VNiVERSiDAD D SALAMANCA

CAMPUS DE EXCELENCIA INTERNACIONAL

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

supervisado por

Rodrigo Santamaría Vicente

José Andrés Vicente Lober

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

9 de julio de 2015

D. Rodrigo Santamaría Vicente y D. José Andrés Vicente Lober, miembros de la Universidad de Salamanca

CERTIFICAN:

Que el trabajo titulado "Diseño e implementación de un sistema de computación distribuida con Raspberry Pi, y estudio comparativo del mismo frente a otras soluciones" ha sido realizado por D. Diego Martín Arroyo, con DNI 70825993T y constituye la memoria del trabajo realizado para la superación de la asignatura Trabajo de Fin de Grado de la Titulación Grado en Ingeniería Informática de esta Universidad.

Y para que así conste a todos los efectos oportunos.

En Salamanca, a 30 de junio de 2015

Rodrigo Santamaría Vicente
Dpto. de Informática y Automática
Universidad de Salamanca

José Andrés Vicente Lober
Dpto. de Informática y Automática
Universidad de Salamanca

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.
- Se ha indicado claramente 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.
- 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: _____

Gather ye rosebuds while ye may,
Old Time is still a-flying:
And this same flower that smiles to-day
To-morrow will be dying.

The glorious lamp of heaven, the sun,
The higher he's a-getting,
The sooner will his race be run,
And nearer he 's to setting.

That age is best which is the first,
When youth and blood are warmer;
But being spent, the worse, and worst
Times still succeed the former.

Then be not coy, but use your time,
And while ye may, go marry:
For having lost but once your prime,
You may for ever tarry.

Robert Herrick

Resumen

por Diego Martín Arroyo

El uso de un sistema distribuido como alternativa económica a un equipo de altas prestaciones es una práctica aprovechada desde hace décadas todo tipo de entornos. El auge actual de sistemas embebidos con gran potencia de cálculo y bajo coste en ha motivado su utilización para la construcción de este tipo de sistemas de computación. Sin embargo, la mayoría de las soluciones responden a una necesidad específica. En el presente documento y los anexos que lo acompañan se plantea y desarrolla la creación de un sistema distribuido compuesto por placas Raspberry Pi que es aprovechable por un gran rango de usuarios con diferentes necesidades (investigadores, desarrolladores, estudiantes…), destacando su papel como herramienta didáctica. El resultado final consiste en un sistema escalable y autoconfigurado que incluye un amplio abanico de herramientas de desarrollo de algoritmos distribuidos.

Todas las herramientas creadas cumplen los requisitos de transparencia encontrados en cualquier sistema distribuido. Sumado a su carácter autoconfigurado, el sistema es una solución autosuficiente que gracias al uso de diferentes mecanismos de optimización, hace de este una herramienta de coste mínimo una herramienta con una gran potencia de cálculo.

La construcción del sistema ha implicado el desarrollo de protocolos, utilidades y aplicaciones en diferentes capas, desde servicios provistos por el sistema operativo, pasando por utilidades consumibles por aplicaciones hasta aplicaciones que interactúan con usuarios finales. Todas estas adiciones son transparentes al usuario y altamente integrables con *software* preexistente.

El proceso de desarrollo es precedido por una serie de etapas de evaluación y de un estudio de las diferentes alternativas valoradas. Otro de los frutos del desarrollo del sistema es, por tanto, un estudio comparativo de diferentes propuestas de solución a un problema concreto y de interés en diversas áreas de las Ciencias de la Computación. El carácter didáctico del sistema es también un aspecto clave del desarrollo, pues uno de los objetivos definidos es su utilización en un contexto universitario, facilitando el análisis, desarrollo y comprensión del paradigma de computación distribuida.

Palabras clave: *zeroconf*, sistema distribuido, plataforma orientada a servicios, enseñanza, marcopolo, raspberry pi, python.

Abstract

by Diego Martín Arroyo

The usage of distributed systems as an economic alternative to a high performace computers is a practice applied in all sorts of environments for decades. The ongoing rise of embedded systems with great computational power and low cost has triggered their usage for that sort of computational systems. However, the majority of proposals are focused on a specific need. This document and the attached appendices, annexes and technical documents cover the design and development of a distributed system made of Raspberry Pi boards which is usable by a broader range of users with different needs (researchers, developers, students…), featuring its role as a didactic tool. The final result is a scalable and autoconfigured system featuring a big set of distributed algorithms development tools.

All the development tools fulfill the typical transparency requirements found in any distributed system. Added to its autoconfigured nature, the system is a self-sustaining solution which, thanks to several optimization mechanisms, offers a big computing power with very little cost.

The development of the system involves the creation of protocolos, utilities and applications in a layered architecture, from operating system services and utilities to user applications. All the internals are transparent to the user and highly compatible with preexisting software.

The development process is preceded by a series of evaluation phases and an analysis on different alternatives. One of the deliverables of the project is therefore a comparative study on several proposals to a this problem.

Another key component of the development process is the didactic focus of the system. One of the main goals is the usage of the product in a college environment, easing the analysis, development and understanding of the Distributed Computation paradigm fundamentals.

Keywordks: zeroconf, distributed system, service-oriented architecture, teaching, marcopolo, raspberry pi, python.

Reconocimientos

El resultado final de este proyecto no se explica sin el apoyo de Carlos Sánchez, promotor de la idea original, Rodrigo Santamaría y José Andrés Vicente, que desde el primer momento aceptaron la propuesta como tutores, el Departamento de Informática y Automática de la Universidad de Salamanca, que ha financiado la totalidad del proyecto, Juan Luis Boya, sin el cual el resultado de varias de las herramientas sería muy diferente, y por supuesto, a todos los sujetos de evaluación que han ofrecido desinteresadamente parte de su tiempo en diferentes fases del Trabajo.

En el plano personal, quiero agradecer a mi familia la enorme paciencia que muestran cada día y su apoyo incondicional, que hace que cualquier adversidad parezca salvable. A los profesores que han hecho que hoy sea capaz de presentar este proyecto y a todos mis compañeros de desventuras, tanto en ACM, BlackBerry, Delegación, RITSI, IAESTE…como cada día en clase, gracias. Alexandra, AJ, Sergio, Daniel, Álex…, *was für vier Jahren!*

Y finalmente a ti, lector, espero que disfrutes de las siguientes páginas y te sean de utilidad.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

En el que se resumen los objetivos a cumplir durante el desarrollo del proyecto y los resultados finales.

Los límites físicos de los que adolecen los computadores en la actualidad [1] hacen de la computación distribuida un recurso para incrementar de forma sencilla y económica el rendimiento total de un sistema. Sumado a estos beneficios, el auge de sistemas como teléfonos inteligentes o aplicaciones web ha potenciado el auge de diferentes paradigmas distribuidos en los últimos años.

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.

Otro de los problemas que acompañan a la utilización de este tipo de sistemas reside en las comunicaciones entre nodos. Ejemplos de este tipo de problemas son la identificación de los diferentes componentes y sus propiedades o los canales de comunicación a establecer y su eficiencia y fiabilidad.

A dichas dificultades técnicas se suman otras de carácter económico o logístico. Generalmente el coste de estos equipos es elevado (si bien la relación coste/beneficio es muy atractiva) y presentan una serie de requisitos de espacio, instalación y mantenimiento difíciles de satisfacer por algunas organizaciones.

La presente memoria recoge el proceso de evaluación de diferentes alternativas para la creación de un sistema distribuido que define entre sus objetivos funcionales el carácter multipropósito, la autoconfiguración de los diferentes componentes y la reducción del coste total, utilizando equipos ya existentes en la organización donde se integrará o adquiriendo componentes de bajo coste. Posteriormente se describirán las diferentes etapas de diseño y desarrollo de una propuesta de solución al problema: un sistema formado por dispositivos Raspberry Pi y un conjunto de protocolos, herramientas y servicios para la utilización del mismo como plataforma 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 paquetes *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, análisis 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.

El producto final creado prueba que la utilización de sistemas embebidos de bajo coste, si bien limitados en rendimiento, constituyen una alternativa económica frente a soluciones como el empleo de equipos de escritorio o computadores diseñados de forma específica para este propósito.

Capítulo 2

Motivación y objetivos

En el que se describe la motivación que lleva al desarrollo de este Trabajo y se describen en detalles los objetivos del sistema como un todo.

El presente proyecto responde al interés personal en las áreas de conocimiento relacionadas con la Computación Distribuida. La propuesta final cuenta con un atractivo añadido, que es la utilización de computadores embebidos, generalmente no considerados como integrantes de un sistema distribuido. En diferentes reuniones entre las diferentes partes se terminan de perfilar los diferentes objetivos a completar, entre los que destaca la potencial integración del sistema en asignaturas del plan de estudios del Grado en Ingeniería Informática como herramienta didáctica.

2.1. Objetivos

El sistema creado se inspira en proyectos similares (descritos en 5.5) 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 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.

Partiendo de la premisa de las potenciales ventajas del uso de este tipo de computadores se plantea el sistema definitivo (en 5.6 se detalla el proceso de decisión), no sin antes realizar una etapa de evaluación de las diferentes alternativas.

2.2. Objetivos del sistema

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

2.2.1. Diseño y construcción de la arquitectura física del sistema

Se deberán definir las interconexiones físicas entre los diferentes componentes del sistema, proponer soluciones a los diferentes aspectos físicos del sistema, 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 y las restricciones (tiempo, coste) planteadas.

2.2.2. Arquitectura multipropósito orientada a servicios

El principal propósito del sistema creado será el alojamiento de un conjunto de aplicaciones (servicios) en el mismo, que podrán ser aprovechados por diferentes usuarios para explotar la capacidad de cálculo de las máquinas.

Sin embargo, una de las características de los sistemas similares al que se modela es su orientación a un único fin concreto. Se plantea crear un sistema que rompa con esta tendencia, siendo capaz de dar cabida a diferentes tipos de aplicaciones.

2.2.3. 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 de la misma (personal de administración).

2.2.4. 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 5.1.1). Es necesario por tanto realizar pruebas que evalúen el rendimiento del sistema creado en la misma[1].

A fin de facilitar el uso del sistema por los usuarios finales, se deberán utilizar los recursos ofrecidos por la infraestructura en aquellos casos que sea posible para tareas como la conexión de red, gestión de usuarios[2], etcétera.

2.2.5. 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. Se crearán de aplicaciones y herramientas como alternativas a las utilizadas actualmente, analizando las demandas de los usuarios, así como pruebas de concepto que garanticen que las diferentes tareas encargadas a los sistemas actuales serán integrables en la propuesta de solución.

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

2.2.6. *Zeroconf*

El sistema debe configurarse de forma automática (*zeroconf*) en cualquier tipo de circunstancia (sin que el número de nodos o la configuración de la red sean aspectos relevantes, por ejemplo). Para ello se deberán utilizar o crear una serie de herramientas que

[1] Con el objetivo de facilitar dicha integración, el sistema se desarrolla parcialmente aprovechando la infraestructura presente.

[2] Existe un directorio de autenticación preexistente en el que todo miembro de la organización cuenta con unas credenciales.

posibiliten esta propiedad del sistema. Entre estas herramientas destaca como elemento clave la utilización de un protocolo de descubrimiento de nodos y servicios. Como se destaca en apartados posteriores de la memoria, se ha optado por la creación de este en lugar de utilizar una alternativa de terceros, recibiendo el nombre *MarcoPolo* (ver 7.3.2.1).

2.2.7. Pruebas y evaluación

El desarrollo de las diferentes herramientas *software* se deberá realizar bajo los principios del desarrollo conducido por pruebas (*Test-Driven Development*, ver 3.7.1) como mecanismo para la detección temprana de errores, automatizando este tipo de dinámica en todos los casos en los que sea posible.

Además, y con el 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.

En conclusión, los objetivos del proyecto son los siguientes:

Diseñar un sistema distribuido no jerárquico autoconfigurado e integrable en una infraestructura existente.

Crear un conjunto de herramientas, protocolos y aplicaciones para la explotación de los recursos que ofrece el sistema.

Orientar la explotación del mismo como herramienta de diseño y prueba de algoritmos.

Explotar las posibilidades didácticas del sistema.

Analizar las ventajas e inconvenientes del sistema frente a otras soluciones similares.

Analizar la efectividad de la solución propuesta mediante herramientas de evaluación.

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 una sección final 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 independencia entre sí, y por tanto se deben diseñar e implementar mecanismos de comunicación 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 [2] 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 3.1.7).

Generalmente los sistemas distribuidos son fácilmente escalables si se mantienen estos principios. 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[3] o el *IDL specification language*[4] (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 física exige una serie de consideraciones adicionales, entre las que se encuentra la latencia y fiabilidad de las interconexiones. En arquitecturas constituidas por componentes relativamente pequeños y adaptables, la flexibilidad del sistema se incrementa significativamente, facilitando su escalabilidad. Es necesario para conseguir dicha flexibilidad el desarrollo de interfaces definidas para los componentes de bajo nivel del sistema, así como una descripción de las interacciones entre las diferentes entidades.

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**).

El sistema final consiste en un conjunto de nodos sin una jerarquía definida, eliminando la dependencia de un coordinador. Esto hace del sistema una solución fácilmente escalable.

A la hora de desarrollar las diferentes herramientas creadas se mantienen todas las "transparencias " citadas anteriormente, salvo en aquellos casos que estas comprometan el rendimiento o funcionalidad de la misma. Ejemplos de este tipo de casos son el uso de la IP asignada a un nodo (se rompe la transparencia de localización) como identificador del mismo o la gestión de errores, que en diversas ocasiones puede ser revelada al usuario. Otro ejemplo claro es la transparencia de acceso, pues los usuarios conocen el mecanismo de almacenamiento de ficheros utilizado, dado que cuentan con acceso a través de una interfaz SSH (no obstante, las diferentes aplicaciones proporcionan una interfaz que abstrae este tipo de detalles en todos los casos en los que puede ser necesario)[2].

Como se detalla en la fase de análisis (ver 6) el sistema incluye mecanismos de autoconfiguración que lo hacen adaptable a un gran número de situaciones (variaciones en el espacio de direcciones, nodos añadidos y eliminados arbitrariamente…). La flexibilidad de este es por tanto alta.

[2]En mayor detalle:

- Aquellas herramientas que utilizan MarcoPolo revelarán las direcciones IP al usuario final si es necesario.
- El reemplazo del sistema operativo en herramientas como **marcobootstrap** (ver 7.3.4.1) no garantiza la transparencia de migración.
- La transparencia de replicación puede ser revelada al usuario en casos como la herramienta **deployer** (ver 7.5.2).
- La mayor parte de los usuarios del sistema son desarrolladores, por lo que se tolerará que la transparencia a la hora de gestionar errores se vea reducida, únicamente en aquellos casos en los que el error sea, o bien no recuperable, o bien causado por la acción del usuario, a fin de que pueda establecer mecanismos de recuperación.

3.1.3. *Middleware*

Un *middleware* es una capa *software* que abstrae 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 de cualquier sistema. 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.

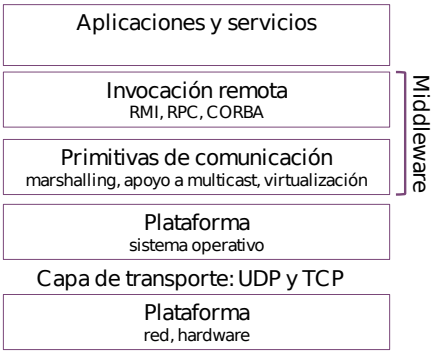

FIGURA 3.1: Ejemplo de una arquitectura que utiliza varias capas *middleware*.

Los diferentes *bindings* de MarcoPolo (ver 7.3.2.5) 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** [5] es considerado un sistema que abstrae los diferentes mecanismos de autenticación y resolución de nombres de la implementación de los mismos puede ser denominado *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.

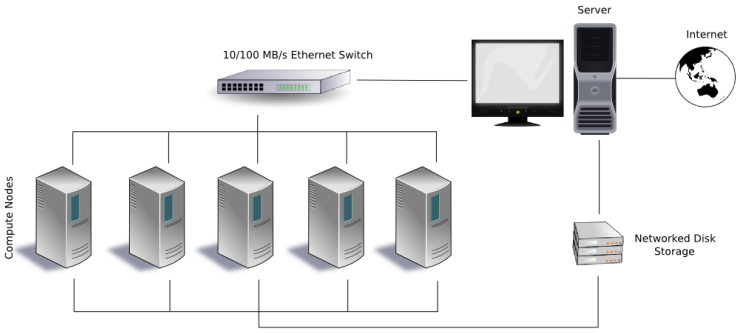

FIGURA 3.2: Esquema de un clúster Beowulf. Este tipo de arquitecturas permiten obtener una gran capacidad de cómputo paralelo con equipos de bajo coste. Generalmente se comportan como una única unidad.

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.

Si bien los nodos principales del sistema final son del mismo tipo y cuentan con el mismo conjunto de herramientas *software*, los nodos secundarios del mismo (encargados de ofrecer servicios auxiliares, ver 7.4) cuentan con una arquitectura *hardware* diferente, así como un sistema operativo diferente. El sistema final puede considerarse un clúster si obviamos estos nodos.

Sin embargo, todas las herramientas han sido diseñadas con el objetivo de ser compatibles con cualquier otro entorno. Han sido integradas en diferentes equipos de escritorio con el sistema GNU/Linux y entornos virtualizados.

En consecuencia, todo el *software* creado es capaz de trabajar en un sistema heterogéneo.

3.1.4.3. 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*.

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).

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 7.3.2.1) para el descubrimiento de nodos y servicios, por lo que la configuración necesaria en el sistema es mínima.

3.1.5. 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 transparencia. 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.

En el sistema final se utilizan gran cantidad de protocolos conocidos (tales como HTTP, JSON, Websockets…) y se ha definido el protocolo de descubrimiento de servicios **MarcoPolo**. Los mecanismos de comunicación entre diferentes componentes de una aplicación son claramente definidos, si bien no de una forma tan exhaustiva como un protocolo.

3.1.6. 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.

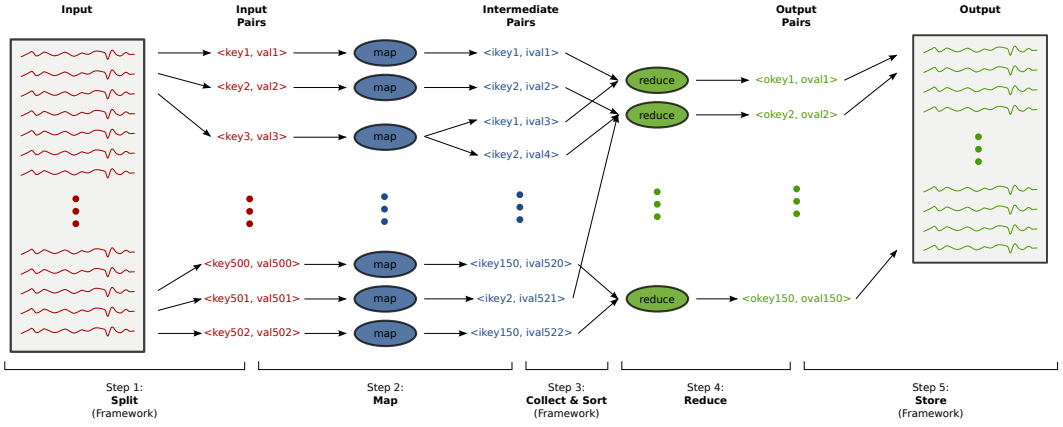

FIGURA 3.3: **MapReduce** se basa en el principio de la distribución de los datos sobre diferentes nodos (*Map*) y la recopilación de los resultados del proceso (*Reduce*).

3.1.7. 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*.

El protocolo **MarcoPolo** resuelve la gran mayoría de los problemas de configuración del sistema sin ningún tipo de ajuste previo. Es capaz además de responder ante alteraciones en la configuración del sistema, como la modificación del esquema de red o la adición de nuevos nodos.

Polousers es una herramienta que, apoyada en **Marcopolo**, gestiona de forma transparente toda la información de los usuarios del sistema.

3.2. Paradigmas de programación

Junto a paradigmas populares como el orientado a objetos y procedimental es necesario mencionar una serie de conceptos menos conocidos:

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 un navegador 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.

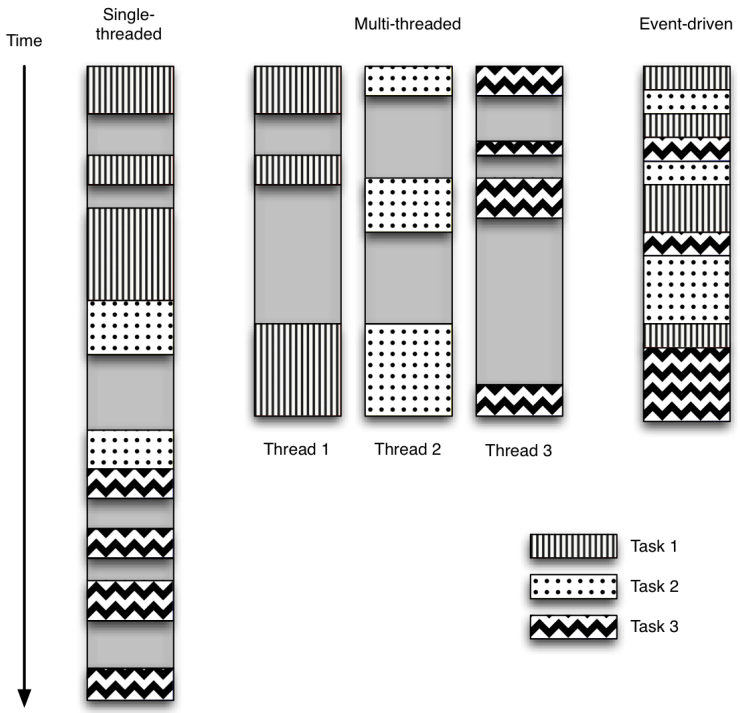

FIGURA 3.5: Comparación de los diferentes modelos de paralelización [6].

En la figura 3.5 se puede observar un claro ejemplo de tres diferentes modelos de programación y sus propiedades. Cada una de las áreas de diferente patrón representan los diferentes instantes de tiempo que un programa ha consumido ciclos de la CPU y el hilo en el que dicha ejecución se ha realizado. En la primera columna únicamente existe un hilo en el cual todos los programas se ejecutan. Esto impide que la CPU pueda alternar entre tareas en momentos en los que se está esperando a una operación que no requiere tiempo de procesado (un ejemplo habitual son las operaciones de lectura/escritura, generalmente lentas). Este modelo no posibilita la paralelización de tareas (como se ve, ninguna tarea comienza hasta que la anterior ha finalizado). En la segunda columna la paralelización es evidente. Sin embargo existe una gran cantidad de tiempo en el que la CPU no está siendo aprovechada. El tercer modelo, dirigido por eventos, ejecuta todas las tareas en un único hilo, pero solapando los instantes en los que la CPU es liberada con otra tarea. El tiempo de ejecución es menor que en el primer modelo y similar al segundo, pero se evitan cambios de contexto y se aumenta el tiempo de utilización de la CPU.

3.2.2.1. Patrón Reactor

El patrón de diseño reactor[7] 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.

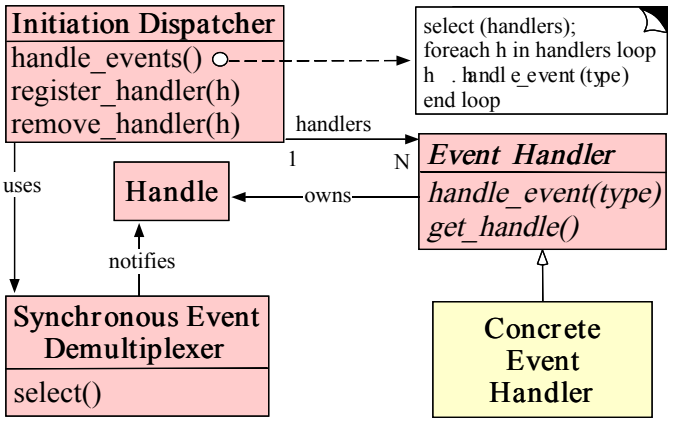

FIGURA 3.6: Diagrama UML del patrón reactor.

La mayoría de las herramientas creadas utilizan mecanismos de sincronización basados en eventos, muchas de ellas utilizando un reactor como demultiplexador. El uso de este paradigma optimiza de forma significativa el rendimiento de cada uno de los nodos del sistema frente a soluciones multiproceso, si bien su uso incrementa la complejidad de las fases de desarrollo del sistema, siendo necesaria en este caso concreto una etapa de formación previa.

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[8] 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 [9]) posibilita la elección de un coordinador en función del cumplimiento de un criterio cuantitativo dado, siendo elegido coordinador aquel que mejor satisfaga dicho criterio.

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[10].

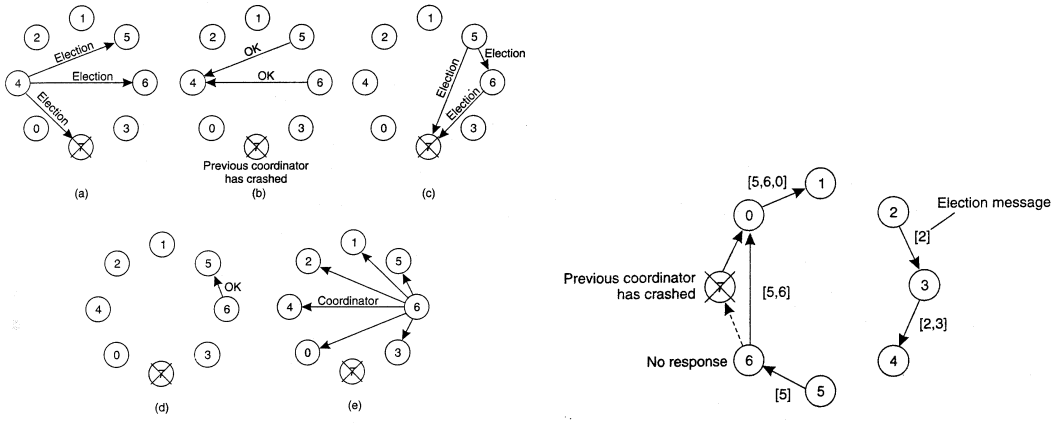

FIGURA 3.7: Secuencia de elección de un coordinador y funcionamiento del algoritmo en anillo

3.4. Seguridad y protocolos criptográficos

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.

3.4.1. 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.

3.4.2.1. X.509

El estándar X.509[11] 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 se define en [12].

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* o activos, como técnicas de *spoofing* o *man-in-the-middle*) 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 realiza un proceso conocido como "apretón de manos" (*handshake*), a través del cual se comprueba la identidad de las partes y se establece un canal seguro.

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. Verificación de doble certificado

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).

Se han establecido las medidas de seguridad oportunas para cada una de las herramientas creadas, como se detalla en los diferentes anexos que describen en detalle cada una de las mismas, garantizando la integridad y veracidad de los datos con mecanismos como sockets TLS o autenticación bidireccional. Con el objetivo de evitar vulnerabilidades se establecen mecanismos de verificación de la información recibida antes de su procesado, entre otras medidas.

3.5. Autenticación y gestión de usuarios

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).

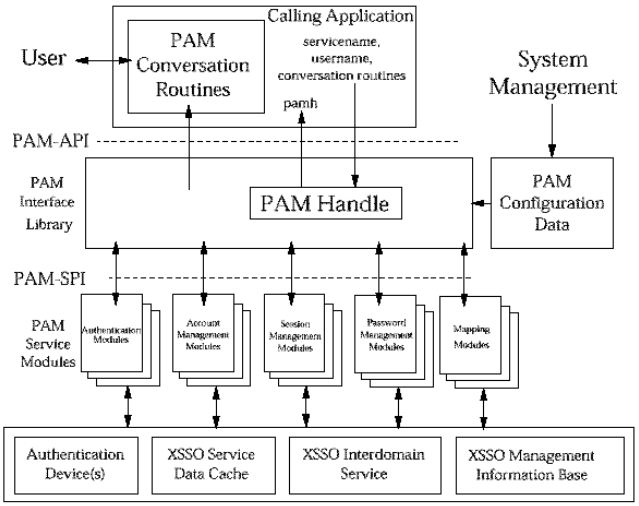

Figure 4-1 PAM Framework

FIGURA 3.8: 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 3.9: 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.

La gestión de usuarios se delega al módulo PAM, utilizando los mecanismos de autenticación que provee en las diferentes aplicaciones del sistema, como en la herramienta **StatusMonitor** (ver 7.1).

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 7.3.3.2).

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 constituyen 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.

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.

3.6.3. WebSockets

El protocolo de comunicación WebSocket [24] permite el establecimiento de conexiones bidireccionales simultáneas (*full-duplex*) entre un cliente y un servidor (al contrario de

la dinámica clásica en una aplicación web, donde toda comunicación es iniciada por el cliente) una vez que el canal es establecido. El modo de operación es muy similar al de la API de sockets de Berkeley y utiliza el protocolo de transporte para establecer la conexión. Utiliza HTTP o HTTPS para la creación del canal y es soportado por todos los navegadores mayoritarios.

El protocolo permite que el servidor envíe eventos asíncronos al cliente, evitando la utilización de técnicas de *polling* y por tanto reduciendo el tráfico de red.

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. Esta información es transmitida sobre diferentes protocolos, destacando multicast UDP (difusión de información), unicast UDP (generalmente respuestas o comandos unidireccionales) y HTTP (o HTTPS). El protocolo WebSocket se utiliza como mecanismo principal de comunicación en diferentes secciones de **deployer** (7.5.2).

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.

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 un paquete *software* consiste en la abstracción de la plataforma física sobre la que este 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 y 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 sencillez distribución, 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 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 compilación 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*

Un juego 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].

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** (ARM versión 7 con instrucciones de punto flotante en hardware) utilizando la herramienta *crosstool-ng*[4], consiguiendo optimizar de forma significativa el tiempo de compilación (ver 7.4.1.4).También es posible crear con pequeños ajustes a la configuración una versión para la arquitectura ARMv6hf. Estas cadenas de herramientas se han utilizado para la creación de un pequeño entorno de desarrollo con Eclipse para Raspberry Pi (ver anexo “Configuración de un sistema de compilación multiplataforma con distcc y aplicación de este en un entorno distribuido”).

[3] Más información sobre este tipo de técnica: http://elinux.org/Toolchains
[4] http://crosstool-ng.org

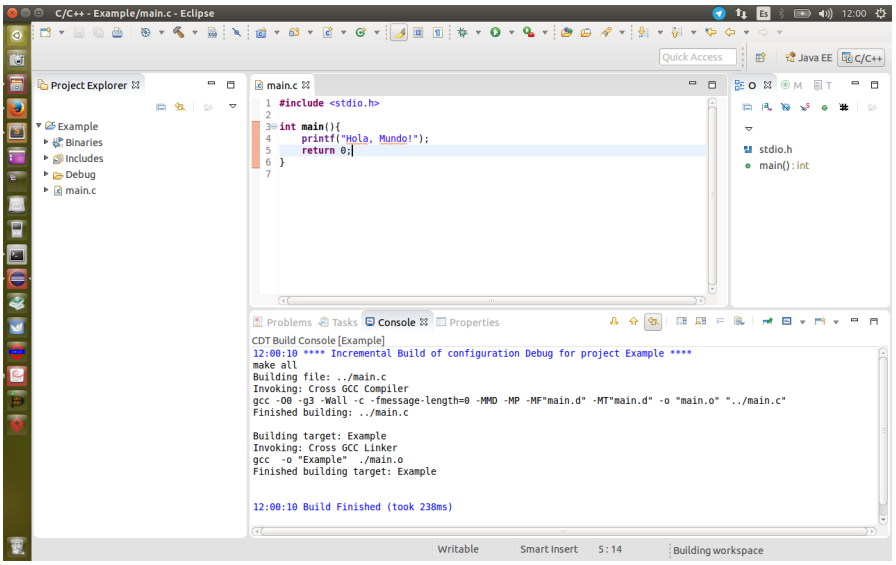

FIGURA 3.10: Compilación cruzada utilizando Eclipse

El uso de **crosstool-ng** permite personalizar la cadena de herramientas creada a cambio de un aumento en la complejidad de uso. Dicha personalización hace que esta alternativa sea la elegida en lugar de las utilidades proporcionadas por los desarrolladores de la plataforma **Raspberry Pi**[5] o el *flag* `ARCH` de gcc.

[5] https://github.com/raspberrypi/tools

Capítulo 4

Herramientas y técnicas

En el que se detalla el análisis de diferentes alternativas de aplicaciones, herramientas y técnicas para su uso en la construcción del sistema final y las decisiones finales tomadas.

4.1. Herramientas utilizadas para la creación del sistema

4.1.1. Lenguajes de programación

Python se ha elegido como lenguaje principal de desarrollo. 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. Este lenguaje 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.

La gran cantidad de bibliotecas disponibles para el lenguaje y su facilidad de uso, así como el hecho de que la fundación Raspberry Pi propicie su uso en los equipos que produce constituyen ventajas competitivas sobre el resto de alternativas.

Junto a Python se han utilizado varios lenguajes de forma complementaria.

Nombre	Características	Ventajas	Inconvenientes	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	Muy eficiente e integrable en cualquier contexto	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 de Python.
C++	Orientado a objetos	Gran rendimiento, acceso a todas las características de C	No es portable fácilmente en algunos casos	Se han creado los *bindings* de MarcoPolo para este lenguaje, así como las herramientas **marcobootstrap**
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** y se ha creado un *binding* de MarcoPolo para el lenguaje.
Bash	Lenguaje de comandos utilizado en sistemas UNIX	Interpretado, portable, sencillo de utilizar.	Es el lenguaje idóneo para una serie concreta de aplicaciones, pero su propósito específico limita su uso más allá de dicho conjunto.	Utilizado en todos los *scripts* de gestión de *daemons* de systemv, y de arranque en **marcobootstrap**, así como herramienta de gestión en varias aplicaciones más.
Perl	Multiplataforma y multiparadigma	Portable, sencillo de utilizar, diseñado para la administración de sistemas	El uso de Perl como lenguaje de programación principal puede dificultar la realización de una serie de tareas clave.	Ninguna

Cuadro 4.1: Lenguajes de programación evaluados para su utilización en el sistema y uso final

Otros lenguajes

Todas las interfaces web han sido programadas utilizando HTML, CSS y JavaScript en el lado del cliente. Dicha combinación evita la dependencia con cualquier herramienta no incluida por defecto en la totalidad de los navegadores mayoritarios (tales como Flash, ActiveX…).

4.2. Herramientas utilizadas para la creación de *software*

4.2.1. Twisted

Twisted [1] 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** (ver 3.2.2.1), que se basa en la gestión de diferentes eventos, su demultiplexación y el envío a los manejadores apropiados de forma síncrona (ver 3.2.1).

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 7.3.2.1) y parte de la herramienta **PoloUsers** (ver 4.9.1).

Esta herramienta es elegida sobre otras alternativas analizadas:

- asyncore[2] se plantea como la primera alternativa y es descartado por la incapacidad de desarrollar un prototipo funcional.
- El bucle de eventos `io_loop` de Tornado[3], si bien no diseñado específicamente para este propósito se considera como una alternativa viable. Sin embargo, la carencia de las facilidades para manejo de una aplicación en red (si bien Tornado es un servidor web, no está diseñado para implementar operaciones a bajo nivel, apoyándose siempre en protocolos como HTTP o WebSocket) hace que no sea una alternativa viable.

[1] https://twistedmatrix.com/
[2] https://docs.python.org/3.5/library/asyncore.html
[3] http://www.tornadoweb.org/en/stable/ioloop.html

Twisted es elegido por la gran cantidad de protocolos que soporta y la posibilidad de manipular la pila de protocolos desde el nivel de red.

4.2.2. 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.

Si bien existen alternativas como node.js que siguen paradigmas similares, se decide utilizar Tornado debido a su implementación en Python, lenguaje conocido previamente, de alto rendimiento y fácil de utilizar para tareas que dependan altamente de la interacción con el sistema operativo y aplicaciones locales. Se descarta también utilizar herramientas web escritos en Python que por su arquitectura (basada en múltiples procesos o hilos) consuman más recursos, como el *framework* Django.

4.2.3. Websockets

El protocolo WebSocket [24] 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, en contraste con estrategias como el *polling*.

4.3. Seguridad

4.3.1. OpenSSL

OpenSSL[4] es la implementación de código abierto más popular de los protocolos SSL y TSL. En el sistema se utiliza de forma intensiva para garantizar la confidencialidad de

[4] https://www.openssl.org/

las transmisiones entre partes así como para verificar la identidad en ambos lados de un canal de comunicación. La biblioteca proporciona *bindings* a C, C++, Java y Python, por lo que su integración en cualquiera de las herramientas creadas ha sido trivial.

4.3.2. Hadoop

Hadoop[5] es una herramienta diseñada para el procesamiento de grandes cantidades de datos de forma distribuida y el almacenamiento distribuido de información. La biblioteca ofrece además un gran nivel de fiabilidad mediante una serie de mecanismos de detección y gestión de errores. Es una de las herramientas de gestión de datos más popular actualmente.

En una de las iteraciones del ciclo de desarrollo se instaló parcialmente una instancia de Hadoop en el sistema. Sin embargo se descartó su continuación al priorizar una serie de tareas de mayor importancia. No obstante, se contempla como línea de trabajo futuro, y teóricamente es integrable con **MarcoPolo** a través de `marcomanager`.

No se ha realizado evaluación de otras alternativas para el proceso de grandes cantidades de datos, como **Apache Spark** [6].

4.4. Aplicaciones distribuidas

4.4.1. *Message Passing Interface*

La necesidad de una herramienta de comunicación independiente de la plataforma derivó en la especificación del estándar MPI [25], 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++ y Fortran, así como mecanismos para ser integrado con Python, entre muchos otros lenguajes.

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**[7] (la implementación original) u **OpenMPI**[8] (presente en la mayoría de supercomputadores),

[5] https://hadoop.apache.org/
[6] https://spark.apache.org/
[7] https://www.mpich.org/
[8] http://www.open-mpi.org/

de tipo libre, o implementaciones propietarias tales como IBM MPI, Intel MPI, Cray MPI o Microsoft 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 7.3.2.6). Además se han creado herramientas accesorias para facilitar varias tareas generalmente necesarias durante el desarrollo con la biblioteca (ver 7.3.2.6).

La popularidad de MPI sobre otras herramientas similares como **PVM** (*Parallel Virtual Machine*) [9] hace que la cantidad de soporte para las placas Raspberry sea mayor. Esta circunstancia junto al hecho de que MPI es utilizado en la asignatura Arquitectura de Computadores (y por tanto se cuenta con experiencia de uso, y el desarrollo del sistema relativo a este tipo de aplicaciones podría utilizarse en la asignatura) hace que MPI sea la alternativa elegida sin evaluación previa.

Durante la etapa de desarrollo del sistema se han utilizado las dos vertientes libres más populares, **MPICH** y **OpenMPI**, siendo esta última la que se incluye en la versión final.

4.4.1.1. *Raspberry Pi Planet Simulator Cluster*

Este proyecto implementa un simulador del clima terráqueo, permitiendo alterar diferentes características del mismo y analizar el resultado. Se implementa sobre MPI y está programado en el lenguaje **Fortran**[10].

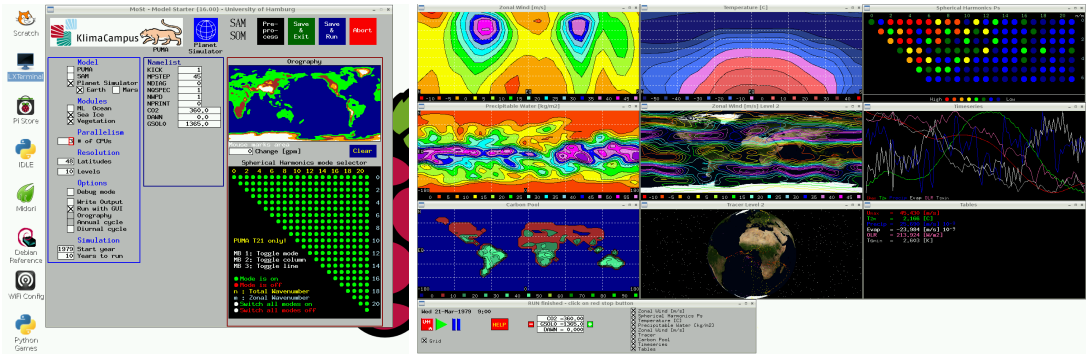

FIGURA 4.1: Interfaz de control del *Planet Simulator* y vista de ejecución de una simulación

[9] http://www.csm.ornl.gov/pvm/

[10] http://econnexus.org/projects/the-distributed-arctic-sea-ice-model/raspberry-pi-planet-simulator-cluster/

Este proyecto ha sido evaluado con el objetivo de integrarlo en el sistema. Sin embargo, el programa depende considerablemente de la interfaz gráfica del mismo, inaccesible desde cualquiera de los nodos del sistema. A fin de solventar este problema se plantea el uso de un equipo de escritorio que cree los ficheros de configuración y los distribuya entre los nodos. Esta posibilidad se debe descartar finalmente, dado que el simulador realiza la compilación del programa con los parámetros fijados en la interfaz gráfica. Dado que los binarios creados para la arquitectura de un equipo de escritorio (x86, x86-64) son incompatibles con el *hardware* de los nodos (arquitectura ARM) y hay tareas con mayor prioridad, se descarta esta tarea de forma indefinida, siendo finalmente no incluido en la versión final del sistema.

4.4.2. Tomcat

Las prácticas de la asignatura Sistemas Distribuidos se realizan sobre el contenedor de servicios Tomcat y el *framework* Glassfish [11] para la creación de APIs **REST**. Por ello el sistema ofrece instancias de Tomcat a todos los usuarios sin que estos deban realizar ningún tipo de configuración. Además, se habilita el uso de Tomcat en herramientas como **deployer** para facilitar el desarrollo de las prácticas.

Se plantea el análisis de alternativas para la creación de APIs **REST** como el *framework* **Flask**.

4.5. Herramientas de virtualización

4.5.1. QEMU

QEMU[12] es un emulador y virtualizador de código abierto compatible con un gran abanico de arquitecturas y sistemas operativos diferentes, siendo compatible con hipervisores como Xen o KVM. En el sistema se utiliza como la primera propuesta de optimización del rendimiento del sistema en tareas como la compilación de código fuente.

QEMU es popular a la hora de virtualizar instancias de sistemas operativos para la arquitectura ARM, y existe documentación sobre la virtualización de sistemas como **Raspbian** o **Arch Linux ARM**, por lo que no se realiza una evaluación de otras alternativas.

[11] https://glassfish.java.net/
[12] http://qemu.org

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

4.6.1. Git

Git es un sistema de control de versiones capaz de gestionar proyectos de cualquier escala, diseñado para flujos de trabajo distribuidos. Git permite realizar operaciones de reversión de cambios, bifurcaciones y uniones de flujos de desarrollo y gestión de varias copias independientes sin conflictos. Es una de las herramientas de gestión de código más utilizadas, siendo diseñada originalmente para la coordinación en el desarrollo del núcleo Linux.

Las diferentes herramientas que componen el sistema son creadas en repositorios independientes de código. Dicho código es almacenado en un servidor con el objetivo de facilitar la movilidad del código entre las diferentes máquinas que componen el sistema y a modo de copia de seguridad. Se sigue un modelo de desarrollo basado en ramas que representan características a añadir a la versión estable, revisión de código, documentación o mantenimiento y solución a *bugs*.

Todos los repositorios de código originales se listan en el apéndice D.

Dado que se cuenta con bastante experiencia en el uso de **git** no se analizan alternativas como **subversion** o **mercurial**.

4.6.2. Redmine

Redmine es una herramienta de gestión de proyectos basada en web que permite a un equipo mantener un registro de todo el trabajo realizado y planificado en un proyecto, con una serie de herramientas como diagramas de Gantt, Wiki o integración con sistemas de control de versiones.

El proyecto cuenta con una instancia de Redmine alojada en http://redmine.martinarroyo.net/projects/tfg. En dicha instancia se han registrado todos los avances en el desarrollo del proyecto desde las fases iniciales del mismo.

4.6.3. 2to3, 3to2

Las versiones del lenguaje Python 2 y 3 son incompatibles entre sí. Sin embargo, las diferencias entre ambas versiones radican en una serie de variaciones sintácticas, nombres de tipos y localización de las bibliotecas estándar, por lo que escribiendo código que tenga

en cuenta dichas variaciones, bien incluyendo sentencias condicionales en función de la versión o bien escribiendo código ambivalente es posible generar código compatible.

2to3 y 3to2 son herramientas que ayudan al programador en la verificación de la compatibilidad entre versiones, generando una lista de modificaciones que posibilitan la ejecución el código en otra versión. Utilizando ambas herramientas es posible crear código ambivalente de forma sencilla. Existen además guías y otras herramientas que facilitan este objetivo[13].

4.6.4. Pylint

Pylint es una herramienta de verificación de código, que evalúa la calidad de un fichero siguiendo una serie de criterios tales como la presencia de errores sintácticos, sangrado del código, convenciones de nombrado y de estilo, errores en la importación de paquetes, etcétera. Pylint incluye además la herramienta **pyreverse**, útil en la generación de diagramas UML.

4.6.5. Desarrollo dirigido por pruebas: Unittest, CppUnit, Trial

Uno de los mecanismos para la detección temprana de errores es el desarrollo dirigido por pruebas (ver 3.7.1). En el proyecto se utilizan diferentes *frameworks* para cada una de las aplicaciones y bibliotecas creadas. Los tests unitarios se incluyen en cada uno de los paquetes para poder ser ejecutados por cualquier usuario en caso de que lo estime oportuno.

4.7. Herramientas de modelado

4.7.1. Visual Paradigm

Junto a **pyreverse** se utiliza la aplicación Visual Paradigm durante las etapas de documentación y modelado de las diferentes aplicaciones, en concreto, las herramientas de creación de diagramas UML y su generación automática a través del análisis del código fuente.

[13] https://docs.python.org/3/howto/pyporting.html

4.8. Herramientas utilizadas para la documentación del proyecto

4.8.1. LaTeX

El sistema de composición de textos LaTeX es utilizado para crear todos los documentos incluidos en el desarrollo del sistema. Se utiliza el motor XeLaTeX para el compilado de los ficheros, debido a su mayor variedad de fuentes y la utilización por defecto de la codificación UTF-8 (útil en idiomas que utilizan un alfabeto diferente al del inglés).

Se utiliza BibTeX como gestor de la bibliografía en todos los documentos producidos. Las diferentes entradas de la base de datos han sido creadas manualmente o proceden de fuentes como la biblioteca digital de **ACM**[14], la biblioteca **IEEE Xplore**[15], **CiteSheer**[16] o el trabajo de documentación de las **RFC** del profesor Roland Bless[17], entre otros.

4.8.2. Sphinx

Sphinx es un sistema de creación y generación de documentación capaz de crear documentos en diferentes formatos (HTML, LaTeX, ePub…) a partir de una serie de archivos en el formato reStructuredText. Es además capaz de crear documentación sobre código Python (lenguaje para el que la herramienta fue creada) a partir de los comentarios presentes en el código (conocidos como **docstrings**). Junto con Doxygen, se ha conseguido documentar código en C, C++, Java, bash y JavaScript. Soporta además referencias cruzadas a diferentes proyectos creados con esta herramienta, muy útil en este caso concreto, donde se cuenta con un número de documentos independientes muy alto que se referencian entre ellos.

Los resultados generados por la herramienta son incluidos como documentación técnica de cada una de las herramientas, y están disponibles en .

4.8.3. Doxygen

Doxygen es un sistema similar a Sphinx utilizado en proyectos escritos en C, C++ y Java entre otros muchos lenguajes. Constituye el estándar *de facto* para la generación de documentación.

[14] http://dl.acm.org
[15] http://ieeexplore.ieee.org/
[16] http://citeseerx.ist.psu.edu/
[17] http://tm.uka.de/ bless/bibrfcindex.html

En el proyecto Doxygen es utilizado para documentar aquellas partes del proyecto que Sphinx no puede procesar (actualmente el soporte de dicha herramienta se limita a Python). Posteriormente la documentación de ambas herramientas se combina mediante ficheros XML generados por Doxygen que Sphinx puede procesar.

4.9. Herramientas para la gestión de usuarios

4.9.1. PAM, LDAP

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**[18] con la información de los usuarios. Mediante la configuración del paquete **LDAP** es posible acceder 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**7.3.3.2. Este módulo, en combinación con un *daemon* que realiza la gestión de las diferentes peticiones realizadas conforman la herramienta polousers (ver anexo técnico correspondiente).

4.10. Herramientas para la optimización del rendimiento

4.10.1. Distcc

Distcc[19] es la herramienta utilizada para el desarrollo del compilador distribuido. Se basa en una arquitectura cliente-servidor donde los trabajos de compilación pueden ser repartidos entre varios servidores para reducir el tiempo total de compilación, implementando mecanismos de verificación y balance de carga.

Junto con **crosstool-ng** conforma el compilador distribuido del sistema (ver 3.8.3.1).

4.10.2. SSH-HPN

Una de las características de OpenSSH es la ejecución de todas las tareas en un único proceso y por tanto, en un único núcleo, constituyendo un cuello de botella que se

[18] ldap1.cie.aulas.usal.es
[19] https://code.google.com/p/distcc/

hace notable en computadores de bajas prestaciones, como el sistema a modelar que sin embargo, cuentan con un procesador multinúcleo.

Con el objetivo de superar este límite nace SSH-HPN[20], 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 una versión parcheada de SSH probada en la instalación de Arch Linux utilizada con el código fuente ya preparado para trabajar en la arquitectura ARM utilizándose en lugar del paquete **OpenSSH** original.

4.11. Metodología de desarrollo

Una metodología de desarrollo reúne el conjunto de procesos y técnicas que se emplean en la construcción de un producto *software*. La metodología debe ser elegida cuidadosamente en virtud de los objetivos y restricciones del proyecto, pues tiene el potencial de condicionar su buen devenir o actuar en detrimento del mismo.

Existen diferentes metodologías que satisfacen un conjunto diferente de demandas y son aptas para un tipo de proyecto, como el Proceso Unificado (cíclico, conducido por casos de uso), procesos ágiles (desarrollo incremental, tolerancia a cambios inesperados) o los modelos tradicionales (modelo lineal, en V, orientado a prototipos…).

En el caso del presente proyecto, se debe construir un conjunto de herramientas *software* sin contar con abundante experiencia en el desarrollo de sistemas de este tipo, por lo que el grado de incertidumbre es muy alto. Es por ello que la metodología elegida debe ser capaz de lidiar con situaciones difíciles de predecir, cambios continuos en los requisitos definidos y poder responder ante problemas como la inviabilidad de una propuesta de solución, detectando dicha circunstancia de forma prematura. Se apuesta por un modelo ágil apoyado en prototipos, descrito en 5.9.1.

[20] http://www.psc.edu/index.php/hpn-ssh

Capítulo 5

Dominio del problema

En el que se detalla la etapa de identificación de las características de la infraestructura donde el sistema se integrará, los diferentes usuarios que interactuarán con este y sus necesidades, y las propuestas de solución actuales relativas a la construcción de un sistema similar al planteado.

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 entidad 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 dificultan el desarrollo de este sistema y 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 computadores 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 como soporte a la toma de una decisión final, y una descripción de los diferentes requisitos del sistema a nivel global. Se omite la definición de objetivos, ya descrita en 2.1.

5.1. Definición del dominio del problema

5.1.1. Sistema actual: Infraestructura de la Facultad de Ciencias

El sistema se ubica en una Facultad universitaria con 1.463 alumnos, de los cuales 562 son estudiantes de titulaciones relacionadas con las Ciencias de la Computación [26]. Estas titulaciones cuentan con varias asignaturas relacionadas con el áreas de conocimiento Computación Distribuida, en particular **Arquitectura de Computadores** y **Sistemas Distribuidos** [27]. La Facultad cuenta con varias aulas y laboratorios de informática donde los alumnos disponen de las herramientas necesarias para realizar los ejercicios y prácticas asignadas. Dichos espacios permiten utilizar cualquier equipo como nodo, dado que se integran en la misma red, siendo factible la comunicación directa entre equipos situados en diferentes aulas o incluso edificios. Todos los equipos cuentan con una conexión de red cableada capaz de soportar teóricamente transferencias de hasta 100 Mb/s de forma bidireccional, compartiendo un mismo espacio de direcciones de red (no existen *routers* entre diferentes secciones de este segmento de la red de la Universidad). La gestión de credenciales de autenticación se delega a un directorio **LDAP** (*Lightweight Directory Access Protocol*) [28],y se cuenta con 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 a través de los protocolos **NFS** y **Samba**. Todos estos datos han sido extraídos de diferentes entrevistas con el Administrador de la infraestructura.

La mayoría de las prácticas asignadas 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.

5.1.1.1. Problemas conocidos

Si bien la infraestructura existente es capaz de proveer a los estudiantes de los recursos necesarios para la realización de los diferentes ejercicios, se identifican una serie de problemas:

- Cada grupo de alumnos necesita tres estaciones de trabajo para poder realizar algunos de los ejercicios propuestos, y acceso físico a los mismos, consumiendo una gran cantidad de espacio.
- El servidor **NFS** constituye un "cuello de botella", pues todos los alumnos acceden a él de forma intensiva, causando una excesiva carga de trabajo por el gran número de peticiones al mismo.
- Existen una serie de problemas con las herramientas de desarrollo utilizadas actualmente, como la dificultad de configuración o la ubicación e identificación de recursos.

5.1.1.2. Rendimiento

Uno de los problemas actuales que la infraestructura presenta es la sobrecarga de los diferentes recursos centralizados, como son el servidor LDAP y el sistema de almacenamiento NFS/Samba.

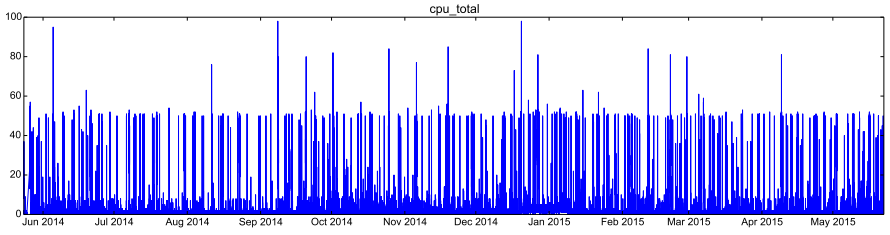

FIGURA 5.1: Porcentaje de CPU utilizada del servidor NFS

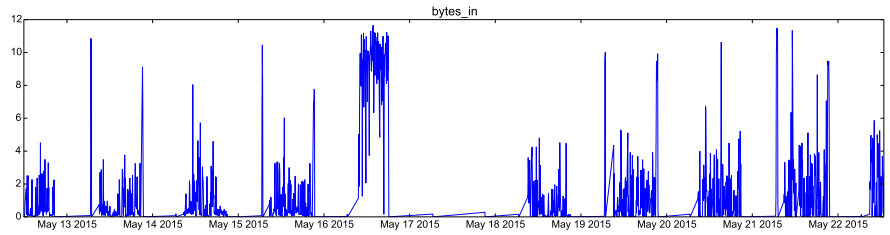

FIGURA 5.2: Datos de entrada al servidor durante un periodo de 10 días

.

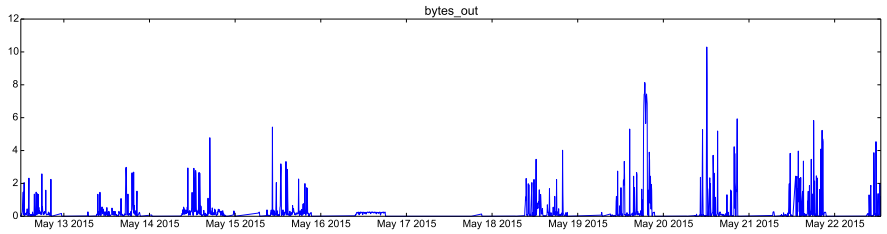

FIGURA 5.3: Datos de salida durante un periodo de 10 días

5.2. Identificación de *stakeholders*

Un *stakeholder* es cualquier grupo o individuo que afecta o es afectado por la consecución de los objetivos de un proyecto marcados por una organización, así como un particular interés en el devenir del mismo. Claros ejemplos de este tipo de entidades son los diferentes usuarios finales del sistema, potenciales clientes o el equipo de desarrollo, entre muchos otros.

En el proyecto en cuestión se identifican los siguientes *stakeholders* y su motivación:

- Equipo de desarrollo. Su motivación es la consecución de todos los objetivos marcados en el sistema. Dentro de este grupo existen dos categorías: los desarrolladores y los supervisores del proyecto.
- Estudiantes de tercer y cuarto curso del Grado en Ingeniería Informática, que podrán utilizar el sistema como herramienta didáctica.
- Profesores de las asignaturas Arquitectura de Computadores y Sistemas Distribuidos, que aprovecharán el sistema final en dichas asignaturas.
- Administradores del sistema, que deberán hacerse cargo del mantenimiento del sistema a largo plazo.

5.2.1. Identificación de las necesidades de cada *stakeholder*

Basándose en la motivación de cada parte es posible definir las demandas de cada una de las partes. Otros mecanismos, como la realización de entrevistas u observaciones permiten complementan dicho proceso.

5.2.1.1. Equipo de desarrollo

- Acceso a diferentes recursos gestionados por la infraestructura.

- Acceso a herramientas que permitan construir el *software* y *hardware* a desarrollar.

5.2.1.2. Alumnos

- Entorno de trabajo intuitivo y documentado 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.

5.2.1.3. 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.
- Instalación y configuración simple.

5.2.1.4. Administradores

- Sistema integrable en la infraestructura actual cuyo mantenimiento sea sencillo y cuyo enfoque garantice la escalabilidad y su durabilidad. Documentación extensa sobre el funcionamiento interno del sistema.
- Instalación y configuración simple.
- Mantenimiento sencillo.

5.3. 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.

5.4. Identificación de requisitos

Los requisitos relativos a cada herramienta son detallados en el anexo correspondientes (ver apéndice A), sirviendo esta sección para aportar una perspectiva a alto nivel de los diferentes requisitos. Es por ello que los requisitos funcionales, vinculados fuertemente a la interacción del usuario con el sistema, son recogidos únicamente en este documento.

5.4.1. Requisitos de almacenamiento de la información

Se identifican los siguientes requisitos de información a alto nivel.

IRQ-1	Gestión de usuarios
Versión	1.1
Autores	Diego Martín
Fuentes	Análisis preliminar del sistema. Reuniones de equipo.
Objetivos asociados	2.2.3, 2.2.7
Requisitos asociados	
Descripción	El sistema deberá consultar una fuente de datos (bien gestionada por el mismo o mantenida por un tercero) para realizar todas las operaciones de autenticación de los usuarios.
Datos específicos	Los propios de un sistema UNIX (UID, GID, nombre de usuario, grupos, directorio `$HOME`).
Tiempo de vida	Permanente
Ocurrencias simultáneas	Se plantea la existencia de una única fuente de datos, consultada tantas veces como un usuario inicie sesión o realice cualquier otra operación en la que esta información deba ser verificada.
Importancia	Alta
Urgencia	Alta
Estado	Completo
Estabilidad	Estable
Comentarios	La versión 1.1 considera una fuente de datos de un tercero para la gestión de los usuarios

Cuadro 5.1: IRQ-1: Gestión de usuarios

IRQ-2	*Logs* del sistema
Versión	1
Autores	Diego Martín
Fuentes	Análisis preliminar del sistema. Reuniones de equipo.
Objetivos asociados	2.2.3, 2.2.4
Requisitos asociados	NFR-1
Descripción	Se deberán gestionar diferentes ficheros que sirvan como registro de todas las operaciones realizadas por el sistema con el objetivo de poder realizar operaciones de análisis o detección de errores posteriormente.
Datos específicos	Operaciones realizadas en las diferentes herramientas, incluyendo parámetros de operación, usuarios, resultados, condiciones irregulares...
Tiempo de vida	Se mantendrán estos archivos de forma permanente.
Ocurrencias simultáneas	Se plantea el uso de un fichero por cada aplicación.
Importancia	Alta
Urgencia	Media
Estado	Completo
Estabilidad	Estable
Comentarios	

CUADRO 5.2: IRQ-2: *Logs* del sistema

IRQ-3	Ficheros de configuración
Versión	1
Autores	Diego Martín
Fuentes	Etapas de desarrollo
Objetivos asociados	2.2.3
Requisitos asociados	NFR-1
Descripción	La configuración de todas las aplicaciones se realizará a través de ficheros que incluyan todos los parámetros de interés.
Datos específicos	Cualquier valor modificable que altere el comportamiento de un programa (puerto de operación, directorios auxiliares…
Tiempo de vida	Permanente
Ocurrencias simultáneas	Se plantea el uso de un fichero por cada aplicación.
Importancia	Alta
Urgencia	Media
Estado	Completo
Estabilidad	Estable
Comentarios	

Cuadro 5.3: IRQ-3: Ficheros de configuración

IRQ-4	Información aportada por los usuarios y de gestión
Versión	1
Autores	Diego Martín
Fuentes	Fases de desarrollo del sistema
Objetivos asociados	2.2.2, 2.2.3, 2.2.5
Requisitos asociados	NFR-5
Descripción	Se utilizarán diferentes gestores de bases de datos para el almacenamiento de diferentes datos manejados por el sistema con el objetivo de facilitar el acceso a los mismos y su persistencia.
Datos específicos	Se plantea el almacenamiento de información aportada por el usuario, datos internos del sistema, etcétera
Tiempo de vida	Generalmente, los datos serán almacenados hasta que no se consideren de utilidad o un usuario o administrador decida eliminarlos.
Ocurrencias simultáneas	Se plantea el uso de una base de datos para cada tarea.
Importancia	Media
Urgencia	Media
Estado	Completo
Estabilidad	Estable
Comentarios	

Cuadro 5.4: IRQ-4: Información aportada por los usuarios y de gestión

5.4.2. Identificación de requisitos no funcionales

NFR-1	Mantenimiento y robustez
Versión	1
Autores	Diego Martín
Fuentes	Análisis preliminar del sistema
Objetivos asociados	2.2.1, 2.2.2, 2.2.4
Requisitos asociados	IRQ-1, IRQ-2, IRQ-3, IRQ-4
Descripción	El *software* debe ser mantenible y robusto, 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. El mantenimiento del sistema ha de ser lo más sencillo posible, utilizando mecanismos como diferentes ficheros de configuración y de registro para la realización de las diferentes operaciones de administración.
Importancia	Alta
Urgencia	Alta
Estado	Completo
Estabilidad	Estable
Comentarios	

Cuadro 5.5: NFR-1: Mantenimiento y robustez

NFR-2	Coste de desarrollo
Versión	1
Autores	Diego Martín
Fuentes	Administración de la organización
Objetivos asociados	2.2.1, 2.2.2
Requisitos asociados	
Descripción	El coste de desarrollo no debe superar un total de 400 €.
Importancia	Alta
Urgencia	Alta
Estado	Completo
Estabilidad	Estable
Comentarios	

Cuadro 5.6: NFR-2: Coste de desarrollo

NFR-3	Definición de los protocolos de comunicación
Versión	1
Autores	Diego Martín
Fuentes	Análisis preliminar
Objetivos asociados	2.2.2
Requisitos asociados	
Descripción	Los diferentes protocolos utilizados o creados para el sistema deben ser públicos y extensibles a diferentes paradigmas de utilización y tecnologías que los implementen.
Importancia	Alta
Urgencia	Alta
Estado	Completo
Estabilidad	Estable
Comentarios	

Cuadro 5.7: NFR-3: Definición de los protocolos de comunicación

NFR-4	Transparencia
Versión	1
Autores	Diego Martín
Fuentes	Análisis preliminar
Objetivos asociados	2.2.2, 2.2.3, 2.2.4, 2.2.6
Requisitos asociados	
Descripción	El sistema deberá cumplir todas las transparencias definidas en 3.1 en todas aquellas ocasiones en las que no se comprometa el funcionamiento del sistema.
Importancia	Media
Urgencia	Media
Estado	Completo
Estabilidad	Estable
Comentarios	

Cuadro 5.8: NFR-4: Transparencia

NFR-5	Compatibilidad con prácticas y otros ejercicios didácticos
Versión	1
Autores	Diego Martín
Fuentes	Reuniones de equipo
Objetivos asociados	2.2.5
Requisitos asociados	IRQ-1, IRQ-4
Descripción	El sistema deberá ser compatible los ejercicios desarrollados en las asignaturas **Arquitectura de Computadores** y en especial **Sistemas distribuidos**
Importancia	Alta
Urgencia	Media
Estado	Completo
Estabilidad	Estable
Comentarios	

Cuadro 5.9: NFR-5: Compatibilidad con prácticas y otros ejercicios didácticos

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

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

5.5.1. 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.

5.5.1.1. 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[29] 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

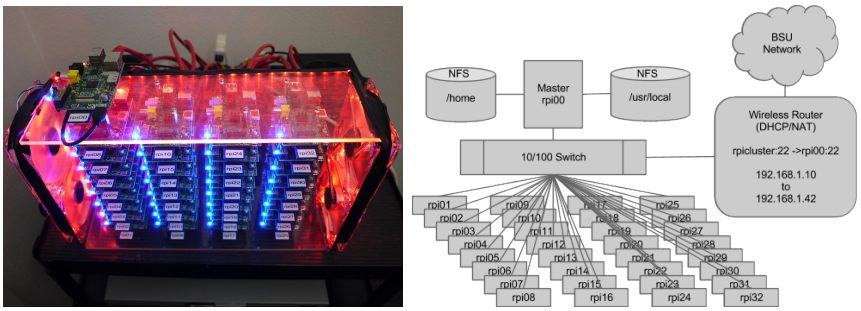

FIGURA 5.4: Vista general y estructura del sistema (Fuente: Joshua Kiepert)

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

5.5.1.2. 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[30]. Se compone del conjunto de nodos *Rasbperry Pi* y los mecanismos de red y alimentación que interconectan y proveen de energía a los mismos.

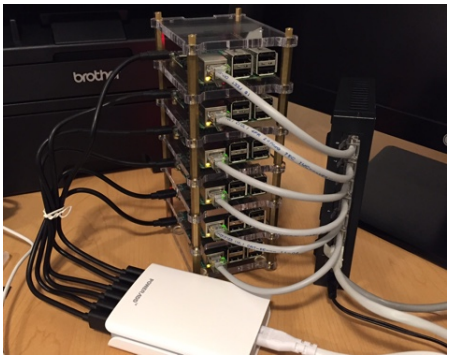

FIGURA 5.5: El *Dramble* en ejecución

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.

5.5.1.3. Bramble (GCHQ)

El organismo gubernamental *Government Communication Headquarters*, agencia de inteligencia del Gobierno Británico, presentó en la *Big Bang Fair* de 2015 un proyecto educativo que combina 66 placas *Raspberry Pi* en un clúster jerárquico con 8 grupos de

[2] drupal.org

8 nodos, cada uno de ellos con un coordinador. El cableado se reduce gracias al uso de la tecnología **PoE** (*Power over Ethernet*), que provee de conexión a la redes eléctrica y de interconexión a través de un único canal físico. Cada **Raspberry** cuenta además con un conjunto de elementos adicionales, como un reloj de tiempo real, disco duro externo, cámara, o punto de acceso **Wi-Fi** [31].

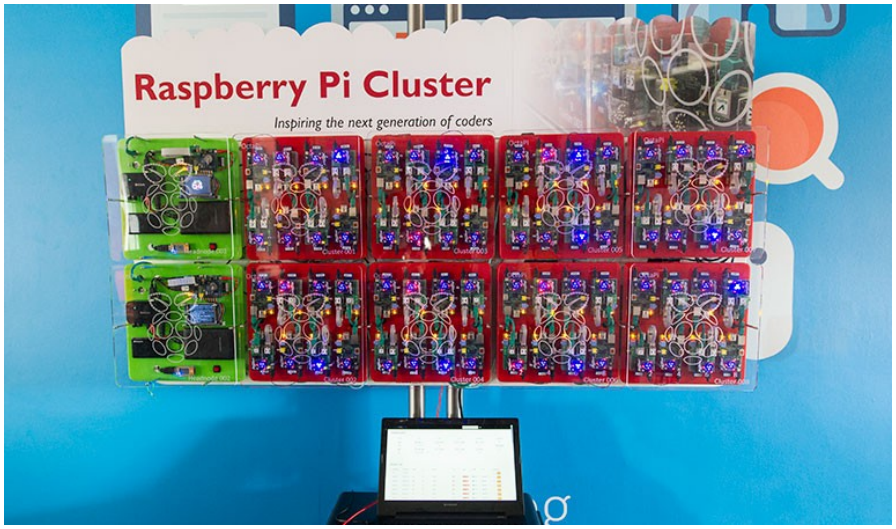

FIGURA 5.6: Vistazo general de la estructura del sistema Bramble

Se desconocen datos sobre el coste total del sistema.

5.5.1.4. 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[32]. El sistema está diseñado para ejecutar aplicaciones sobre *MPI*. Se desconocen datos sobre el coste total del sistema.

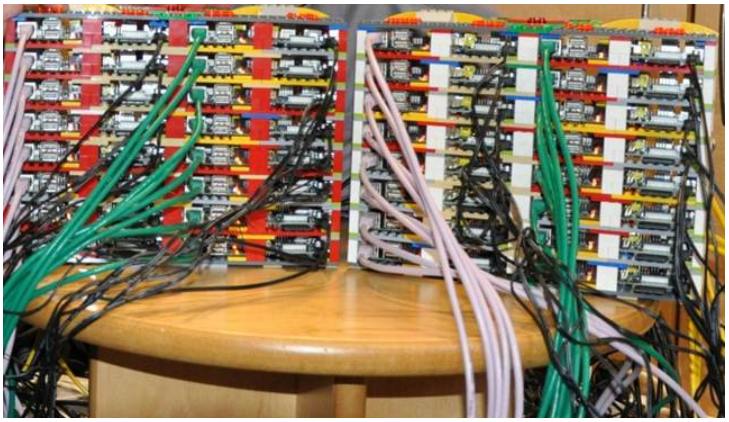

FIGURA 5.7: Clúster Iridis

5.5.1.5. 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, de 99 dólares por unidad [33].

5.5.2. 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.

5.5.3. *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*[34], 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/

5.6. Evaluación de alternativas

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

- Coste económico.
- Prestaciones técnicas (potencia de procesamiento, entrada/salida, capacidad de almacenamiento,facilidad de interconexión con otros elementos…).
- 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).

	Virtualización de entornos de trabajo
Descripción	Crear un conjunto de nodos virtuales dentro de una máquina que simulen un sistema distribuido
Ventajas intrínsecas	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	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	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. Tres equipos Dell PowerEdge r410, con dos procesadores Xeon E5620, 32GB de RAM y disco duro de 500GB en RAID1.
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	Dependiente de las capacidades de virtualización del equipo disponible, y el número de nodos y usuarios a gestionar.
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	Se desconocen datos concretos.
Obsolescencia	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	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.
Análisis coste/beneficio	El coste es muy bajo, dado que no es necesario adquirir ningún equipo. El consumo energético también es bajo, pues estos equipos ya se encuentran en funcionamiento, y añadir un sistema como el que se plantea crear no incrementaría previsiblemente el gasto.

Cuadro 5.10: Evaluación de las características de la virtualización de entornos de trabajo

	Clúster con equipos de escritorio
Descripción	Crear un conjunto de nodos virtuales dentro de una máquina que simulen un sistema distribuido
Ventajas intrínsecas	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	No se exploran las características únicas de otros sistemas menos "convencionales", como los sistemas embebidos. El consumo energético es mayor y existe una mayor demanda de espacio que puede dificultar la implementación de diferentes aplicaciones didácticas ya planteadas como objetivo funcional del sistema.
Facilidad de trabajo	Soporte completo de casi la totalidad de las distribuciones de GNU/Linux. Las necesidades de manipulación de hardware se minimizan.
Prestaciones técnicas	Los equipos cuentan con una arquitectura x86/x86-64, entre 2 y 4 GB de memoria principal, conectividad Ethernet y USB, así como espacio de almacenamiento en un 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	Dependiente únicamente del coste económico de la adquisición de nuevos equipos, o de la disponibilidad de equipos que no estén utilizados.
Mantenimiento	Las propias de cualquier sistema multiusuario y las específicas del montaje dado (en materia de refrigeración, gestión de cableado, etcétera).
Consumo energético	El típico de cualquier equipo de escritorio.
Obsolescencia	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	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…).

Cuadro 5.11: Características de un clúster con equipos de escritorio

	Clúster con computadores embebidos
Descripción	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	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	La potencia del sistema es pequeña.
Facilidad de trabajo	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. Disponen de entre 512 MB y 2GB de memoria principal, conectividad Ethernet, I^2C, GPIO y USB. La alimentación se realiza Alimentación a través de un puerto USB o el GPIO.
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	Dependiente únicamente del coste económico de la adquisición de nuevos equipos.
Mantenimiento	Las mismas que cualquier sistema multiusuario.
Consumo energético	Variable según modelo, entre 3 y 4 W, con 5V de tensión y un amperaje variable entre 0.6 y 0.8 A.
Obsolescencia	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	El Departamento de Informática y Automática cuenta con varios de estos equipos actualmente que podrían disponerse para el uso en el proyecto.
Análisis coste/beneficio	

CUADRO 5.12: Características de un clúster con computadores embebidos

	Clúster con equipos embebidos multimedia
Descripción	Utilización de equipos embebidos diseñados para aplicaciones multimedia en el sistema (ejemplos de alternativas comerciales son **Chromecast**, **Apple TV**, **Amazon Fire TV**…).
Ventajas intrínsecas	Relación potencia/precio presumiblemente similar o superior a soluciones de coste similar como las placas Raspberry Pi.
Inconvenientes intrínsecos	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	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**[8]: Procesador ARM de 2 núcleos a 1.2 GHz. 512 MB de memoria principal. 2 GB de almacenamiento no extensibles. La alimentación se realiza por micro-USB. Utiliza un sistema operativo basado en Google TV, ChromeOS y Android.
Coste económico	El coste de estos equipos es reducido, inferior a 30 € por unidad.
Escalabilidad	Dependiente del coste de adquisición de nuevos equipos y las facilidades de interconexión de la plataforma (previsiblemente compleja, debido a la ausencia de otra conexión diferente a la inalámbrica).
Mantenimiento	Dependiente del número de modificaciones que se realicen a las diferentes capas del sistema. En el peor de los casos puede que el administrador tenga que someterse a una etapa de formación para realizar un mantenimiento adecuado del sistema sin depender de los desarrolladores. Otras necesidades son aquellas derivadas del mantenimiento de un sistema multiusuario sumadas a posibles problemas de interconexión si se utiliza una red inalámbrica.
Consumo energético	El diseño de estos equipos está orientado a la minimización del consumo energético, por lo que se estima reducido.
Obsolescencia	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. No obstante, el sistema subyacente es conocido (Linux). Se prevé que las necesidades que el sistema cubre no demandarán una mayor potencia de cálculo en el futuro.
Material	No se dispone de material de estas o similares características.
Análisis coste/beneficio	

Cuadro 5.13: Características de un clúster con equipos embebidos multimedia

5.6.1. 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.

Dentro de las diferentes alternativas, se escoge la placa **Raspberry Pi** como modelo concreto.

5.6.2. 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[9] 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.

[9] http://elinux.org/RPi_Distributions

	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[10][11]
Memoria	512 MB compartidos con GPU	512 MB compartidos con GPU	1 GB compartido con GPU
LINPACK [35–37]	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	700 mA, 5 V (3.5 W)	600 mA, 5 V (3 W)	800 mA, 5 V (4 W)[12]
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 [13]	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 [14]
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 5.14: 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 (además de otras características necesarias).

5.6.3. 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 que el usuario esté familiarizado con GNU/Linux.	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 y 57000 en el **AUR**[15]
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.	Unos 40000[16]
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 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.	Unos 20000

Cuadro 5.15: 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).	No se conocen cifras
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 maximizar el grado de personalización del sistema.	Permite ser modificado de forma sencilla.	Poco soportado en Raspberry Pi.	No se conocen cifras
Windows 10	Diseñado para el paradigma del Internet de las Cosas,	Sencillo de utilizar, con soporte (previsiblemente) del *framework* **.NET**.	Soporta un conjunto de tecnologías ya conocidas pero no compatibles con otras alternativas. El soporte de **.NET** constituiría una ventaja clave.	Aún no se encuentra disponible[38]. Esta diseñado para un propósito especifico. No compatible con software para Linux de forma nativa.	No se conocen cifras

Cuadro 5.16: Comparativa de sistemas operativos (2)

5.7. Ingeniería de sistemas: propuesta de solución definitiva

En esta sección se describen los aspectos a alto nivel sobre todos los componentes que se integrarán en el sistema final (tanto *hardware* como *software*), así como las relaciones entre los mismos. En función de la evaluación llevada a cabo se extraen las siguientes decisiones de diseño que conforman la propuesta de solución definitiva.

5.7.1. *Hardware*

5.7.1.1. Nodos

Todo el sistema se construirá sobre placas **Raspberry Pi 2** debido a su alta versatilidad, gran potencia de cálculo, interfaces de comunicación, soporte por parte de las diferentes comunidades de desarrolladores, coste y bajo consumo eléctrico.

5.7.1.2. Equipos auxiliares

En caso de que sea necesario integrar algún nodo en el sistema se utilizarán equipos **COTS** en desuso presentes en la infraestructura. La utilización de este tipo de nodos se realizará siempre con carácter auxiliar (p.ej. para ofrecer un servicio que sea difícil de soportar por los nodos principales).

5.7.1.3. Interconexión física

El sistema deberá interconectar los diferentes nodos utilizando la red presente en la organización. No obstante se plantea el uso de dispositivos de enrutado o conmutación para mejorar del rendimiento.

5.7.1.4. Alimentación

La alimentación del sistema deberá proceder de una única fuente, a fin de minimizar las cantidades de cableado, transformadores u otro tipo de elementos.

5.7.1.5. Estructura

Se deberá diseñar una estructura propia que recoja todos los componentes del sistema, a fin de facilitar la instalación en su ubicación final.

5.7.2. *Software*

5.7.2.1. 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.

Se partirá de la instalación base del sistema operativo debido a que incluye el conjunto más pequeño de herramientas con el que el sistema operativo es capaz de trabajar, añadiendo posteriormente aquellos paquetes necesarios para la construcción del sistema. Este enfoque evita la inclusión de *software* innecesario que afectaría negativamente al rendimiento de la solución final.

5.7.2.2. Componentes del sistema operativo

Se crearán un conjunto de componentes que se integrarán en el sistema operativo y podrán ser aprovechados por aplicaciones creadas por usuarios o por herramientas internas del propio sistema. Dichos servicios incluyen mecanismos de descubrimiento, coordinación, acuerdo, autoconfiguración de componentes o gestión de usuarios, entre otros.

5.7.2.3. Interfaces de conexión entre componentes

Aquellos servicios ofrecidos por el sistema operativo que sean aprovechables por los usuarios finales deberán ofrecer una interfaz de conexión con los mismos. Dicha interfaz deberá apoyarse en mecanismos de interconexión conocidos y deberá ofrecerse en el mayor número de plataformas y lenguajes de programación posible.

5.7.2.4. Servicios

El sistema se plantea como un conjunto de nodos que ofertan servicios consumibles por otros componentes o terceras partes. Dichos servicios deberán contar con una serie de

puntos de acceso claramente definidos que constituirán las interfaces de uso de dichos servicios.

5.7.2.5. Instalación y mantenimiento

El sistema final se compondrá de un conjunto elevado de nodos, por lo que se espera que la instalación y configuración de los mismos sea un proceso tedioso. A fin de minimizar dicha carga de trabajo se crearán herramientas que automaticen dichas etapas, idealmente, de forma completa.

5.7.2.6. Herramientas de desarrollo a utilizar

Se plantea el uso del lenguaje de programación Python[17] como herramienta principal de desarrollo, debido a su potencia de cálculo y simplicidad, que permite crear aplicaciones demandantes de 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.

5.8. Integración

El sistema se integrará en la organización objetivo utilizando diferentes recursos de la misma, como la red que interconecta todos los equipos o el mecanismo de autenticación.

5.9. Proceso

La determinación del proceso de desarrollo del sistema influirá significativamente en la consecución de los diferentes objetivos marcados.

Las particulares características cada proyecto propician la elección de un proceso u otro. En el caso concreto del proyecto en cuestión se deben considerar las siguientes características:

[17] http://python.org

- No se cuenta con experiencia previa en la construcción de varios componentes cruciales. Serán necesarias etapas de aprendizaje y la tolerancia a decisiones erróneas deberá ser alta.
- El proyecto tiene un alto componente de experimentalidad: el sistema final será el resultado de los diferentes procesos de investigación y prueba de soluciones para cada uno de los problemas planteados. La incertidumbre es por tanto alta, y por ello la toma de decisiones debe ser reversible, a fin de poder cambiar el enfoque aplicado a un problema si se detecta que los resultados del mismo serán infructuosos. Dicha reversibilidad implica principalmente la capacidad de poder rectificar en un periodo de tiempo pequeño, debido principalmente a la restricción de tiempo que el proyecto incluye.
- Es esperable que los requisitos definidos en las diferentes fases del proyecto deban ser modificados debido a la gran incertidumbre con la que se debe lidiar.

Es por todo ello que es necesaria la determinación de un proceso de desarrollo que deje espacio a etapas de aprendizaje y experimentación, así como la modificación de los diferentes requisitos preestablecidos.

5.9.1. Elección del proceso

En virtud de las razones expuestas se opta por un proceso ágil apoyado sobre prototipos.

Un prototipo es un componente *software* que implementa un subconjunto de la funcionalidad del sistema final y es operativo. El desarrollo de prototipos permite realizar evaluaciones de la funcionalidad implementada de forma efectiva ahorrando costes y tiempo de desarrollo. En el caso concreto de este proyecto posibilitan la evaluación de una alternativa sobre el resto, así como la viabilidad de una solución sobre el resto. Se apuesta por un modelo de desarrollo de prototipos evolutivo[39], creando versiones funcionales que paulatinamente crezcan en complejidad. Este tipo de prototipado permite experimentar con una estrategia de desarrollo (probar un algoritmo, experimentar con un lenguaje de programación, aprovechar una biblioteca, *framework*...) y analizar las ventajas e inconvenientes de la misma en poco tiempo, siendo posible desechar el prototipo en caso de que el camino elegido no satisfaga los requisitos planteados.

Los procesos ágiles suelen ser utilizados a la hora de realizar un proyecto en equipos pequeños en entornos cambiantes o con un grado de incertidumbre muy alto. Procesos "tradicionales", como el desarrollo *en cascada* o el lineal no responden de forma dinámica a dichas propiedades, y su adaptabilidad es menor.

En el caso del presente proyecto, se opta por un proceso ágil basado en iteraciones de corta duración (entre 7 y 12 días) con una serie de objetivos definidos y producen una versión más avanzada de los prototipos. Debido a que el proyecto comprende el desarrollo de una serie de productos independientes que conforman un sistema (en contraste con un único producto monolítico final) es necesario paralelizar el desarrollo de todos los procesos a lo largo de las diferentes iteraciones, balanceando la carga de trabajo entre las diferentes tareas.

5.9.2. Desarrollo de subsistemas

Los diferentes aspectos relativos al desarrollo de cada uno de los subsistemas se definen en los anexos dispuestos a tal efecto.

5.9.3. Integración del sistema

Se ha realizado un proceso de integración creciente de los diferentes prototipos creados como resultado de cada uno de los ciclos de desarrollo. Esta práctica es de utilidad para detectar diferentes fallos en el sistema o anomalías fruto de las interacciones entre componentes de forma temprana.

5.9.4. Instalación del sistema

Se han creado herramientas que facilitan la instalación del sistema (ver 7.3.4.1) en diferentes tipos de infraestructuras que cuenten con una serie de propiedades mínimas (que se reducen a una conexión de red funcional).

5.9.5. Evolución del sistema

La solución presentada es altamente escalable y existe un equipo de soporte para todos los componentes el *software* utilizado, aspectos que garantizan la inclusión de nuevas características y el mantenimiento de los paquetes instalados.

5.9.6. Desmantelamiento del sistema

Ningún componente contiene materiales cuya manipulación incorrecta pueda dañar su entorno. En el caso de que el desmantelamiento no se produzca por fallos en el *hardware* del sistema, sus componentes pueden ser aprovechados para otros proyectos académicos o

profesionales por su poseedor. Los diferentes paquetes *software* no dependen del hardware del sistema para funcionar, salvo varias excepciones, tales como herramientas creadas para un problema concreto de esta plataforma (como Marcobootstrap, entre otros).

5.9.7. Planificación temporal

La planificación temporal del proyecto se realiza en base a las diferentes iteraciones llevadas a cabo. Al finalizar una de las fases, se realiza una reunión con todo el equipo de desarrollo y se analiza el estado de las nuevas versiones de los prototipos, problemas que se han dado y nuevas vías de actuación. A partir de este análisis se determinan los objetivos a cumplir en la siguiente iteración.

El resto de aspectos relativos a la gestión del proyecto se definen en el anexo “Gestión del proyecto”.

Capítulo 6

Análisis y diseño

En el que se describen los diferentes aspectos de las fases de análisis y diseño del sistema desde una perspectiva holística.

6.1. Análisis

Recoger todos los aspectos de análisis de un sistema como el creado en un único capítulo es contraproducente para la adecuada comprensión de los diferentes procesos llevados a cabo. Es por ello que en el presente capítulo se detallarán los diferentes aspectos de análisis para el sistema como unidad, que ayudarán a la identificación de las necesidades a satisfacer por el mismo. Dichos aspectos serán de utilidad durante el desarrollo de las restantes etapas de análisis centradas en cada uno de los diferentes componentes del sistema.

6.1.1. Identificación de componentes

Todo sistema se compone del conjunto de integrantes y las relaciones que estos mantienen entre ellos mismos y su entorno, con una serie de objetivos a cumplir a través de dichas interacciones. El límite entre el sistema y su entorno es de necesario estudio para comprender los diferentes procesos de entrada y salida que se desarrollan.

6.1.1.1. Componentes principales

Los componente principales del sistema son los nodos Raspberry Pi, que serán los encargados de la ejecución de las diferentes tareas solicitadas por los usuarios del sistema.

6.1.1.2. Componentes secundarios

En una primera instancia del proceso de análisis no se detalla ningún tipo de nodo secundario, delegando en los componentes principales del sistema todas las tareas a llevar a cabo.

Sin embargo, en las diferentes etapas de desarrollo se plantea el uso de varios componentes secundarios para la gestión de una serie de tareas cuya ejecución en el conjunto de componentes principales es dificultosa, o su delegación beneficia al conjunto de nodos principales. En cualquier caso, dichas tareas pueden ser asignadas a los componentes principales en cualquier momento (ver 7.4).

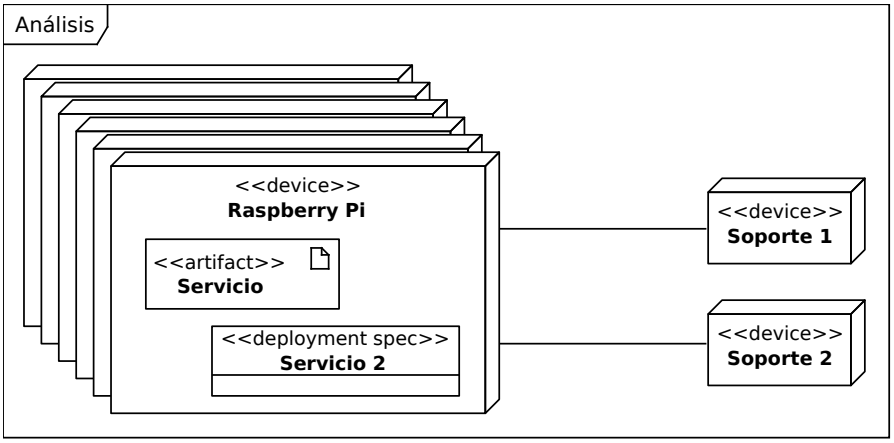

FIGURA 6.1: Análisis de componentes del sistema

6.1.1.3. Interconexión

Se plantea el uso de cableado físico Ethernet para la comunicación entre los diferentes nodos. Este tipo de conexión es soportada por todos los componentes anteriormente mencionados, y se cuenta con dicho cableado en la infraestructura del centro (ver 5.1.1).

6.1.2. Información gestionada por el sistema

Se plantea un conjunto de requisitos de información gestionados por el sistema pequeño, sin embargo de alta sensibilidad, que el siguiente conjunto de datos.

6.1.2.1. Credenciales de usuario

Claves de acceso al sistema. Generalmente estas se componen de un par usuario-contraseña. Sin embargo, también se contemplan sistemas alternativos, como la autenticación de en

clave pública. La manipulación de la identidad del usuario también es un aspecto clave del sistema.

6.1.2.2. Archivos personales

Junto a las claves de usuario es necesario gestionar los diferentes ficheros de trabajo que los usuarios manipulen y almacenen en el sistema. Esto implica proporcionar los diferentes mecanismos de acceso a dichos datos y un mecanismo de privilegios (lectura, escritura, ejecución) para impedir la manipulación de forma no deseada de los mismos.

6.1.2.3. Ficheros de configuración e información del sistema

Diversos componentes del sistema utilizan mecanismos de cifrado cuyas claves no deben ser conocidas por ninguna entidad mas que el administrador del sistema. Los ficheros de configuración del sistema y de las diversas aplicaciones a construir no deben ser modificables por usuarios no autorizados, pues definen aspectos del comportamiento del sistema que pueden comprometer la integridad del mismo si son modificados con fines perniciosos.

6.1.2.4. Registros

Diversos registros del comportamiento del sistema y de las operaciones realizadas en el mismo serán almacenados en el sistema para su posterior análisis. Dichos ficheros pueden incluir información sensible, como datos de usuarios, por lo que el acceso a los mismos deberá estar restringido.

6.1.2.5. Información de estado

Si bien la mayor parte de la información se describe en ficheros de carácter permanente, una parte de la información de importancia en el sistema es generada y gestionada por las propias aplicaciones sin almacenar la misma en ningún tipo de soporte permanente. La volatilidad de la información hace que la versatilidad de la misma sea mayor, sin embargo, será necesario contar con una serie de mecanismos que preserven la información frente a circunstancias como recargas de información, reinicios. La manipulación de este tipo de información (actualizaciones, consulta, modificación) presenta una serie de aspectos diferentes a los propios de las estructuras de datos tradicionales.

6.1.2.6. Equipo de soporte

Como equipos de soporte se plantea el uso del almacenamiento presente en los nodos principales, utilizando, en caso de que sea conveniente, algún nodo secundario como almacén de información.

6.2. Identificación de transacciones

Se plantean los siguientes tipos de transacciones. La frecuencia estimada de estas operaciones se debe determinar en fases posteriores basados en las estadísticas del sistema actual (ver 5.1.1.2).

- Operaciones de usuario

  Autenticación contra el sistema.

  Ejecución y diseño de trabajos a realizar por el sistema.

  Realización de pruebas de algoritmos y despliegues.

  Almacenamiento de ficheros.

- Operaciones de administración.

  Actualizaciones del sistema.

  Operaciones rutinarias de mantenimiento.

  Acceso y análisis de registros.

6.2.1. Evolución del sistema

En caso de proporcionar una solución exitosa para el conjunto de problemas a resolver por el sistema, es esperable un incremento en el número de componentes principales con el fin de aumentar su capacidad de cómputo. Por ello, la escalabilidad del sistema debe ser uno de los requisitos fundamentales del mismo.

6.2.2. Identificación de usuarios

Los siguientes usuarios harán uso del sistema de forma directa o indirecta[1].

[1] En los diferentes documentos de análisis adjuntos se detalla la interacción de cada entidad con cada uno de los componentes del sistema de forma más detallada, sirviendo esta enumeración como vista global de los diferentes agentes.

6.2.2.1. Desarrolladores

Son aquellos individuos que utilizan el sistema como herramienta de creación y prueba de aplicaciones distribuidas.

6.2.2.2. Estudiantes

Utilizan la plataforma como herramienta para la elaboración de trabajos académicos relacionados con el área de conocimiento de la computación distribuida y como mecanismo para facilitar el estudio de dicha rama.

6.2.2.3. Profesorado

Docentes de las áreas anteriormente mencionadas, que utilizarán el sistema para planificar diferentes ejercicios didácticos a resolver por los alumnos.

6.2.2.4. Administradores

Realizan tareas de mantenimiento en el sistema.

6.3. Diseño

6.3.1. Ámbito del sistema

Se distinguen dos tipos de nodos en el sistema. Los nodos principales se agruparán en una misma ubicación física y llevarán a cabo todas las tareas principales del sistema. De forma complementaria y accesoria existirán nodos especializados en una tarea que se les delegará. Las características de estos nodos (tiempo de computación superior, información almacenada en ellos…) hacen que el rendimiento del sistema aumente gracias al proceso de delegación.

6.3.2. Diseño de datos

Se plantean de forma global las siguientes fuentes de datos.

6.3.2.1. Estructuras de archivo

- Directorio LDAP.
- Bases de datos.
- Directorios de usuario.
- Ficheros de configuración.
- Ficheros de *log* para registrar todas las operaciones llevadas a cabo por el sistema, situaciones irregulares…, con el objetivo de ser analizados posteriormente.

6.3.3. Diseño arquitectónico

Se plantea el siguiente esquema arquitectónico:

- Un conjunto de nodos principales centralizados en una misma localización física.
- Varios nodos secundarios a los que se delegará una serie de tareas. Sus características (mayor capacidad de procesamiento, *hardware* exclusivo…hacen que esta delegación mejore el rendimiento global del sistema o cualquier otra propiedad).
- Mecanismos de interconexión, enrutado y alimentación.

6.3.4. Diseño de la interfaz

Las interfaces de acceso al sistema son las siguientes:

- Acceso mediante SSH
- Interfaces web, descritas en la documentación adjunta referente a cada una de las herramientas creadas o utilizadas.

6.3.5. Diseño procedimental

Todos los aspectos de programación (algoritmos empleados, ejemplos de código) se reflejan en la documentación adjunta referente a cada una de las herramientas creadas o utilizadas (ver A).

6.3.6. Pruebas

Se ha utilizado un desarrollo dirigido por pruebas en la mayoría de las herramientas creadas, como se detalla en la documentación técnica que acompaña a cada una de ellas, así como una batería de pruebas de una de las últimas versiones del sistema, como se detalla en el anexo "Evaluación del rendimiento del sistema en un entorno real".

6.3.7. Entorno tecnológico del sistema

El sistema se integrará en la Facultad de Ciencias de la Universidad de Salamanca. Todos los aspectos de esta organización se definen en 5.1.1.

6.3.8. Plan de desarrollo e implementación

El plan de desarrollo del sistema plantea las siguientes fases:

- **Febrero de 2015 a Julio de 2015**: Desarrollo de todo el sistema (*hardware*, *software*, pruebas de integración e instalación.)
- **Principios de septiembre de 2015**: Etapa de formación a los diferentes usuarios (profesores, administradores), y primeras pruebas masivas.
- **Septiembre de 2015-Junio de 2015**: Integración no definitiva del sistema en diferentes asignaturas.
- **Junio de 2015 en adelante**: Integración definitiva del sistema.

6.3.9. Adquisición del sistema

Antes de realizar cualquier operación de adquisición de componentes es necesario analizar diferentes factores, como las diferentes alternativas viables o soluciones de coste menor o incluso nulo.

En el caso del presente proyecto, la adquisición de una serie de componentes no es opcional, pues no se cuenta con ellos previamente. Tras evaluar diferentes opciones, se decide adquirir los siguientes componentes:

- Nodos **Raspberry Pi**.
- Cables de alimentación.

- Tarjetas SD/micro-SD (soporte de almacenamiento de la Raspberry Pi).
- Elementos estructurales.

En la fase inicial, se evalúa un gran número de opciones atendiendo a las prestaciones, precio, vida útil y tiempo de entrega[2].

El proceso de compra de los diferentes componentes necesarios para el sistema se ha realizado de forma incremental, comenzando por una compra de tamaño significativo que incluye los diferentes componentes del sistema y posteriormente adquiriendo los componentes supletorios necesarios.

Esta estrategia ha permitido refinar las necesidades inicialmente planteadas, lo cual ha reducido significativamente el coste, al poder evaluar alternativas de igual efectividad y menor coste o incluso buscar mecanismos para reutilizar componentes con los que ya se contaba. El coste total del sistema se define en 7.1.2.

Ítem	**Unidades**	**Precio**/ud.	**Total**	**Notas**
Raspberry Pi B+	4	€25.84	€103.36	Disponible para entrega en 2 día(s) laborable(s).
Raspberry Pi Rev 2	4	€30.58	€122.32	Temporalmente fuera de stock. Disponible a partir del 20/04/2015, con entrega en 2 día(s) laborable(s).
Rasbperry Pi B	4	€26.05	€104.20	

Cuadro 6.1: Coste de cada uno de los diferentes modelos de placa Raspberry Pi

Modelo	**Total**
Raspberry Pi B, Tarjeta SD, Ethernet	€145.08
Raspberry Pi B+, Tarjetas MicroSD, Ethernet	€169.72
Raspberry Pi Rev 2, Tarjetas MicroSD, Ethernet	€188.68

Cuadro 6.2: Coste total de cada una de las alternativas

Finalmente, se adquieren cuatro nodos Raspberry Pi Rev 2 con los componentes accesorios necesarios. El coste total es de 188.68 €.

[2]Las restricciones de tiempo que el proyecto implica exigen que los materiales sean entregados en un periodo de tiempo corto

Capítulo 7

Aspectos relevantes del desarrollo

En el que se detallan las diferentes etapas de desarrollo del sistema y las propuestas de solución final, incluyendo los aspectos de pruebas y evaluación del sistema.

7.1. Arquitectura física

El sistema a crear, si bien de carácter distribuido, localiza el conjunto de nodos principales en una misma ubicación física, externalizando únicamente los componentes secundarios.

Esta centralización apoya uno de los objetivos principales del proyecto. Al contar con todo el sistema en una única ubicación es posible comprender mejor los mecanismos de distribución de tareas de forma más efectiva que en un sistema distribuido por toda la infraestructura. Esta decisión aporta además una serie de ventajas técnicas, entre las que figuran las siguientes:

- Centralización del sistema de alimentación eléctrica.
- Reducción del cableado de red.
- Favorece la organización de todos los componentes del sistema.
- Facilita el traslado del sistema a otra ubicación.
- Simplifica el mantenimiento del sistema.

7.1.1. Propuestas de solución

7.1.1.1. Primera propuesta: estructura básica

En las primeras fases del proyecto se plantea la utilización de separadores de nylon atornillados a los orificios que las placas disponen para tal fin, creando una pequeña estructura en forma de torre (muy similar al *Dramble*, ver 5.5.1.2). Dicha solución se acepta debido a su efectividad con un precio bajo.

7.1.1.2. Segunda propuesta

La segunda propuesta apuesta por mejorar la versatilidad del diseño. Se plantea un diseño en forma de "estantería" que incluya una bahía para cada nodo, permitiendo la instalación de cada uno de ellos de forma independiente y la instalación de nuevos nodos. Se decide desestimar el primer prototipo y desarrollar esta solución.

Elección de materiales

Inicialmente se valora el uso de metal para la construcción del esqueleto de la estructura, pero finalmente se opta por el metacrilato, pues mejora la visualización de cada uno de los nodos y añade un mayor atractivo estético al proyecto.

Las diferentes placas de metacrilato se unirán mediante barras metálicas. Las uniones se realizarán con tornillos a menos que estos deterioren la estética de la estructura, en cuyo caso se utilizarán adhesivos.

Construcción

La fase de construcción se extiende durante aproximadamente dos semanas (compaginadas con con el resto de tareas de desarrollo. El tiempo total dedicado a esta tarea asciende a 20 horas, como se puede observar en el anexo referentes a la gestión del proyecto "Gestión del proyecto").

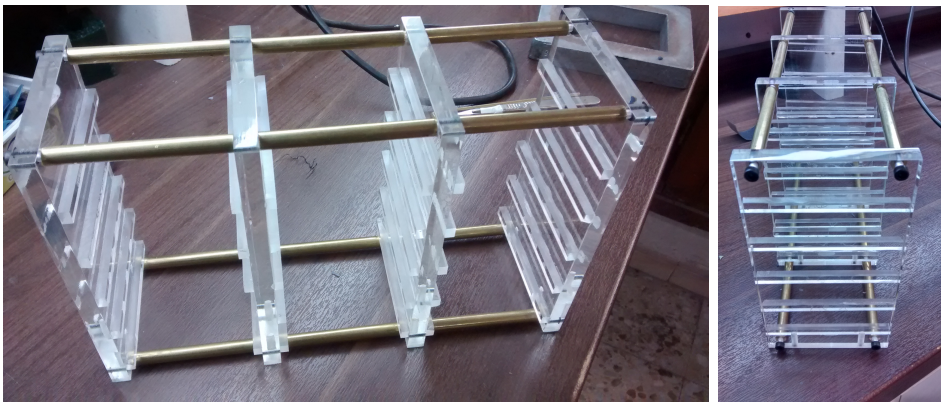

FIGURA 7.1: Vista general de la estructura del primer prototipo y vista de perfil del mismo.

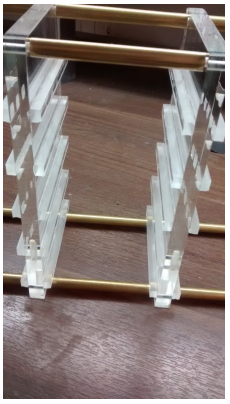

FIGURA 7.2: Vista en detalle de los "raíles" creados para cada uno de los nodos. Nótese la ausencia de tornillos, que han sido reemplazados por adhesivo.

Este prototipo es considerado aceptable, y se decide partir del mismo para elaborar la solución final.

7.1.1.3. Tercera propuesta

A pesar del éxito del segundo prototipo, durante las siguientes fases de desarrollo se plantea la viabilidad del prototipo para albergar una serie de componentes que acompañen a los nodos. Utilizar, como se planteaba inicialmente, un ladrón USB, para la alimentación del sistema parece una solución poco efectiva (incrementa la cantidad de cableado a utilizar). La conexión a la red también genera incertidumbre, pues se planteaba inicialmente la conexión a la misma de forma individual para cada nodo. Esta decisión complicaría la gestión del cableado.

Además, se observan errores en las medidas iniciales para cada uno de los raíles, pues no se contó con el espacio que las placas de diodos LED consumirían.

Por ello, se plantea la elaboración de un tercer prototipo basado en el anterior que tenga las siguientes propiedades:

- La estructura albergará a todos los nodos así como los diferentes componentes que estos requieran para su funcionamiento.
- El sistema deberá centralizar en un único mecanismo de alimentación el suministro de energía eléctrica a los nodos y a cualquier otro componente integrado en la estructura.
- La conexión a la red de datos deberá estar centralizada.

Elección de materiales

Se mantienen las decisiones de materiales realizadas para el segundo prototipo, y se intentará reutilizar todo el material posible del mismo.

Para la alimentación eléctrica se decide utilizar una fuente de alimentación de 5 V capaz de suministrar hasta 20 amperios de energía que se puede conectar a la red eléctrica presente en la infraestructura.

Las conexiones de red de datos se centralizarán en uno o varios *switches* con velocidades de hasta 100 Mb/s paralelos.

Sin embargo, queda aún una incógnita por resolver: si bien la fuente de alimentación es capaz de suministrar energía a los nodos, es necesario definir cómo se suministrará dicha energía. Se plantean dos opciones:

Alimentación a través del puerto GPIO

El puerto **GPIO** de la Raspberry Pi es utilizado comúnmente para la alimentación de periféricos conectados a la misma, a través de varios pines dispuestos para tal fin. Sin embargo, es posible alimentar a la propia placa a través de los mismos[1].

Extendiendo dos cable desde la fuente de alimentación hasta los pines 1 y 3 (+5V y neutro) es posible alimentar la placa.

Esta solución implica un riesgo considerable. El puerto GPIO no dispone de mecanismos de protección ante un uso como este (en el caso de la alimentación por USB se cuenta con dicha protección). En caso de una sobrecarga eléctrica la placa sería dañada de forma irreversible. Se plantea el uso de fusibles de 2 amperios para proteger a las placas de

[1] Esquema completo del circuito de la Raspberry Pi: https://www.raspberrypi.org/wp-content/uploads/2012/04/Raspberry-Pi-Schematics-R1.0.pdf

este tipo de sobrecargas (es el máximo amperaje que la placa soporta), en caso de que esta solución sea la aceptada.

Alimentación a través del puerto USB

Es posible adaptar la propuesta de solución anterior a una alternativa más prometedora. Aprovechando los mecanismos de protección frente a sobrecargas del puerto USB se decide adquirir y modificar cabezas micro-USB macho para soldarlas a cables conectados a la fuente de alimentación. Dichas cabezas se conectarán a cada una de las placas. Se mantienen no obstante los fusibles ya adquiridos como medida preventiva conectando uno a cada cable de corriente positiva.

Esta solución permite centralizar todos los cables de alimentación en un único punto a la vez que se mantienen los mecanismos de protección que integran las placas, manteniendo a efectos prácticos el sistema de alimentación que se ideó para la Raspberry Pi.

FIGURA 7.3: Cables modificados

Gestión de la red

Se crearán cables Ethernet a medida para cada uno de los raíles.

LEDs

Situar los diodos supone un problema significativo. Utilizar cualquier tipo de cableado complicaría el diseño de forma significativa, por lo que es necesario buscar una alternativa.

Como solución se decide crear placas con circuitos impresos que conecten los diferentes diodos al puerto GPIO, utilizando una tira GPIO soldada al circuito que se conectará a los pines de la Raspberry. De esta forma se evita cualquier tipo de cableado y se evita realizar soldaduras a la placa.

A fin de ahorrar espacio, los LED se situarán sobre la Raspberry cubriendo la parte inferior de la misma.

7.1.2. Coste

Ítem	Unidades	Precio unitario	Total
Placas de metacrilato de 10mm	1 m^2	1	30 €
Placas de metacrilato de 5mm	1 m^2	1	10 €
Placa de metacrilato de 4mm	1 m^2	1	20 €
Varilla de latón de 8mm	1	1	15 €
Cable Ethernet	3 metros	1.2 €/metro	3.6€
Fusibles de 2 A	10	0.3 €	3.15
Portafusibles	12	0.5	6 €
Fuente de alimentación de 5 V	1	16.63	16.63
Tira de pines GPIO	3	0.9	2.84
Diodos LED	30	0 €	0 €
Resistencias	30	0 €	0 €

CUADRO 7.1: Coste de los materiales de la estructura final. Los materiales con coste igual a 0 euros no han sido adquiridos, ya se contaba con ellos o han sido reutilizados

7.1.3. LEDs

Para facilitar la inclusión de una serie de diodos LED en el sistema, se diseñará un circuito impreso que aglutine todos los componentes y sea fácil de instalar.

7.1.4. Resultado final

El resultado final es una estructura que es capaz de aglutinar todos los componentes del sistema salvo los nodos secundarios del mismo, utilizando una única fuente de alimentación para todos los componentes.

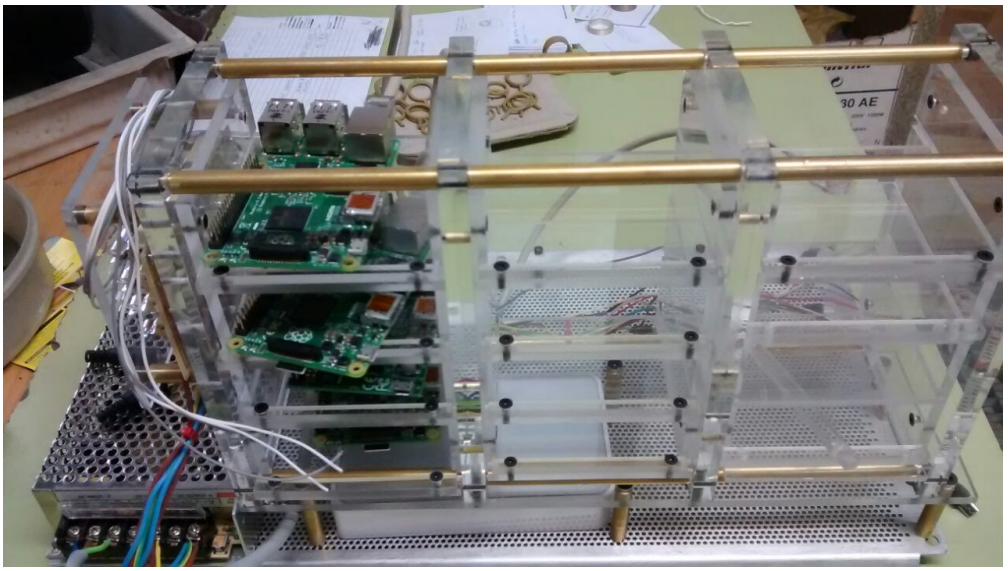

FIGURA 7.4: Vista general de la estructura final

7.1.4.1. Alimentación eléctrica

La fuente de alimentación protege todos los componentes del sistema al incluir un conjunto de fusibles.

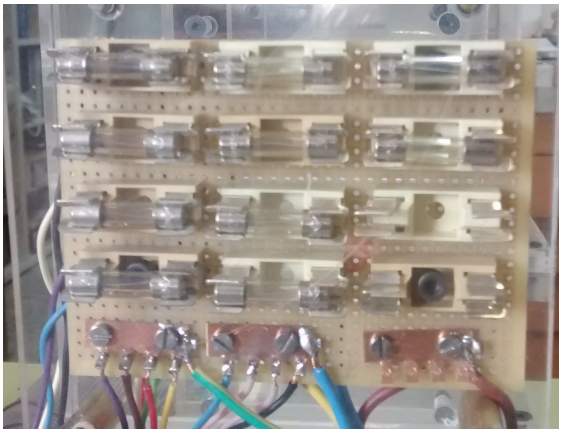

FIGURA 7.5: Panel de fusibles protectores

7.1.4.2. LEDs

Los LED se disponen en un circuito impreso diseñado para este propósito específico, donde se sitúan en fila acompañados por una resistencia protectora y un punto de conexión al puerto GPIO.

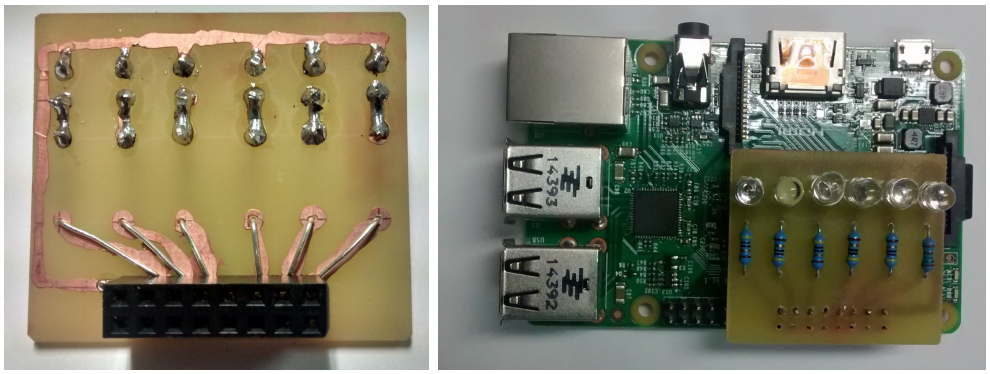

FIGURA 7.6: Vista del reverso de la placa y el anverso de la misma conectada a una Raspberry Pi.

La conexión con el puerto GPIO se realiza a través de unas "tiras" hembra que se sueldan a la placa y posteriormente se conectan a la **Raspberry Pi**, ahorrando espacio y permitiendo su sustitución en cualquier momento (al contrario que alternativas como, por ejemplo, soldar la placa directamente al puerto GPIO de la Raspberry Pi).

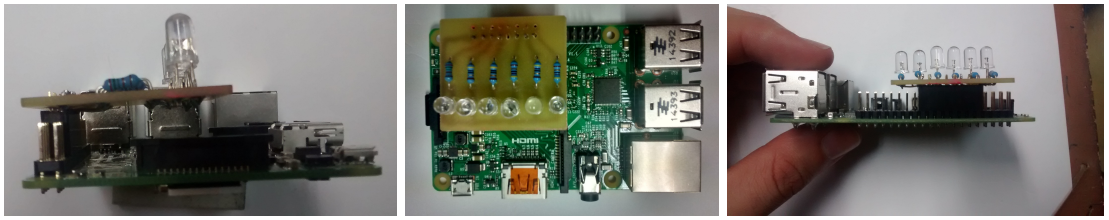

FIGURA 7.7: El conector GPIO hembra eleva la placa, pudiendo superponer la misma al resto de componentes, ahorrando espacio.

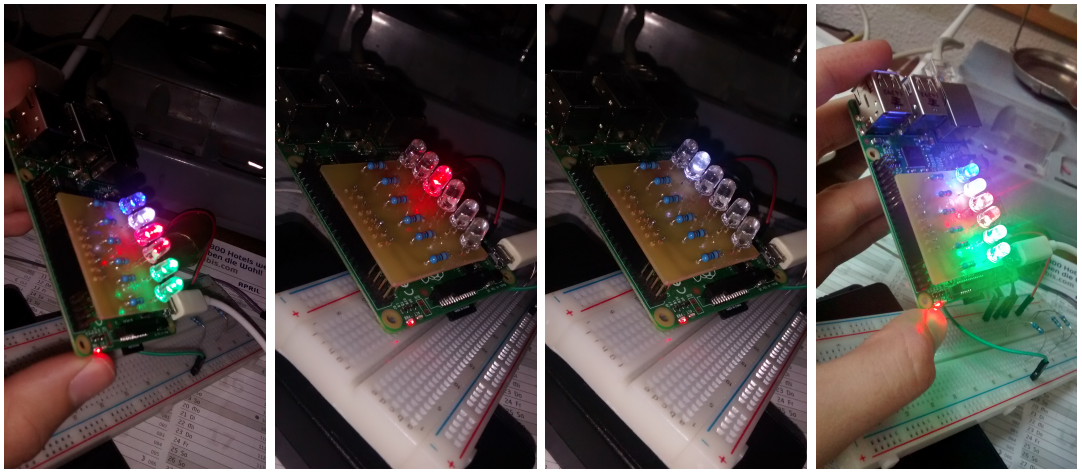

FIGURA 7.8: El circuito impreso en funcionamiento

Todos los diagramas que describen la estructura pueden consultarse en el anexo "Estructura física".

7.2. Arquitectura software

Con el objetivo de facilitar la comprensión del sistema como un todo esta memoria se estructura partiendo de los cimientos del sistema, ascendiendo hasta las aplicaciones de más alto nivel manipuladas por los usuarios, describiendo de forma exhaustiva todos los aspectos de relevancia y esfuerzos llevados a cabo para diseñar y desarrollar cada uno de los componentes del sistema.

7.2.1. Sistema operativo

Una pieza fundamental del *software* del sistema es la capa más básica del mismo, el sistema operativo. Es por ello clave elegir un sistema adecuado para los objetivos que se desean alcanzar.

Como se describió en 5.7.2.1, se utilizará el sistema operativo Arch Linux en su variante para procesadores ARM por su diseño altamente modular, eficiencia y capacidad de adaptabilidad. Sobre dicho sistema operativo se desarrollarán el resto de componentes, sin que ello implique un diseño adaptado a este sistema operativo.

7.2.1.1. Características técnicas de Arch Linux

Arch Linux es una distribución del sistema operativo GNU/Linux creada y mantenida por una comunidad de usuarios. Desarrollada de forma independiente, es lo suficientemente versátil como para satisfacer cualquier propósito. Su desarrollo se centra en la simplicidad, elegancia y minimalismo, asumiendo que el usuario añadirá los componentes que restan para conseguir el entorno que desee. 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 elementos 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 incluir paquetes innecesarios en la instalación básica.

La gestión de paquetes se realiza utilizando la herramienta **pacman**[40], creada por y para el proyecto. El repositorio oficial ofrece una gran cantidad de *software*, si bien su tamaño es significativamente inferior al *Arch User Repository* (AUR), que contiene paquetes creados y mantenidos por usuarios.

Arch Linux utiliza el sistema de arranque **systemd**[41], un conjunto de módulos que proporcionan una gestión de dependencias entre servicios del sistema más eficiente y sencilla que el cargador clásico de las distribuciones GNU, **init**[42], inspirado en el utilizado por UNIX System V.

Otro de los aspectos de relevancia del sistema es la comunidad creada alrededor del mismo, que fomenta la implicación de cualquier usuario en cualquiera de los aspectos de relevancia en el desarrollo del sistema. La documentación del sistema es extensa y cuenta con recursos tanto para las características propias del sistema como para herramientas de terceros utilizadas en el mismo. Dichos aspectos del proyecto y una serie de consideraciones adicionales se recogen en los preceptos definidos en el documento *The Arch Way*[43], inspirado en el principio de desarrollo **KISS** (*Keep It Simple, Stupid*).

Arch Linux ARM[2] es un proyecto derivado de Arch Linux que tiene como objetivo portar el sistema operativo a dispositivos basados en la arquitectura ARM (pues el proyecto raíz está enfocado únicamente en las arquitecturas i686 y x68_64).Es una de las principales opciones a la hora de realizar proyectos con este tipo de computadores[44].

	Consumo de memoria (`free -m`)	**Espacio ocupado (`df -h`)**
Ubuntu Mate	702 MB	2.5 GB
Raspbian	61 MB	2.4 GB
Arch Linux	16 MB	400 MB

CUADRO 7.2: Comparativa de las demandas de espacio y memoria de diferentes sistemas operativos. Los datos corresponden a la huella de la instalación base del sistema operativo, sin ningún tipo de añadido instalado. Ubuntu, si bien conocido por su gran demanda de recursos, es el único que incluye una interfaz gráfica activada por defecto, lo cual eleva el consumo de memoria

El proceso detallado de elección del sistema operativo puede observarse en 5.6.3.

7.2.1.2. Sistema operativo en los nodos secundarios

En el caso del servidor que provee diferentes servicios de apoyo a los nodos principales (ver 7.4.2.1) se utiliza el sistema operativo Xubuntu GNU/Linux[3].

[2] http://archlinuxarm.org/
[3] http://xubuntu.org/

7.3. Servicios integrados en el sistema operativo

Diversos componentes del sistema operativo son utilizados para llevar a cabo operaciones de gestión o como herramienta para la creación de aplicaciones en niveles superiores. Gracias a la integración de estos componentes en niveles inferiores, es posible ofrecer dichos servicios a cualquier aplicación de usuario.

7.3.1. Gestión de servicios

systemd[4] es utilizado como gestor de los diferentes servicios del sistema. Se aprovecha su capacidad para el manejo dependencias entre los diferentes componentes (orden de arranque, qué servicios deben arrancarse para que otros puedan funcionar…) para establecer el orden de inicio al arrancar el sistema o iniciar automáticamente servicios sobre los que dependan otros.

7.3.2. Descubrimiento de servicios: el protocolo MarcoPolo

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. Al no existir un coordinador, es difícil establecer un mecanismo de interconexión entre los diferentes nodos que sea escalable e independiente de factores externos (e.g. IP asignada en un momento dado, número de nodos en la red).

Soluciones como el uso de servidores de nombres (**DNS**) permiten crear estructuras jerárquicas donde cada nodo está identificado por un nombre previamente asignado y conocido. Como primera propuesta de solución se plantea el uso de los nombres de equipo que el servidor **DHCP** presente en la infraestructura otorga. Con esta solución los nodos serían accesibles mediante un identificador invariable en el tiempo (en contraste con una IP dinámica). Sin embargo, dicha solución presenta un grave problema en materia de escalabilidad: no existe un "directorio" que recoja qué nodos están activos en un momento dado ni que refleje adiciones en la lista original de nodos, por lo que para poder realizar cualquier cambio en el conjunto de nodos del sistema será necesario configurar cada equipo manualmente. También existen protocolos inspirados en este enfoque como **mDNS** (*Multicast Domain Name Service*) donde la necesidad de un servidor de nombres desaparece, y los nodos son capaces de realizar el descubrimiento mediante mensajes

[4] http://www.freedesktop.org/wiki/Software/systemd/

uno-a-muchos [45]. Implementaciones de este protocolo, como **Bonjour**[5], **Avahi**[6] o alternativas similares, como**AppleTalk** (ya descontinuado) han sido evaluadas a la hora de buscar una solución a este problema. Sin embargo, estas y otras propuestas 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** desaparece la necesidad con un servidor de nombres y es posible realizar operaciones de descubrimiento de servicios, este y otros protocolos similares asumen que la información de un nodo presente de una red local es de interés para el resto de miembros de la misma, lo cual dificulta la independencia de un conjunto de equipos frente al resto en el mismo espacio de direcciones.

7.3.2.1. MarcoPolo, el protocolo de descubrimiento de servicios

Una de las características clave del sistema consiste en la escalabilidad del mismo en tiempo de ejecución: no es necesario conocer qué nodos participan en el sistema hasta que no se requiera de los mismos. Además, se pretende optimizar al máximo cada uno de los nodos del sistema por separado, por lo que designar a uno de ellos como "autoridad" frente a la que el resto de nodos se registren y esta actúe posteriormente como 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, que es compartido con una gran cantidad de equipos adicionales, no recaería en dicho coordinador. Esto implica que las direcciones de cada nodo son asignadas por un servidor DHCP sobre el que no se tiene control y cuyas asignaciones son dadas por intervalos de tiempo pequeños[7]. 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 serán denominadas "servicios"). Un nodo puede contar con un conjunto de servicios diferente al de sus vecinos, y por tanto colaborará en las tareas para las que disponga delos recursos necesarios.

Motivada por esta serie de características surge la necesidad de crear un protocolo de descubrimiento de nodos y servicios basado principalmente en los servicios que estos pueden (y desean) ofrecer. 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

[5] https://developer.apple.com/bonjour/index.html
[6] http://avahi.org/
[7] 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.

aproximación para satisfacer estas necesidades surge el protocolo de descubrimiento de servicios **MarcoPolo**.

Introducción

MarcoPolo es un protocolo de descubrimiento de servicios cuya dinámica y nombre se inspiran en el juego homónimo[8], 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 (e 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.

Con el objetivo de posibilitar la coexistencia de varias "mallas" de nodos independientes (donde los servicios ofrecidos por un nodo sean conocidos y consecuentemente aprovechables únicamente por el resto) 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 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).

Funcionamiento

El protocolo se basa en el envío de mensajes a un grupo *multicast* acordado previamente (existiendo un grupo designado por defecto). Los nodos suscritos a dicho grupo recibirán dicho mensaje y lo procesarán, emitiendo una respuesta dirigida únicamente al nodo que ha realizado la solicitud en caso de que el mensaje solicite un servicio que el nodo ofrezca. En caso contrario no se emitirá ninguna respuesta. Dicho mensaje incluirá información adicional, como el método de acceso al servicio, estado del nodo, etcétera.

El solicitante esperará durante un tiempo pequeño a que haya respuesta por parte del resto de nodos, recogiendo todos los mensajes recibidos. Una vez que el tiempo de espera haya pasado, se retornarán los resultados al programa o usuario solicitante.

Comandos

Los mensajes utilizados se denominan *comandos* y contienen las consultas sobre un servicio, nodo o información sobre la propia *malla* que se desea conocer, así como la respuesta a dichas consultas. Dichos comandos son enviados como cadenas de texto que almacenan la información en estructuras de datos JSON (*JavaScript Object Notation*)

[8] https://en.wikipedia.org/wiki/Marco_Polo_%28game%29

debido a la gran legibilidad de estas por humanos y máquinas, y la popularidad de este formato como mecanismo de transmisión de información, que ha propiciado la aparición de una gran cantidad de herramientas disponibles para su creación y procesado.

Nombre	**Emisor**	**Función**	**Información**	**Respuesta esperada**	**Protocolo**
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 o ninguna si no existe ningún nodo.	UDP *multicast* al puerto 1338.
Polo	Polo	Notificar de la presencia de este nodo al recibir un mensaje **marco**	Información opcional sobre el nodo (e.g. estado, *hostname*...), incluyendo opcionalmente información adicional	Ninguna	UDP *unicast* al puerto efímero del nodo solicitante.
Request-For	Marco	Conocer todos los nodos que ofrecen un servicio a través de su identificador único en el sistema	Identificador del servicio a descubrir	Un mensaje *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.	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 se encuentra activo.	UDP *unicast* al puerto 1338.

Cuadro 7.3: Comandos del protocolo MarcoPolo

7.3.2.2. Esquemas de comunicación

El comando Marco se envía al grupo *multicast* definido en la configuración de la instancia local de **Marco** o a aquel definido en tiempo de ejecución. 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 (todos los mensajes son intercambiados utilizando el protocolo **UDP**) se fija un tiempo de espera, durante el cual se reciben y acumulan todas las respuestas. Al pasar dicho tiempo, se retornan los resultados y los mensajes recibidos posteriormente son ignorados.

7.3.2.3. 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.

Estos ejecutables están diseñados para ser iniciados al arranque el equipo, aprovechando para ello las herramientas que el este provee[9], 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**[10], que simplifica el trabajo con la API, en particular a la hora de crear *sockets* asíncronos.

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* [46]).

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 los servicios a ofrecer sigue la sintaxis de un fichero **JSON**[11]. Todos estos

[9] Los ejecutables han sido configurados para ser compatibles con los gestores de arranque **init** y el más reciente **systemd**.

[10] https://twistedmatrix.com/trac/

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

ficheros son leídos al arrancar el ejecutable, y su modificación no tendrá efectos hasta la próxima vez que se inicie el servicio.

```
{
    "id": "statusmonitor",
    "params":{
        "port":1342
    },
    "groups":["224.0.0.112", "224.0.0.114"],
    "disabled": false
```

Listing 7.1: Un archivo que describe el servicio status monitor

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.
- **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 almacenan en los ficheros `/var/run/polod.pid` y `/var/run/marcod.pid` estos identificadores, que pueden ser aprovechado por el gestor de arranque del proceso.

7.3.2.4. 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`[12] 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.

[12] http://www.tldp.org/HOWTO/HighQuality-Apps-HOWTO/boot.html

7.3.2.5. Conexiones con MarcoPolo

La funcionalidad de **MarcoPolo** no se limita al descubrimiento de los servicios del sistema, por lo que es necesario ofrecer a los usuarios del clúster herramientas que permitan integrar sus aplicaciones distribuidas con estos servicios. Dichas herramientas, conocidas 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 **C**, **C++**, **Python** y **Java**. Todos ellos son consistentes entre sí, manteniendo 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`) [47], 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, definidas en la documentación de referencia de MarcoPolo (ver anexo técnico de MarcoPolo). 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

7.3.2.6. 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.

`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 [-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 7.9: Opciones de uso de `marcodiscover`

`marcoinstallkey`

Esta utilidad responde a la necesidad de instalar una clave pública en un nodo para poder acceder al mismo de forma remota sin necesidad de un par usuario-contraseña. Utiliza una llamada a marcodiscover internamente y aprovecha la información retornada para ejecutar el comando `ssh-copy-id` incluido en **OpenSSH**.

```
Usage: marcoinstallkey [-h|-n] [-i [identity_file]]
[-p port] [[-o <ssh -o options>] ...] [-u user]
Arguments
 -h, --help     show this help message and exit
 -i [identity_file]   Use only the key(s) contained in identity_file
                      (rather than looking for identities
                      via ssh-add(1) or in the default_ID_file).
                      If the filename does not end in .pub this is added.
                      If the filename is omitted, the default_ID_file is used.
                      Note that this can be used to ensure that the keys
                      copied have the comment one prefers and/or
                      extra options applied,
                      by ensuring that the key file has these set
                      as preferred before the copy is attempted.
 -p [port]
 -o [ssh_option]
 -n                   dry run
 -u                   user
```

FIGURA 7.10: Mensaje de ayuda de marcoinstallkey

`marcoscp`

Utilizando una llamada a `marcodiscover`, realiza una copia de los ficheros utilizando **scp**.

7.3.3. Gestión de los usuarios

El acceso a cualquier nodo del sistema debe realizarse mediante un sistema de credenciales homogéneo. Dicho enfoque es el propio de la infraestructura en la que el sistema se integra, que centraliza dicho repositorio de credenciales en un único punto.

Un primer intento de posibilitar esta propiedad consistió en 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 añadan 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 posterior reemplazo.

Habiendo descartado dicha estrategia, queda como alternativa más adecuada a las necesidades del sistema el uso del sistema de credenciales ya existente.

La infraestructura del centro comprende varios servicios que interactúan entre sí, siendo el pilar clave el servidor LDAP (*Lightweight Directory Access Protocol*). 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.

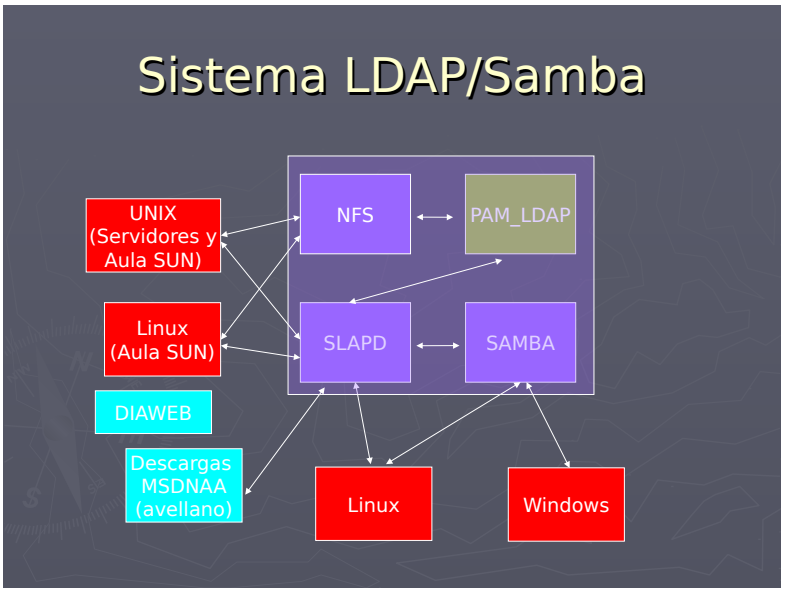

FIGURA 7.11: Esquema de los diferentes componentes del sistema de autenticación y gestión de archivos, así como de una serie de componentes adicionales. Obsérvese la interacción entre los componentes situados en el rectángulo interior

7.3.3.1. 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*).

7.3.3.2. 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. Se facilitará la replicación de servicios con herramientas creadas a tal efecto (ver 7.3.2.6) en su lugar.

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

`pam_mkpolohomedir`

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 `pam_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.

Sin embargo, el directorio únicamente es creado en el nodo al que se accede. La filosofía del sistema implica 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 anula dicha transparencia. Es 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_mkpolohomedir`, un módulo basado en `pam_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 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[13].

El módulo realiza la siguiente secuencia de pasos:

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[14].
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.

[13] http://pam-python.sourceforge.net

[14] 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

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

En cada uno de los nodos existirá por tanto una instancia del servicio **polousers** que se encargará de procesar las peticiones.

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 (ver 3.4.5).

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**.

Los detalles técnicos de este sistema se encuentran en la documentación técnica adjunta.

7.3.4. Operaciones auxiliares

Sumada a las herramientas descritas anteriormente, es necesario llevar a cabo una serie de operaciones que posibiliten la realización de diferentes tareas de administración, acceso a recursos, etcétera.

7.3.4.1. 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*) [48], un estándar *de facto* [49] 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** (*Trivial File Transfer Protocol*) 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 [50]. 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 [51].

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 implementen esta funcionalidad. Es por ello que el uso de **PXE** como herramienta de arranque debe ser desestimado.

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 (*fork*) del proyecto **rasbpian-ua-netinst** [52]. 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[15].
- Nuevo *script* de carga del *software* en la tarjeta SD (en el paquete original se delega a utilidades de terceros).
- Detección del proveedor del sistema operativo sin configuración previa utilizando **MarcoPolo**.

[15] **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

MarcoBootstrap

MarcoBootstrap es una herramienta de gestión del sistema distribuido, centrada principalmente en la instalación y actualización del sistema operativo (de ahí su nombre). `marco-netinst` obtiene una imagen del sistema operativo de esta herramienta, que ofrecerá una interfaz de gestión de las diferentes imágenes. A través de esta interfaz es posible además programar operaciones de actualización y reinicio del sistema y es extensible a otro tipo de operaciones.

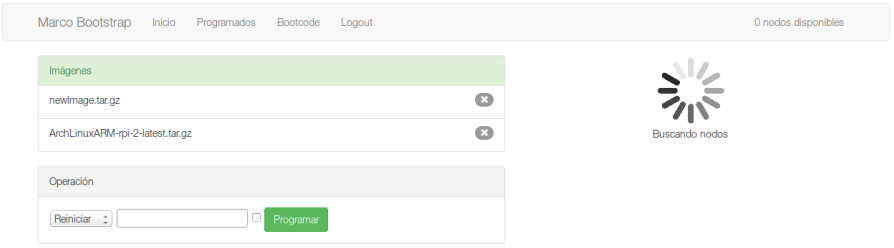

FIGURA 7.12: Interfaz administrativa de MarcoBootstrap

La especificación en detalle del funcionamiento de estas dos herramientas se detalla en el anexo técnico correspondiente.

7.3.5. Bibliotecas

Se han creado además una serie de bibliotecas que responden a diferentes necesidades dentro del sistema.

7.3.5.1. quick2wire-cpp-api

Uno de los periféricos presentes en la Raspberry Pi de interés en el desarrollo del sistema es el puerto **GPIO** *General Purpose Input-Output*. Dicho puerto permite a los usuarios del sistema analizar de forma visual el comportamiento de una aplicación distribuida, ser utilizado como indicador del estado de la máquina, etcétera.

El funcionamiento del puerto presenta un problema para el desarrollo del proyecto. El acceso al hardware se consigue mediante el acceso a una serie de direcciones de memoria sobre las que se definen diferentes valores (dirección de cada uno de los pines, valor, etcétera). Dichas direcciones de memoria se definen en el fichero `/dev/mem`, que representa la memoria física presente en todo el nodo, así como este tipo de periféricos.

Debido al riesgo que conlleva el acceso a este dispositivo por cualquier usuario del sistema, únicamente el superusuario tiene permisos para manipular el mismo (poder modificar este fichero implica ganar control absoluto sobre la memoria del sistema). Esto implica que el acceso al puerto **GPIO** está restringido al superusuario o alguien con permisos para hacerse pasar por este (mediante la herramienta `sudo`).

Quedando descartada la opción de elevar los privilegios de todos los usuario para que puedan hacer uso del puerto, es necesario buscar alternativas. Se requiere una solución más eficaz que otorgar permisos de forma temporal o contar con la supervisión de un administrador o profesor durante el uso del sistema.

Tras la revisión de diferentes alternativas de terceros existentes que proporcionan el acceso al puerto GPIO, como **wiringPi2**[16], **RPi.GPIO**[17] y **quick2wire**[18], únicamente esta última implementa un mecanismo que permite a los usuarios utilizar el puerto GPIO sin permisos. Por desgracia, esta biblioteca está implementada únicamente en Python, y el uso del puerto en el sistema está inicialmente planteado para el uso en aplicaciones creadas en C/C++ debido a que son los lenguajes utilizados en la asignatura **Arquitectura de Computadores**. Se valora la posibilidad de crear una traducción de la biblioteca quick2wire-python-api a C/C++, y dicha propuesta es admitida como camino de solución tras un prototipo inicial.

Funcionamiento

La biblioteca original aprovecha otro producto de **quick2wire**, la aplicación **gpio-admin**[19]. Escrita en C, dicha aplicación permite realizar operaciones sobre las direcciones de memoria que se corresponden con el puerto **GPIO**. Mediante la activación de del bit de usuario (`SETUID`) en el fichero ejecutable es posible hacer que el usuario que utiliza la herramienta sea a efectos prácticos el superusuario. Este sistema es similar al de la herramienta **passwd** o **ping**.

La herramienta establece una correspondencia (*"mapping"*) entre las direcciones asociadas al GPIO y un directorio virtual en `/sys/class/gpio/`[20]. Posteriormente se utiliza esta herramienta mediante llamadas al sistema a través de la biblioteca. Es por ello que **quick2wire-cpp-api** (y la implementación original en Python) pueden considerarse *wrappers* de esta utilidades, aportando poca funcionalidad más allá de una capa de abstracción.

[16] http://wiringpi.com/
[17] https://pypi.python.org/pypi/RPi.GPIO
[18] https://github.com/quick2wire-python-api
[19] https://github.com/quick2wire/quick2wire-gpio-admin/
[20] En la biblioteca original, escrita para Raspbian, la correspondencia se realizaba con `/sys/devices/virtual/gpio/`. Se ha modificado el código para posibilitar la compatibilidad con Arch Linux

Descripción del código

La biblioteca es compatible con C y C++ ofreciendo un conjunto de llamadas similares a la implementación original en Python. Sólo se ha implementado aquella funcionalidad necesaria para posibilitar el acceso a los pines GPIO (y actualmente únicamente en modo salida, pues es el único modo cuyo uso se plantea, delegando el resto de tareas de desarrollo a líneas de trabajo futuro).

7.4. Servicios auxiliares

7.4.1. Compilador cruzado

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.

Una primera propuesta de solución es contar con una instancia virtualizada del sistema operativo de las placas en un equipo más potente, utilizando QEMU como virtualizador.

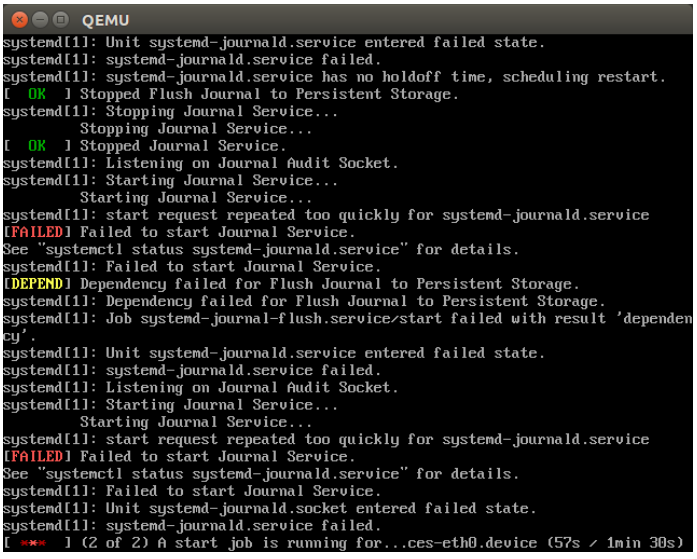

FIGURA 7.13: Arch Linux ARM virtualizado arrancando en QEMU. Nótense los diferentes errores fruto de una configuración incompleta (no se terminó de refinar el prototipo, siendo el único objetivo conseguir una compilación exitosa virtualizada). Al abandonar esta propuesta, no se invirtió más tiempo en el prototipo.

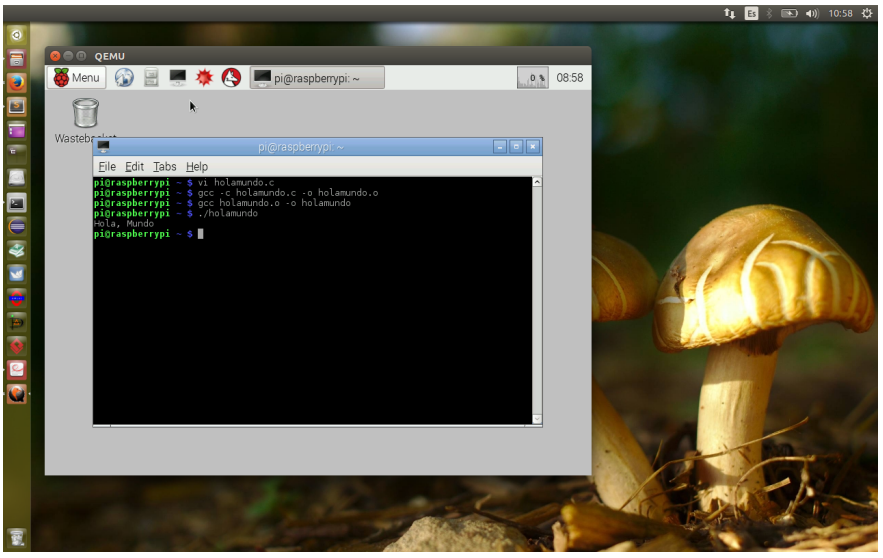

FIGURA 7.14: Raspbian con interfaz gráfica ejecutando un proceso de compilación sobre QEMU en Ubuntu.

Los resultados de esta propuesta son prometedores. No obstante, la solución implica que todos los usuarios deben contar con acceso físico a la instancia virtual (o configurar una en su propia máquina). Aspectos como la dificultad de configuración de QEMU dificultan la viabilidad de la propuesta pues hace de su administración una tarea compleja. Se opta por descartar esta solución y utilizar soluciones que faciliten la integración en un sistema multiusuario, como un compilador distribuido.

7.4.1.1. Creación de un compilador cruzado

Un compilador cruzado (*cross-compiler*) es una herramienta capaz de generar código máquina para una arquitectura determinada utilizando un equipo equipado con un juego de instrucciones diferente. El uso de compiladores cruzados es habitual a la hora de desarrollar aplicaciones para sistemas embebidos o móviles, debido a la dificultad o incluso incapacidad de realizar las diferentes etapas de desarrollo en el propio equipo. Uno de los problemas que aparecen a la hora de utilizar este tipo de equipos es la escasa velocidad de procesamiento, que hace de tareas con gran demanda de recursos como una compilación un proceso tedioso (ver 3.8.3).

El compilador ha sido creado utilizando la herramienta **crosstool-ng**. No se ha generado únicamente un compilador, sino un conjunto completo de herramientas para el desarrollo de aplicaciones para las arquitecturas ARMv6 y ARMv7. **Crosstool-ng** permite generar una "cadena de herramientas" (*crosstool* en inglés), que incluye una copia de las bibliotecas estándar de C (**glibc** en este caso), depurador y bibliotecas de depuración, además de permitir optimizar las herramientas creadas para una plataforma

determinada o un conjunto de instrucciones determinado dentro de la plataforma (por ejemplo, para las operaciones en punto flotante).

7.4.1.2. Compilación distribuida

Si bien el uso de un compilador distribuido permite aprovechar la potencia de un segundo equipo para crear *software*, se requiere previamente la instalación de la cadena de herramientas (o en el peor de los casos, la generación de la misma, un proceso que suele requerir entre 30 y 60 minutos). Por ello, se plantea una solución que sea más transparente para el usuario final.

distcc es una herramienta que permite gestionar trabajos de compilación distribuida en un esquema cliente-servidor. Distribuye diferentes etapas de compilación a todos los nodos presentes en una red que cuenten con la aplicación servidor activada, estableciendo los diferentes mecanismos de sincronización y validación de resultados oportunos, haciendo que su uso sea completamente transparente al usuario (incluso en caso de fallo es capaz de realizar el trabajo encomendado, realizando la compilación en el propio equipo). Un trabajo de compilación en **distcc** con gcc se encarga de la siguiente forma:

```
distcc gcc -c main.c
```

Como se puede observar, todas las opciones de gcc se mantienen en dicha llamada, siendo necesario únicamente añadir distcc al inicio de la llamada. Por ello es una herramienta muy sencilla de utilizar e integrable en cualquier tarea de compilación[21] [22].

Distcc utiliza por defecto el compilador indicado por el cliente en la llamada al proceso cliente, utilizando dicho compilador en el servidor. En el caso de una compilación no cruzada esta situación no causa problemas, sin embargo, en el presente caso es necesario realizar una serie de modificaciones en el servidor para posibilitar el uso de la cadena de herramientas creada.

Creando enlaces simbólicos que no incluyan el prefijo de los binarios generados por la cadena de herramientas (generalmente del estilo `arm-linux-gnueabi-armhfv7-gcc`) y modificando la variable `$PATH` para incluir de forma prioritaria estos binarios en el entorno del proceso servidor (incluyendo dichas modificaciones en un *script* de inicio) el servidor utilizará el compilador cruzado para las tareas que se le encarguen, sin entrar en conflicto con el resto del sistema.

[21] Un ejemplo son los *flags* de la herramienta **make**. Únicamente es necesario realizar la llamada `make CC="distcc gccÇXX="distcc g++"` para utilizar distcc en lugar del compilador predeterminado.

[22] También es posible crear un alias para todo el sistema que realice una llamada a `distcc` cada vez que el comando `gcc` es invocado. Esta configuración no se ha implementado con el objetivo de dar al usuario final la opción de utilizar el compilador local o el distribuido.

7.4.1.3. Integración con MarcoPolo

El servidor distcc es detectable a través de MarcoPolo gracias a la herramienta **Marco-Manager** (ver 7.5.4).

7.4.1.4. 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:

OpenMPI

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

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

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

OpenSSH

La versión de OpenSSH modificada para incluir las mejoras del proyecto SSH-HPN (ver 4.10.2) que se instala en el sistema ha sido compilada utilizando esta herramienta, con un tiempo de trabajo menor a un minutos.

Esta herramienta se ejecuta en el servidor secundario.

7.4.2. Nodos secundarios

Una serie de nodos secundarios han sido instalados en el sistema con el único propósito de realizar una serie de tareas que tienen como objeto simplificar u optimizar las diferentes tareas asociadas a los nodos. En ningún caso el uso de estos nodos secundarios implica una dependencia de los mismos, siendo posible prescindir de ellos sin consecuencias graves para el sistema como conjunto.

7.4.2.1. Servidor

El equipo utilizado como servidor es un equipo en desuso presente en la Facultad de Ciencias, que se encuentra disponible para cualquier alumno interesado en integrarlo en un Trabajo de Fin de Grado. Las características del equipo, una estación de trabajo Sun Ultra, son las siguientes:

- Procesador AMD Opteron 1200 de dos núcleos (arquitectura x86).
- 4 GB de memoria RAM.
- Partición de 50 GB para el sistema operativo del servidor.

Las características del sistema lo hacen idóneo para su uso como servidor de compilación distribuida. Se ha generado en el mismo cadenas de herramientas compatibles con ARMv7 y ARMv6, que se integran con la herramienta **marcomanager** (ver 7.5.4).

Además, este servidor incluye el servidor **marcoboostrap** (ver 7.3.4.1), incluyendo una interfaz de gestión para el administrador y un conjunto de imágenes de sistemas operativos para los usuarios finales.

7.4.2.2. Servidor LDAP

Con el objetivo de mejorar la accesibilidad del sistema, se delega la gestión de las diferentes cuentas de usuario a un servidor LDAP preexistente en la infraestructura. Dicho servidor se encuentra alojado en la dirección `ldap1.aulas.cie.usal.es` y es aprovechado por los nodos principales a través de nsswitch (ver 7.3.3).

7.5. Aplicaciones

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 aplicaciones distribuidas.

Muchas de las aplicaciones distribuidas utilizadas incluyen varias utilidades 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.

7.5.1. 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* [24]. Esto es posible debido a que según la especificación del estándar de Websocket, la *Same-Origin Policy* no es utilizada de la misma forma que en peticiones HTTP [53].

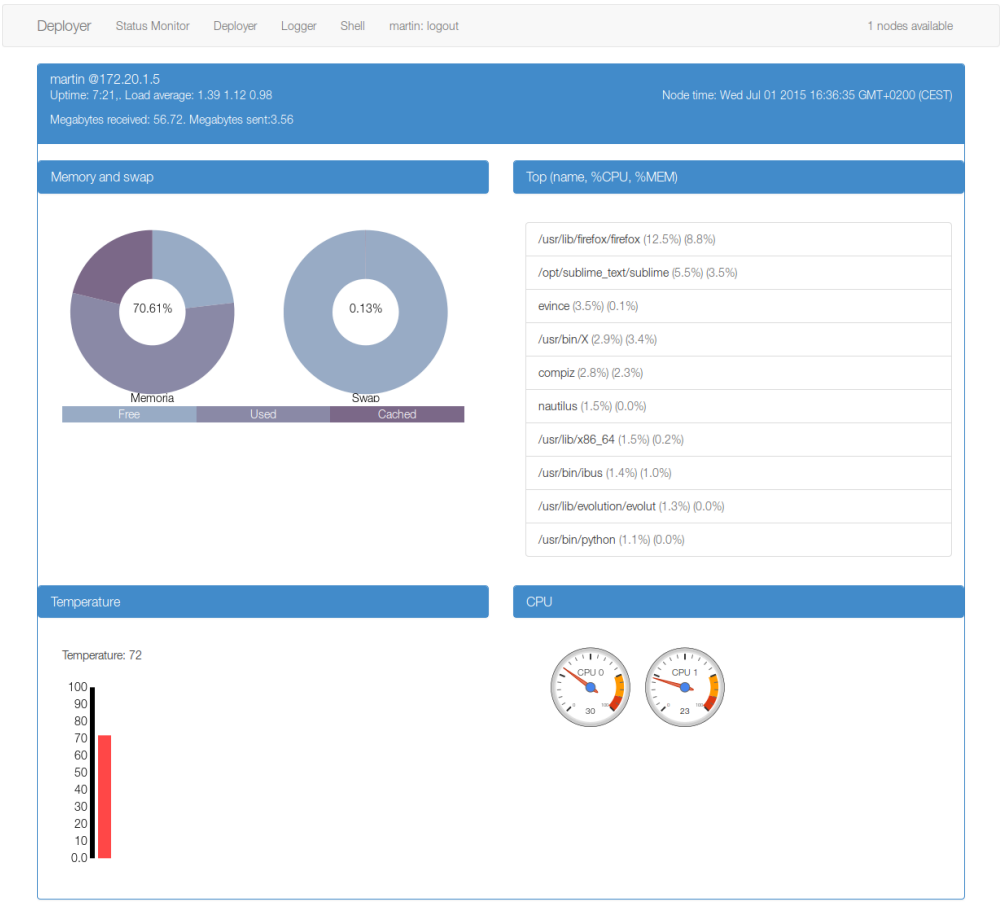

FIGURA 7.15: Vista de la interfaz web una vez obtenidos los nodos y establecida la conexión a los mismos. Se observa el porcentaje de memoria y principal y de intercambio utilizadas, la temperatura del procesador, los procesos con más consumo de CPU y el porcentaje de CPU utilizado en cada núcleo.

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[23], un servidor web ligero asíncrono implementado íntegramente en Python y mantenido por Facebook.

7.5.2. Deployer

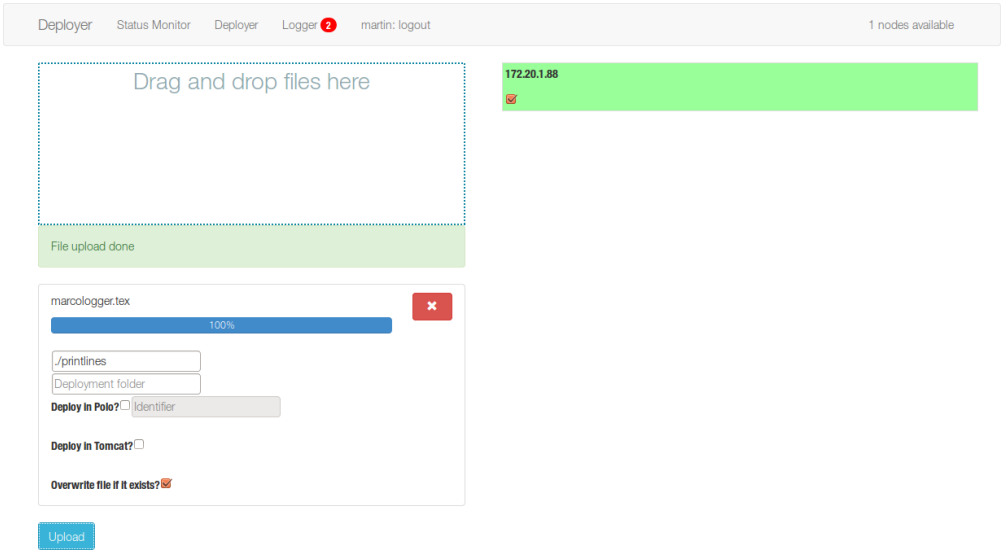

FIGURA 7.16: Interfaz web del deployer. A la izquierda figuran los controles y a la derecha la lista de nodos sobre los que se puede realizar el despliegue

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 realizadas, ver 7.7), de replicar de una forma sencilla un ejecutable entre los diferentes nodos que conformarán el sistema distribuido.

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

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**.

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

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[24].

[24] jquery.com/

7.5.3. Logger

Uno de los problemas típicos que dificultan el desarrollo de aplicaciones distribuidas son las tareas de análisis y depuración del código desarrollado. Soluciones "creativas" como utilizar el puerto GPIO[25] para este tipo de tareas son efectivas, si bien no aplicables a cualquier dispositivo o aplicación (por ejemplo, un ejecutable que no pueda ser modificado). Se debe por tanto buscar una solución complementaria para este tipo de casos.

El **logger** es una herramienta que, a través de WebSockets y redireccionamiento de *streams*, permite enviar la salida que una aplicación emite a través de los canales estándares (**stdout**, **stderr**) a una interfaz web.

Esta herramienta se integra con el **deployer**, mostrando la salida por pantalla del comando ejecutado por pantalla. Una vez que se completa el despliegue, se ejecutará el comando indicado en el panel de subida.

FIGURA 7.17: La herramienta logger en ejecución

El usuario puede además enviar señales de terminación al proceso a través de la interfaz, útiles en situaciones en las que el programa se comporta de forma errónea o incontrolada. Se enviará una señal de terminación que, en caso de no surtir efecto, será sucedida por una señal **kill**.

La gestión de esta funcionalidad se realiza a través del bucle de eventos de **Tornado**, añadiendo al bucle de eventos de la aplicación web el descriptor de fichero de las salidas **stdout** y **stderr** del proceso que ejecuta el comando.

[25] http://elinux.org/RPi_Serial_Connection Uso del puerto GPIO como conexión serie.

7.5.3.1. Shell

Marcoshell sigue un funcionamiento similar al de **logger**. La diferencia clave es la independencia de **deployer**. La herramienta simula una consola que implementa un subconjunto de la funcionalidad esperada de una consola tradicional. Incluye también funcionalidad para el envío de señales de terminación.

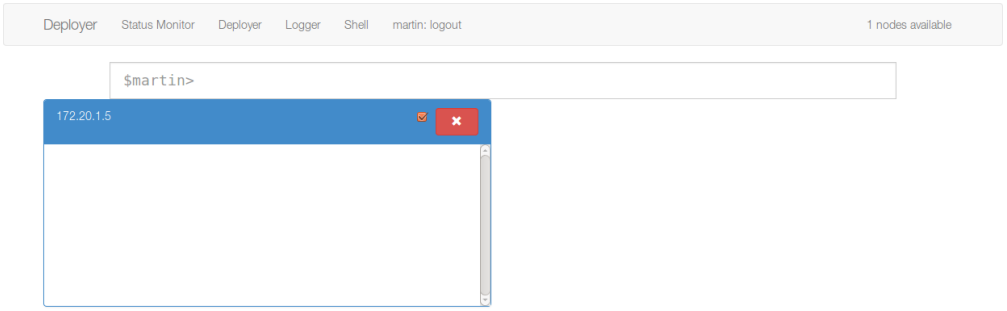

FIGURA 7.18: La aplicación shell en ejecución.

7.5.4. Marcomanager

Marcomanager es una pequeña utilidad de gestión de servicios. Es diseñada con el propósito específico de posibilitar la integración con **MarcoPolo** de diferentes componentes de terceros que, por su diseño, no pueden (o la modificación es compleja) utilizar cualquier otro mecanismo disponible, como los diferentes *bindings*. Sin embargo, durante su desarrollo se añaden diferentes mecanismos para la programación de diferentes tareas en el futuro, siendo el resultado final una herramienta de integración y planificación que implementa una parte de la funcionalidad ofrecida por **cron**.

Todos los detalles técnicos sobre esta herramienta, manuales de usuario y programador y la descripción de las diferentes fases del desarrollo se encuentran en el anexo técnico correspondiente.

7.6. Vista general del sistema

El sistema finalmente se compone de la siguiente combinación de *software* y *hardware*.

7.6.1. Componentes *hardware*

- Un conjunto escalable de nodos Rasbperry Pi (actualmente 9).
- Un servidor Xubuntu accesorio.
- Todos los mecanismos de interconexión y alimentación requeridos.

7.6.2. Componentes *software* en los nodos Raspberry Pi

- Una instancia de **MarcoPolo** como aglutinante de todos los servicios del sistema, tanto los creados como los servicios de terceras partes.
- Una instancia de la biblioteca **OpenMPI**.
- Diferentes instancias del contenedor de servicios **Tomcat** para cada usuario, pudiendo existir una instancia global.
- Bindings de **MarcoPolo** para C, C++, Java y Python
- Una instancia de los servicios **Deployer**, **StatusMonitor** y **MarcoShell** aglutinados en la herramienta **MarcoDeployer**.
- Diferentes utilidades de manipulación de **MarcoPolo**.
- Una instancia de MarcoManager para configurar **distcc** o cualquier otro servicio de tercero no enlazable con **MarcoPolo** de otra forma.
- Una instancia del servicio **Marcobootstrap-slave** para la gestión de operaciones de reinico.
- El módulo de PAM **pam_mkpolohomedir** y una instancia de **polousers** para la gestión de usuarios.

7.6.3. Componentes *software* en el servidor Sun

- Una instancia del servidor **distcc** que utiliza la cadena de herramientas generada por **crosstool-ng** vinculada a **MarcoManager**.
- Una instancia del servicio **Marcobootstrap-backend** para ejecutar diferentes operaciones de administración y albergar las imágenes del sistema operativo.

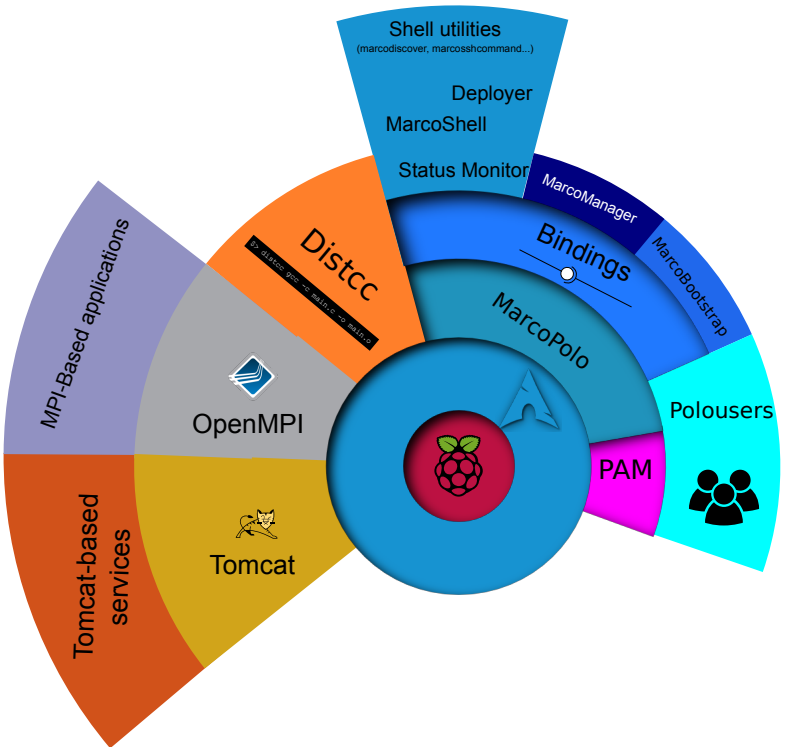

FIGURA 7.19: Representación artística de un nodo Raspberry Pi

7.7. Evaluación y pruebas

7.7.1. Pruebas unitarias

Uno de los mecanismos de apoyo a la detección de errores en el código es la realización de pruebas unitarias (3.7.1). Gracias a esta práctica se ha conseguido detectar de forma temprana (antes de ejecutar el código en un contexto de pruebas) potenciales situaciones donde el comportamiento del programa difiere del esperado. De forma indirecta, la realización de pruebas unitarias ha tenido como resultado la generación de código más modular, y por tanto mejor trazable, depurable y reutilizable.

7.7.2. Pruebas en entorno aislado

Tras la fase de pruebas unitarias es necesario ejecutar el código en un ambiente controlado. Dichas pruebas se han realizado en cada una de las fases de desarrollo de prototipos. Gracias a dichas pruebas se consiguen detectar fallos no hallados en las pruebas unitarias (por ejemplo, potenciales fallos en la conexión de red como retardos o fallos de sincronización, pues se simula en dicha fase).

7.7.3. Pruebas en un entorno real

Tras las diferentes pruebas realizadas en contextos controlados es necesario verificar que el comportamiento del *software* es similar en el entorno donde este deberá operar. Estas pruebas se han realizado de forma periódica, aprovechando que las fases de desarrollo se realizan en la misma infraestructura donde se integrará finalmente.

En el anexo "Evaluación del rendimiento del sistema en un entorno real" se encuentran los resultados de diferentes pruebas en este tipo de contextos, que evalúan el correcto funcionamiento del *software* así como el tiempo de ejecución (relevante si se considera que la red de la infraestructura es compartida con un gran número de equipos, y no se tiene control sobre ella).

7.7.4. Evaluación de usuarios

Uno de los procesos escogidos para la constatación de la efectividad de las diferentes propuestas de solución a problemas planteados por los alumnos, profesores y administradores es la realización de evaluaciones de los prototipos creados así como las versiones finales de los productos *software* y *hardware*.

Dichos procesos de evaluación han sido muy efectivos a la hora de detectar potenciales fallos en las interfaces de usuario que empobrecen de forma significativa la experiencia de estos con el sistema. Además, permiten conocer necesidades no identificadas que se han traducido en nuevas características del sistema.

Se han seguido diferentes estrategias de evaluación. El método más utilizado en la fase de captura de requisitos ha sido la realización de entrevistas. Posteriormente se han utilizado prototipos de las herramientas en las fases de evaluación, tanto de forma guiada (se muestran los pasos al usuario y se evalúa su opinión) como autónoma (se observa el comportamiento del usuario con el sistema).

Tras cada evaluación se realiza un análisis, incluyendo en un informe todos los aspectos de relevancia extraídos de la sesión. Las conclusiones de dicho informe determinan las acciones a llevar a cabo.

Se ha documentado un total de 9 evaluaciones. Cada una de ellas se recoge en un informe adjunto a este documento.

7.8. Prácticas

Varias de las prácticas de las asignaturas Arquitectura de Computadores y Sistemas Distribuidos han sido implementadas y probadas en el sistema distribuido, con el objetivo de analizar la viabilidad del sistema como herramienta didáctica. El único problema detectado es el bajo rendimiento del contenedor de servicios **Tomcat** al cargar nuevos servicios en el sistema. El rendimiento del resto de fases es equivalente al de los equipos utilizados actualmente.

Además de la integración de las prácticas tal y como se plantean en las asignaturas, se han realizado las modificaciones necesarias para aprovechar herramientas integradas en el sistema como **MarcoPolo** y la biblioteca **quick2wire-cpp-api**. Integración realizada de forma trivial y de resultados satisfactorios.

Para finalizar y de forma experimental se ha analizado la integración del lenguaje Python para el desarrollo de las prácticas de la asignatura Sistemas Distribuidos, utilizando el *framework* Flask[26] como herramienta para la creación de APIs REST. Los resultados del análisis reflejan que la sustitución de la tecnología utilizada actualmente, Java, por Python simplificaría una serie de tareas como el despliegue de los diferentes servicios sin alterar el rendimiento de la solución (y en casos como las placas Raspberry Pi, mejorándolo). No obtante, es necesario valorar la etapa de aprendizaje que los alumnos tendrían que llevar a cabo, pues el currículo de la titulación no incluye el uso de este lenguaje de forma obligatoria en ninguna asignatura.

En el caso de la asignatura Arquitectura de Computadores se ha conseguido integrar **MarcoPolo** con la biblioteca MPI. MPI está diseñado de tal forma que los nodos participantes deben ser definidos durante la carga del código, mediante un parámetro en la llamada al ejecutor de trabajos:

```
mpiexec --hostfile <hostfile.txt>./executable
```

Esta dinámica hace incompatible el uso de cualquiera de los *bindings* de MarcoPolo, por lo que se debe plantear una alternativa. Aprovechando la herramienta **marcodiscover** (ver 7.3.2.6) se añade un parámetro de formato a la misma, permitiendo la creación de un archivo compatible con el formato del fichero `hostfile` que puede ser utilizado en la llamada.

```
 marcodiscover -s executable --format mpi >marcopolohostfile.txt mpiexec
--hostfile marcopolohostfile.txt ./executable
```

[26] http://flask.pocoo.org/

Incluso es posible incluir el descubrimiento en la propia llamada a **mpiexec**, mediante el uso de pseudoficheros en la consola [54].

```
mpiexec --hostfile <(marcodiscover -s executable --format mpi) ./executable
```

7.9. Instalación

Se ha instalado la totalidad o parte del software desarrollado en los siguientes equipos, con resultados satisfactorios:

- Equipo personal (Portátil x86_64 con sistema operativo Ubuntu 14.04).
- Equipo que actúa como nodo secundario (Sun Workstation con sistema operativo Xubuntu 12.04).
- Placas Raspberry Pi B versión 1 y 2 con los sistemas operativos Arch Linux ARM y Raspbian.
- Equipos Lenovo x86_64 del aula Sun de la Facultad de Ciencias con sistema operativo Debian 7.
- Instancia virtualizada de Debian 6 estable (ver anexo F).

Capítulo 8

Conclusiones y líneas de trabajo futuro

En el que se realiza un análisis retrospectivo de todas las etapas del Trabajo de Fin de Grado valorando los diferentes aspectos de las mismas y los resultados finales. Se incluyen además una serie de propuestas de trabajo futuro para el equipo de desarrollo o potenciales interesados y tareas ya planeadas que complementan el Trabajo.

El trabajo descrito en los anteriores capítulos ha probado ser de gran utilidad como síntesis de los conocimientos a adquirir según se plantea en el currículo del Grado en Ingeniería Informática de la Universidad de Salamanca. Además, el sistema ha permitido desarrollar expandir los conocimientos adquiridos en las asignaturas Sistemas Distribuidos, Redes de Computadores, Administración de Sistemas, Interacción Persona-Ordenador, Interfaces Gráficas de Usuario y Gestión de Proyectos.

Como proyecto, la gestión del mismo ha permitido realizar una aproximación más cercana a un proyecto "real" que cualquier otro trabajo del plan de estudios, aprendiendo a lidiar con una incertidumbre muy alta y compaginando etapas de aprendizaje con etapas de desarrollo. El modelo de desarrollo utilizado, una metodología ágil basada prototipos, ha probado ser una vía efectiva para las demandas y restricciones del proyecto.

Los resultados finales del sistema prueban que el desarrollo de sistemas distribuidos en hardware de bajo coste es factible e incluso ventajoso para diferentes propósitos. La relación rendimiento/coste es muy alta y sus características (como el puerto GPIO, o la pequeña demanda de espacio físico) pueden ser aprovechadas en un contexto didáctico, aportando nuevas herramientas que facilitarán la etapa de aprendizaje a los estudiantes de las asignaturas.

Sin embargo, todas las herramientas creadas son independientes de la plataforma donde se ejecutan (excepto aquellas que dependan del *hardware* presente en los nodos, como la biblioteca **quick2wire-cpp-api** o **marco-netinst**), por lo que su integración en cualquiera de los equipos de las aulas de informática de la organización no requiere adaptaciones al código fuente.

En el plano personal, la gran cantidad de conocimientos adquiridos durante todas las fases del Trabajo de Fin de Grado hacen del mismo un recurso de gran valor como síntesis y expansión de las habilidades desarrolladas durante el transcurso de los cuatro años del Grado, en particular en cuanto a las áreas de conocimiento aplicadas al desarrollo.

Sin embargo, no se plantea con el fin de las etapas de integración y desarrollo que el proyecto quede abandonado. Se plantea continuar el desarrollo con el objetivo de añadir nueva funcionalidad, explorar nuevas vías de trabajo y promocionar el uso del sistema. Siguiendo estas premisas se definen las líneas de trabajo futuro.

8.1. Líneas de trabajo futuro

- Instalar de forma definitiva (en lugar de las instalaciones de prueba realizadas) el sistema en la organización.
- Analizar los resultados de la integración del sistema y sus herramientas en las asignaturas planteadas durante el curso académico 2015-2016.
- Añadir a **MarcoPolo** la funcionalidad de publicación activa de servicios (anunciando en la red la adición de un nuevo servicio), en lugar de la publicación pasiva actual.
- Consolidar el protocolo **MarcoPolo** como herramienta de descubrimiento de servicios, portándolo a sistemas como impresoras en red, dispositivos multimedia…y facilitar su uso en redes inalámbricas.
- Promocionar los resultados de desarrollo en foros especializados en la placa Raspberry Pi.

Apéndice A

Lista de anexos

En este apéndice se detalla la lista de todos los anexos que complementan el presente documento.

- Configuración de un sistema de compilación multiplataforma con distcc y aplicación de este en un entorno distribuido.
- Evaluación del rendimiento del sistema en un entorno real.
- Estructura física.
- Gestión del proyecto.
- Informes de viabilidad de las prácticas de la asignatura Sistemas Distribuidos.
- Evaluación del rendimiento del sistema en un entorno real.
- Anexos técnicos, detallando en cada uno de ellos la especificación de los requisitos del *software*, especificación del diseño, documentación técnica, manuales de usuario y cualquier otro aspecto de relevancia.
  - Documentación técnica de MarcoPolo.
  - Documentación técnica de Marcodeployer.
  - Documentación técnica de PoloUsers.
  - Documentación técnica de MarcoBootstrap.
  - Documentación técnica de MarcoManager.
  - Documentación técnica de quick2wire-cpp-api herramientas relacionadas.
- Evaluaciones de usuario.
- Pruebas del sistema.

Apéndice B

Listado de paquetes

Paquetes alojados en el *Python Package Index*, instalables mediante gestores como **pip**.

- https://pypi.python.org/pypi/marcopolo
- https://pypi.python.org/pypi/marcopolo.bindings
- https://pypi.python.org/pypi/marcopolo-deployer
- https://pypi.python.org/pypi/marcopolo-shell
- https://pypi.python.org/pypi/marcobootstrap.backend
- https://pypi.python.org/pypi/marcopolo-manager

Apéndice C

Listado de contenidos del DVD

El DVD que acompaña a esta memoria incluye los anexos relativos al desarrollo del proyecto, el código fuente, imágenes del sistema operativo y cualquier otro fichero considerado de relevancia. El DVD sigue la siguiente estructura:

- Directorio "documentacion", en el que se incluye toda la documentación del proyecto en el siguiente árbol:

  Directorio "memoria", con una copia del repositorio del presente documento con el código LaTeXcorrespondiente y aquellos anexos no relativos a los aspectos técnicos de los productos *software* (evaluaciones de usuario, aspectos de gestión de proyectos, pruebas, estructura física...).

  Directorio "anexos" en el que se incluyen varios directorios con copias de la documentación técnica de todos los productos *software* creados (en inglés) y los aspectos relativos a la ingeniería del software. Dicha documentación ha sido generada con la herramienta Sphinx, y se incluye tanto en formato PDF como HTML. Se adjunta también, en los casos que proceda, una copia del proyecto de Visual Paradigm utilizado en las etapas de análisis y diseño.
- Directorio "código", en el que se incluyen los repositorios de código (incluyendo el directorio `.git`) de cada uno de los productos *software* creados. En este directorio se encuentra una copia de la documentación técnica, a fin de no modificar el repositorio de código.
- Directorio "imágenes", en el que se incluyen diferentes imágenes del sistema operativo utilizado en las placas Raspberry Pi.

En los casos en el que la documentación o el código fuente requiera ser compilado, se incluye un fichero **Makefile** tradicional, o en ciertos casos, un fichero **latexmk**. Toda la documentación ha sido compilada previamente.

Apéndice D

Listado de repositorios de código

Lista de los diferentes repositorios de git donde se encuentra todo el código de los productos *software* realizados.

- https://bitbucket.org/Alternhuman/marcopolo
- https://bitbucket.org/Alternhuman/deployer
- https://bitbucket.org/Alternhuman/crosstool-conffiles
- https://bitbucket.org/Alternhuman/memoria-tfg
- https://bitbucket.org/Alternhuman/marco-netinst
- https://bitbucket.org/Alternhuman/marcopolo-bindings-cpp
- https://bitbucket.org/Alternhuman/marcopolo-shell
- https://bitbucket.org/Alternhuman/quick2wire-gpio-admin
- https://bitbucket.org/Alternhuman/mpi_led
- (*Fork*) https://bitbucket.org/Alternhuman/quick2wire-python-api
- https://bitbucket.org/Alternhuman/quick2wire-cpp-api
- https://bitbucket.org/Alternhuman/marcopolo-cluster-estructura
- https://bitbucket.org/Alternhuman/marcomanager
- https://bitbucket.org/Alternhuman/polousers
- https://bitbucket.org/Alternhuman/marcopolo-doc
- https://bitbucket.org/Alternhuman/marcopolo-bindings-python
- https://bitbucket.org/Alternhuman/marcopolo-bindings-java
- https://bitbucket.org/Alternhuman/tfg-utils/src

  Paquetes descontinuados cuya funcionalidad se ha integrado en otro repositorio
- https://bitbucket.org/Alternhuman/marco-bootstrap.
- https://bitbucket.org/Alternhuman/statusmonitor

Apéndice E

Listado de puertos utilizados

Puerto	**Protocolo**	**Uso**	**Interfaz**
1337	UDP	Puerto del demonio **Marco**	127.0.1.1
1338	UDP	Puerto de la instancia de **Polo**	IP del grupo suscrito
1390	TCP (TLS/SSL)	Puerto del bindings de **Polo**	127.0.0.1
1342	TCP (HTTPS)	Puerto de **Deployer**	eth0
1442	TCP (HTTP)	Puerto no seguro de **Deployer**	eth0
1339	TCP (HTTPS)	Puerto de **Receiver**	eth0
1370	TCP (WSS)	Puerto para los WebSockets de **receiver**	eth0
1343	TCP (TLS/SSL)	Puerto de **polousers**	eth0
1345	TCP (HTTPS)	Puerto de ficheros de **Marcobootstrap-backend**	eth0
1346	TCP (HTTPS)	Puerto de la interfaz web de **Marcobootstrap-backend**	eth0
1446	TCP (HTTP)	Puerto de la interfaz web de **Marcobootstrap-backend**	eth0
1360	TCP (HTTPS)	Puerto de **Marcobootstrap-receiver**	eth0
8080	TCP (HTTP)	Puerto de servicios de **Tomcat**	eth0
8005	TCP (HTTP)	Puerto de apagado de **Tomcat**	lo
<uid>+ 10000	TCP (HTTP)	Puerto de servicios de **Tomcat** para un usuario	eth0
<uid>+ 20000	TCP (HTTP)	Puerto de apagado de **Tomcat** para un usuario	lo

Cuadro E.1: Listado de puertos utilizados en el sistema

Apéndice F

Listado de recursos en línea

Junto a la documentación y código fuente adjunto se incluyen los siguientes recursos en línea:

- Página de presentación del proyecto, incluyendo toda la documentación en HTML, descripción de todas las herramientas, galería de imágenes, enlaces a repositorios y otros contenidos relevantes en http://marcopoloproject.martinarroyo.net.
- Instancia de Redmine con todos los aspectos de relevancia del proyecto en http://redmine.martinarroyo.net.
- Una pequeña instancia de parte del *software* ejecutándose en las direcciones IP **172.20.1.62**, **172.20.1.63** y **172.20.1.64**, accesibles únicamente desde la red local de la Universidad de Salamanca. El usuario y la contraseña de acceso a un usuario no privilegiado es **marcopolo**.

Bibliografía

[1] Lloyd Seth, "Ultimate physical limits to computation," *Nature*, vol. 406, pp. 1047–1054, ago 2000. 10.1038/35023282.

[2] A. S. Tanenbaum and M. v. Steen, *Distributed Systems: Principles and Paradigms (2Nd Edition)*, ch. 1. In [58], 2006.

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

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

[5] The Linux Documentation Project, "NSSWITCH.CONF(5)." http://man7.org/linux/man-pages/man5/nsswitch.conf.5.html, 2013.

[6] A. Fettig, *Twisted Network Programming Essentials*. O'Reilly Media, Inc., 2005.

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

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

[9] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems: Concepts and Design*, ch. Chapter 15. In [55], 5th ed., 2011.

[10] A. S. Tanenbaum and M. v. Steen, *Distributed Systems: Principles and Paradigms (2Nd Edition)*, ch. 6. In [58], 2006.

[11] M. Cooper, Y. Dzambasow, P. Hesse, S. Joseph, and R. Nicholas, "Internet X.509 Public Key Infrastructure: Certification Path Building." RFC 4158 (Informational), sep 2005.

[12] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile." RFC 5280 (Proposed Standard), may 2008. Updated by RFC 6818.

[13] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2." RFC 5246 (Proposed Standard), ago 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), sep 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, jun 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), ago 2000.

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

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

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

[26] Unidad de Evaluación de la Calidad, "Estadísticas generales de la Universidad de Salamanca." http://campus.usal.es/~estadisticasgenerales/contenido/grado.html, 2015.

[27] 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, oct 2014.

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

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

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

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

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

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

[34] 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.

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

[36] T. Nishinaga, "Raspberry Pi 2 Linpack Benchmark ," feb 2015.

[37] ELinux.org, "RPi Performance," ago 2012.

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

[39] V. R. Basil and A. J. Turner, "Iterative enhancement: A practical technique for software development," *IEEE Transactions on Software Engineering*, vol. 1, no. 4, pp. 390–396, 1975.

[40] Pacman Development Team, "Pacman Home Page," 2014.

[41] freedesktop.org, "systemd," 2015.

[42] M. van Smoorenburg, "INIT(8)," 2004.

[43] The Arch Linux Project, "The Arch Way," 2015.

[44] Distrowatch.com, "Distrowatch.com Search Distributions," 2015.

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

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

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

[48] Corporation, Intel and Systemsoft, "Preboot Execution Environment (PXE) Specification," *Preboot Execution Environment (PXE) Specification*, sep 1999.

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

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

[51] Corporation, Intel and Systemsoft, "Preboot Execution Environment (PXE) Specification," in *Preboot Execution Environment (PXE) Specification* [48].

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

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

[54] Free Software Foundation, "Bash Reference Manual." http://www.gnu.org/software/bash/manual/bashref.html#Process-Substitution, 2015.

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

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

[57] *A Guide To The Project Management Body Of Knowledge (PMBOK Guides)*. Project Management Institute, 2004.

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