*Pluggable Authentication Module*) se integra con el resto de métodos de autenticación presentes en el sistema.

Se ha creado un módulo para **PAM** para facilitar las tareas que el sistema debe realizar. Dicho módulo integra **MarcoPolo** y es conocido como **pam_mkpolohomedir**8.2.3.1.

10.2.5. OpenSSL

10.2.6. Distcc

10.2.7. Hadoop

10.2.8. *Message Passing Interface*

La necesidad de una herramienta de comunicación independiente de la plataforma derivó en la especificación del estándar MPI[41], un conjunto de interfaces para la creación de aplicaciones paralelas mediante la gestión de las operaciones de entrada-salida, definición de tipos de datos, grupos de proceso, creación y gestión de procesos, interfaces externas, etcétera. La especificación se define independientemente del lenguaje, si bien incluye implementaciones en C, C++, Fortran 77 y Fortran 95.

MPI se ha convertido con el paso de los años en la interfaz de referencia para la creación de aplicaciones distribuidas, contando con varias implementaciones como **MPICH** (la implementación original) u **OpenMPI** (presente en la mayoría de supercomputadores), de tipo libre, o implementaciones propietarias, tales como IBM MPI, Intel MPI, Cray MPI, Microsoft MPI, Myricom MPI.

MPI es utilizado en el sistema como herramienta de desarrollo de aplicaciones distribuidas, utilizando **MarcoPolo** para simplificar el proceso de descubrimiento de nodos (ver 8.2.1.1). Además se han creado herramientas accesorias para facilitar varias tareas generalmente necesarias durante el desarrollo con la biblioteca (ver 8.2.1.2).

10.2.9. Tomcat

10.3. Herramientas utilizadas para la gestión de código, calidad de *software* y el proyecto

10.3.1. Git

10.3.2. Redmine

10.3.3. Unittest, CppUnit, Trial

10.4. Herramientas utilizadas para la documentación del proyecto

10.4.1. LaTeX

10.4.2. Sphinx

10.5. Herramientas para la gestión de usuarios

10.5.1. SSH-HPN

Una de las características de OpenSSH es que todas las tareas se ejecutan en un único proceso y por tanto, en un único núcleo, constituyendo un cuello de botella que se hace notable en sistemas de bajas prestaciones, como el sistema a modelar.

Con el objetivo de superar este límite nace SSH-HPN[Citation needed: None], un conjunto de modificaciones al código fuente de OpenSSH que optimiza la ejecución del mismo mediante el uso de diferentes procesos repartidos en los diferentes núcleos del sistema. El proyecto se distribuye como un archivo `.diff` que se incluye en los archivos del código fuente con la herramienta **GNU patch**.

Se ha creado un paquete de Arch Linux con el código fuente ya preparado para trabajar en la arquitectura ARM, y todos los nodos del sistema utilizan esta versión en lugar del paquete **OpenSSH** original.

Capítulo 11

Técnicas

11.1. Otras tecnologías de terceros

Archivo BibTeX con todas las referencias de las RFCs: http://tm.uka.de/ bless/bibrf-cindex.html

Capítulo 12

Metodología de desarrollo

Capítulo 13

Evaluación de usuarios

Capítulo 14

Conclusiones

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

[4] S. Cox, "Southampton engineers a Raspberry Pi Supercomputer ." http://www.southampton.ac.uk/~sjc/raspberrypi/Raspberry_Pi_supercomputer_11Sept2012.pdf, Feb. 2011.

[5] Paralella, "Paralella." https://www.parallella.org/.

[6] D. J. Becker, T. Sterling, D. Savarese, J. E. Dorband, U. A. Ranawake, and C. V. Packer, "Beowulf: A parallel workstation for scientific computation," *Proceedings of the International Conference on Parallel Processing*, 1995.

[7] Object Management Group, "Common Object Request Broker Architecture (CORBA) Specification, Version 3.3," 2012.

[8] J. Nestor, D. A. Lamb, and W. A. Wulf, "IDL, Interface Description Language," 1981.

[9] E. J. Coplien, D. C. Schmidt, and D. C. Schmidt, "Reactor - An Object Behavioral Pattern for Demultiplexing and Dispatching Handles for Synchronous Events," 1995.

[10] J. Moy, "OSPF Version 2." RFC 2328 (INTERNET STANDARD), Apr. 1998. Updated by RFCs 5709, 6549, 6845, 6860, 7474.

[11] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems: Concepts and Design*, ch. Chapter 15. In [42], 5th ed., 2011.

[12] A. S. Tanenbaum and M. v. Steen, *Distributed Systems: Principles and Paradigms (2Nd Edition)*, ch. 6. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006.

[13] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2." RFC 5246 (Proposed Standard), Aug. 2008. Updated by RFCs 5746, 5878, 6176, 7465, 7507.

[14] E. Rescorla, "HTTP Over TLS." RFC 2818 (Informational), May 2000. Updated by RFCs 5785, 7230.

[15] V. Samar and R. Schemers, "Unified login with pluggable authentication modules." http://www.opengroup.org/rfc/rfc86.0.html, Oct. 1995.

[16] J. Postel, "Internet Protocol." RFC 791 (INTERNET STANDARD), Sept. 1981. Updated by RFCs 1349, 2474, 6864.

[17] S. Liang and D. Cheriton, "TCP-SMO: extending TCP to support medium-scale multicast applications," in *INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 3, pp. 1356–1365 vol.3, 2002.

[18] M. Mysore and G. Varghese, *FTP-M, an FTP-like Multicast File Transfer Application.* Department of Computer Science and Engineering, University of California, San Diego, 2001.

[19] M. Barcellos, A. Detsch, H. H. Muhammad, G. B. Bedin, P.-g. C. Aplicada, C. Centro, and C. E. Tecnolgicas, "Efficient TCP-like Multicast Support for Group Communication Systems," in *In Proceedings of the IX Brazilian Symposium on Fault-Tolerant Computing*, pp. 192–206, 2001.

[20] V. Visoottiviseth, T. Mogami, N. Demizu, Y. Kadobayashi, and S. Yamaguchi, "M/TCP: The Multicast-extension to Transmission Control Protocol," in *In Proceedings of ICACT2001, Muju, Korea*, 2001.

[21] R. R. Talpade and M. H. Ammar, "Single Connection Emulation (SCE): An Architecture for Providing a Reliable Multicast Transport Service," 1994.

[22] M. Handley, S. Floyd, B. Whetten, R. Kermode, L. Vicisano, and M. Luby, "The Reliable Multicast Design Space for Bulk Data Transfer." RFC 2887 (Informational), Aug. 2000.

[23] T. Bray, "The JavaScript Object Notation (JSON) Data Interchange Format." RFC 7159 (Proposed Standard), Mar. 2014.

[24] Universidad 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.

[25] Network Working Group, "Comment on RFC 4516 - Lightweight Directory Access Protocol (LDAP)," *RFC*, June 2006.

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

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

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

[29] 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/.

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

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

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

[33] I. Corporation and Systemsoft, "Preboot Execution Environment (PXE) Specification," in *Preboot Execution Environment (PXE) Specification* [30].

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

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

[36] Filesystem Hierarchy Standard Group, "Filesystem Hierarchy Standard," 2004.

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

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

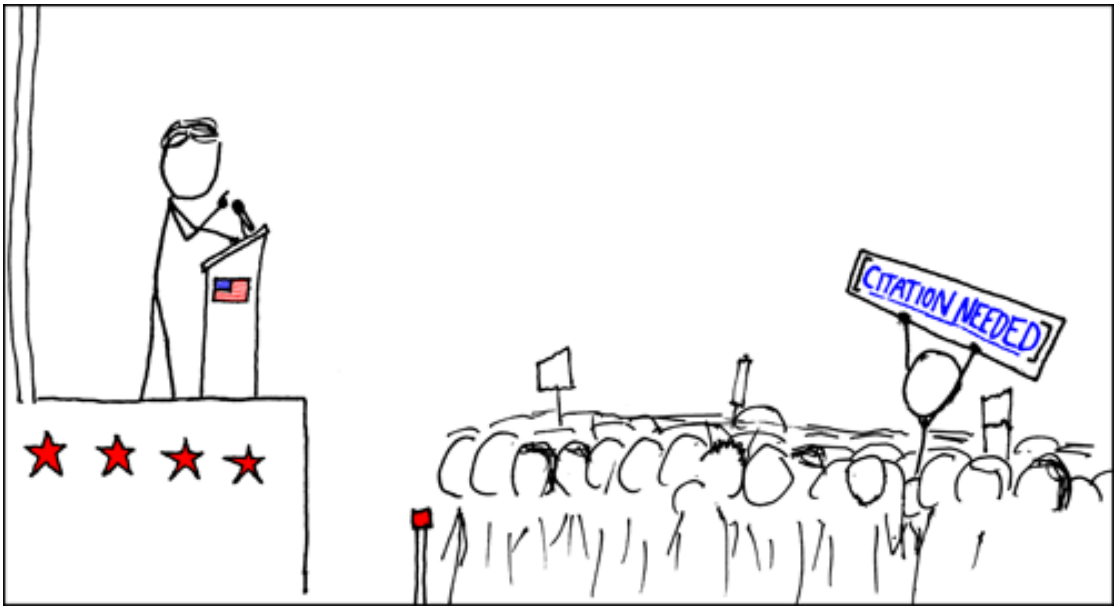
[39] A. Barth, "The Web Origin Concept." RFC 6454 (Proposed Standard), Dec. 2011.

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

[41] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard, Version 2.2," specification, September 2009.

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

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



There are 48 undefined references