Universidad Politécnica de Madrid
Escuela Técnica Superior de Ingenieros de Telecomunicación

DISEÑO E IMPLEMENTACIÓN DE UN SISTEMA DE DISTRIBUCIÓN DE COMPONENTES OSGI PARA UNA PASARELA DOMÓTICA

TRABAJO FIN DE MÁSTER

Jaime Caffarel Rodríguez

2014

Universidad Politécnica de Madrid
Escuela Técnica Superior de Ingenieros de Telecomunicación

Máster Universitario en
Ingeniería de Redes y Servicios Telemáticos

TRABAJO FIN DE MÁSTER

DISEÑO E IMPLEMENTACIÓN DE UN SISTEMA DE DISTRIBUCIÓN DE COMPONENTES OSGI PARA UNA PASARELA DOMÓTICA

Autor
Jaime Caffarel Rodríguez

Director
David Fernández Cambronero

Departamento de Ingeniería de Sistemas Telemáticos

2014

Resumen

La automatización de edificios se define como la utilización de diversas tecnologías que permiten monitorizar y controlar los dispositivos y sistemas eléctricos presentes en un edificio. Los sistemas de gestión de edificios son sistemas automatizados que permiten el control de dichos componentes, como por ejemplo la iluminación, la climatización o los componentes antiincendios o de seguridad. A diferencia de Internet, donde la familia de protocolos TCP/IP es universalmente utilizada para la comunicación de todos sus sistemas, en la automatización de edificios existen multitud de protocolos de comunicaciones disponibles, en la mayoría de los casos incompatibles entre sí o cuya comunicación no resulta fácil de implementar, lo cual dificulta tanto el desarrollo de este tipo de sistemas de gestión como la creación de nuevas aplicaciones para los ya existentes.

Por otra parte, la utilización de un modelo de componentes dinámico basado en Java para implementar un sistema de automatización de edificios proporciona algunas ventajas, principalmente por la posibilidad de crear aplicaciones específicas para el edificio, generar sistemas más modulares y adaptables a las tecnologías de control existentes, así como la posibilidad de utilizar diferentes plataformas hardware para su implantación debido a las características del lenguaje Java. Además, la utilización de Open Service Gateway initiative (OSGi), una especificación de sistemas modulares utilizando Java puede facilitar la implementación de este tipo de sistemas. En este sentido, el Centro de Domótica Integral (CeDInt) está desarrollando una plataforma basada en OSGi para el control domótico de un edificio que permita la comunicación con diferentes protocolos domóticos así como la integración de aplicaciones en dicha plataforma mediante una interfaz común de desarrollo.

Además, uno de los objetivos que se pretende con esta plataforma es aprovechar la modularidad que proporciona OSGi para instalar dinámicamente nuevos componentes en la misma, ya sean éstos nuevas tecnologías de control (drivers) de los protocolos del edificio o nuevas aplicaciones que aumenten la funcionalidad del sistema. Esta integración dinámica (hot deployment) de nuevos components todavía no ha sido implementada en la plataforma. Por ello, el objetivo principal de este Trabajo Fin de Máster es el desarrollo de esta nueva funcionalidad que permita la descarga y puesta en ejecución de componentes en la plataforma de control domótico de forma remota, a la vez que permita realizar dicho control en varias de estas plataformas de forma simultánea.

Para cumplir con estos objetivos, en este trabajo se propone la utilización de un contenedor OSGi Apache Karaf que permite extender las funcionalidades del sistema desarrollado en el CeDInt así como el diseño y desarrollo de una aplicación web de

control que permita la descarga, instalación y ejecución de nuevos componentes en varios de estos sistemas simultáneamente.

Abstract

Building automation can be defined as the use of different technologies which allows monitoring and controlling devices, electrical systems and other appliances within a building. In this regard, Building Management Systems (BMS) are a particular type of automation systems aimed to control components such as lighting, heating, ventilation and air conditioning (HVAC) or safety systems. Differently from the Internet, where TCP/IP family of network protocols is universally employed as the common communication protocol among the systems, when it comes to building automation there are several different communication protocols available, meaning communication is either incompatible between them or at least difficult to implement. This situation makes the development of new BMSs difficult and complicates the development of new applications for the existing ones.

The use of a dynamic component model based on Java in order to implement a BMS provides certain advantages, mainly by the possibility of creating more modular, flexible and adaptable systems to the existing control technologies, the ease of implementing specific building applications or the availability of different hardware platforms due to the features of the Java language. In this respect, the use of the Open Service Gateway initiative (OSGi), a specification which describes a modular system and service platform for the Java language can be useful to implement these kind of control systems. In that respect, CeDInt (Centro de Domótica Integral-Research Centre for Smart Buildings and Energy Efficiency) is developing a platform focused on making possible the management of building systems and allowing the development and integration of new applications in the platform due to an application programming interface.

Moreover, one of the expected goals of this platform is to make the most of modularity, allowing dynamic integration of new components, whether they are new drivers for the communication protocols of a building or new applications aimed to increase the whole system functionality. However, this hot deployment of components in the platform (i.e. making changes to the running system) has yet to be fulfilled. It is for this reason that the main objective of this thesis is the development of a new functionality which allows the remote downloading and execution of components in the platform developed by CeDInt.

In order to meet this goal, two actions are proposed in this thesis, on the one hand the integration of a new OSGi container (Apache Karaf) which is expected to extend the platform functionality and on the other hand the design and implementation of a Java web application which allows to download, install and control new components in various of these platforms simultaneously.

Índice general

Índice de figuras

Siglas

API: Application Programming Interface.
CSS: Cascading Style Sheet.
IDE: Integrated Development Environment.
Java EE: Java Platform Enterprise Edition.
Java RMI: Java Remote Method Invocation.
JVM: Java Virtual Machine.
LDAP: Lightweight Directory Access Protocol.
SNMP: Simple Network Management Protocol.

1 Introducción

Este Trabajo Fin de Máster pretende contribuir a la extensión de un proyecto del Centro de Domótica Integral (CeDInt) de la UPM[1]. Este proyecto, denominado internamente como Building Automation Technologies Management Platform (BatMP) está enfocado a la gestión de los diferentes sistemas de automatización presentes en un edificio o entorno relacionado. Para ello, se pretende desarrollar un sistema con dos objetivos principales. Por un lado, trata de ofrecer una capa de abstracción para el control de diferentes tecnologías domóticas (como por ejemplo KNX[2], LonWorks[3], 6LoWPAN[4] o Constrained Application Protocol CoAP) a través de una API que oculte los aspectos técnicos específicos del acceso a cada tecnología.

Además, pretende ofrecer una interfaz para el desarrollo de aplicaciones que hagan uso de dicha capa de abstracción y de este modo puedan utilizar las diferentes tecnologías domóticas de una forma sencilla. Estas aplicaciones estarían destinadas al control de diferentes elementos domóticos del edificio o de su entorno, como por ejemplo la monitorización de parámetros del edificio o el control de elementos como luminarias o aparatos de climatización. En este sentido, la pasarela BatMP actúa como un middleware de servicios orientados a la gestión de edificios (BMS).

A continuación se describe brevemente la estructura de la pasarela BatMP (apartados 1.1 y 1.2) y a continuación las motivaciones principales del proyecto.

Nota: En lo sucesivo, se utilizarán indistintamente los términos "BatMP" y "Pasarela BatMP" y "Pasarela".

1.1 Pasarela CeDInt

BatMP [10] es un sistema modular basado en Java y en OSGi[5] que permite las siguientes funcionalidades:

- Comunicación a bajo nivel con diferentes protocolos domóticos y de automatización de edificio.
- Modelado de las diferentes estancias del edificio, así como de los diferentes dispositivos y sistemas presentes en el mismo.

[1] http://www.cedint.upm.es/es
[2] http://www.knx.org/knx-en/index.php
[3] http://www.echelon.com/technology/lonworks/
[4] http://6lowpan.net/
[5] http://www.osgi.org/Technology/HomePage

- Gestión de aplicaciones que hacen uso de los sistemas y dispositivos. Las aplicaciones hacen uso de los sistemas del edificio a través de los parámetros del mismo (como por ejemplo temperatura, iluminación, nivel de humedad, etc).

BatMP consta de tres capas que gestionan el modelado del edificio y sus elementos, el control de las diferentes tecnologías y la gestión de las aplicaciones. Estas capas son las siguientes:

- Technology Manager

El objetivo principal de esta capa es facilitar la integración de las diferentes tecnologías y protocolos de comunicaciones presentes en el edificio. Este objetivo se logra mediante la definición de los llamados dispositivos físicos y dispositivos lógicos. Los primeros se encargan de agrupar las características específicas propias de cada tecnología, como por ejemplo su formato de direccionamiento. Los segundos se utilizan para agrupar diferentes elementos del edificio en base a su localización. Así, un dispositivo lógico puede agrupar por ejemplo todas las luminarias cercanas a una determinada ventana, con el objetivo de poderlas controlar de forma conjunta e independiente de los demás elementos.

Así, el Technology Manager proporciona una capa de abstracción sobre los diferentes elementos del edificio. Este objetivo se consigue a través de la definición de los llamados parámetros. Cada parámetro representa una determinada magnitud física que es controlable por la pasarela (ya que existe un dispositivo físico capaz de interactuar directa o indirectamente con ella), así como el conjunto de acciones que se pueden realizar sobre ella. De este modo, se consigue que las aplicaciones que hacen uso de los dispositivos no tengan que conocer los detalles tecnológicos de cada dispositivo.

Por ejemplo, para una determinada habitación en la que exista un sensor de temperatura u otro dispositivo directamente accesible por la pasarela que mida temperatura, se crearía el parámetro "Temperatura" que almacenaría el valor de la misma en grados centígrados. Las aplicaciones que deseen conocer el valor de dicha temperatura, acceden únicamente a dicho parámetro, sin tener que comunicarse físicamente con el sensor o dispositivo que la mide. El Technology Manager se encarga de actualizar el valor de cada uno de los sensores a través del correspondiente driver de la tecnología que éstos utilizan, con el objetivo de que las aplicaciones no tenga que conocer dicha tecnología ni comunicarse directamente con cada uno de los sensores.

Del mismo modo, los parámetros incluyen el conjunto de acciones de control que se puede realizar sobre ellos. Por ejemplo, en el caso del mismo parámetro Temperatura, si en la habitación se dispone de un sistema de climatización, el conjunto de acciones del parámetro incluirán el poder fijar un determinado valor de la misma, por lo que las

acciones de control dependerán siempre de los dispositivos de los que se disponga físicamente.

- Model Manager

El conjunto de información que un edificio puede almacenar incluye la siguiente:

- Características estructurales del edificio, tales como el número de plantas, el número y la distribución de las habitaciones, la orientación geográfica, etc.
- Información sobre los sistemas eléctricos del edificio tales como las redes de sensores, dispositivos de seguridad, sistemas de iluminación o sistemas de climatización.
- Magnitudes físicas que pueden ser medidas, modificadas, o ambas, como por ejemplo temperatura, nivel de iluminación, consumo eléctrico, etc.

El Model Manager de la pasarela se encarga de almacenar esta información utilizando dos modelos de datos diferentes, uno basado en semántica que almacena el modelo del edificio, y otro basado en el modelo relacional para almacenar información de configuración de dispositivos y tecnologías, así como los datos manejados por el sistema.

Con el objetivo de almacenar la información del modelo del edificio de una forma flexible y escalable, se utiliza un modelo del mismo basado en ontologías. El uso de ontologías proporciona algunas ventajas frente a las bases de datos relacionales en determinados dominios, como por ejemplo facilitar la interoperabilidad entre diferentes aplicaciones y ofrecer mecanismos de inferencia sobre los datos del modelo.

Si bien las bases de datos relacionales ofrecen una forma de almacenar y acceder a la información de una forma estructurada y eficiente, presentan algunas desventajas, como por ejemplo la pérdida de significado semántico de las relaciones cuando éstas se representan en forma de tablas, y la dificultad para ampliar o modificar un modelo de datos cuando éste ya se encuentra implementado. Las bases de datos basadas en ontologías por el contrario facilitan la publicación de los datos así como su ampliación y el razonamiento semántico sobre los mismos.

- Application Manager

El propósito de esta capa es proporcionar a las aplicaciones del sistema una interfaz simple y unificada para acceder a los datos de la plataforma, permitiéndoles acceder tanto a los datos abstractos del modelo como a los dispositivos físicos sin tener que conocer los datos específicos de cada tecnología. Con este objetivo, el Application Manager proporciona una interfaz común para el acceso a la pasarela a través de una arquitectura orientada a servicios (Service Oriented Architecture, SOA). En este contexto, una aplicación es cualquier programa independiente de la arquitectura que

interactúa directamente con los parámetros a través de los dispositivos (gestionados de forma transparente por la pasarela). El modelo SOA se basa en un intercambio de mensajes entre las entidades involucradas y permite un acoplamiento débil entre las aplicaciones y la pasarela. El objetivo principal de este diseño es lograr una integración más sencilla entre las aplicaciones, dando libertad al diseñador de las aplicaciones, ya que el único requisito de comunicación es el formato de los mensajes intercambiados.

La principal desventaja de esta arquitectura es su menor eficiencia comparada con otras implementaciones más fuertemente acopladas, debido entre otros motivos a que es un diseño pensado para entornos distribuidos y el intercambio de los mensajes introduce un *overhead* (coste extra) en las transmisiones. La Figura 1 representa la arquitectura general de la pasarela CeDInt.

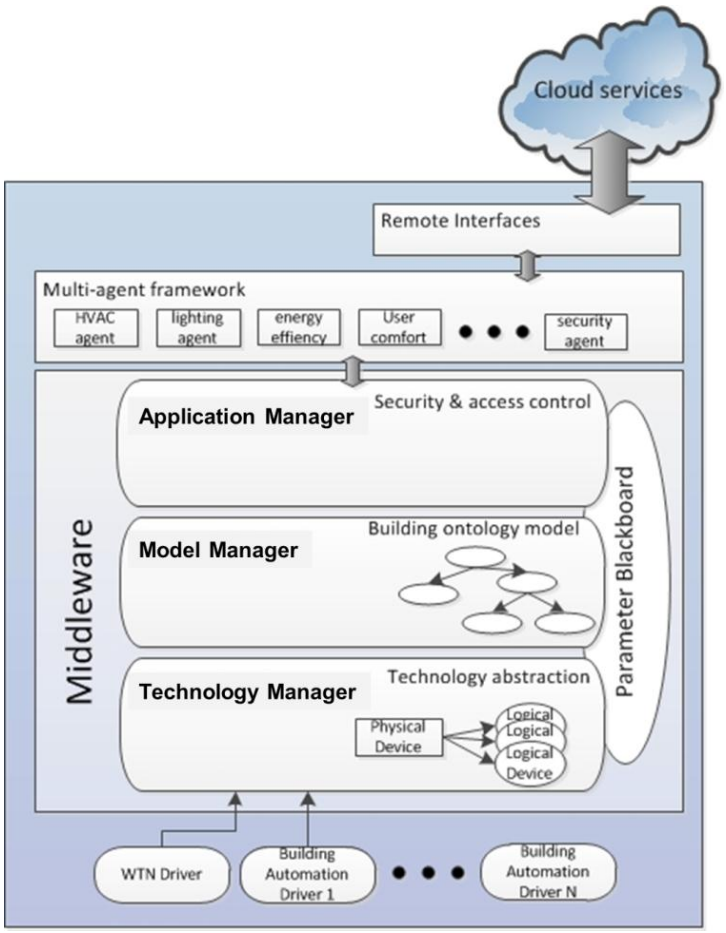

Figura 1. Estructura de BatMP.

BatMP se ha implementado utilizando la tecnología OSGi, una plataforma basada en Java que implementa un sistema modular basado en componentes software (denominados bundles) y que permite una gestión dinámica de los mismos, controlando el ciclo de vida de cada componente y diferentes aspectos de seguridad.

Finalmente, otro de los objetivos del diseño de BatMP es hacerlo lo suficientemente ligero como para que se pueda ejecutar en dispositivos como Raspberry Pi[6] o BeagleBone[7]. Estos dispositivos, que generalmente utilizan microprocesadores ARM[8], se caracterizan por tener una baja capacidad de proceso y de memoria, así como un coste reducido. En particular, se han realizado pruebas de control de una veintena de dispositivos sobre un dispositivo BeagleBone como el de la Figura 2, que posee un micro ARM a una frecuencia de 1GHz, 512MB de memoria RAM y almacenamiento externo mediante una tarjeta SD de 2GB.

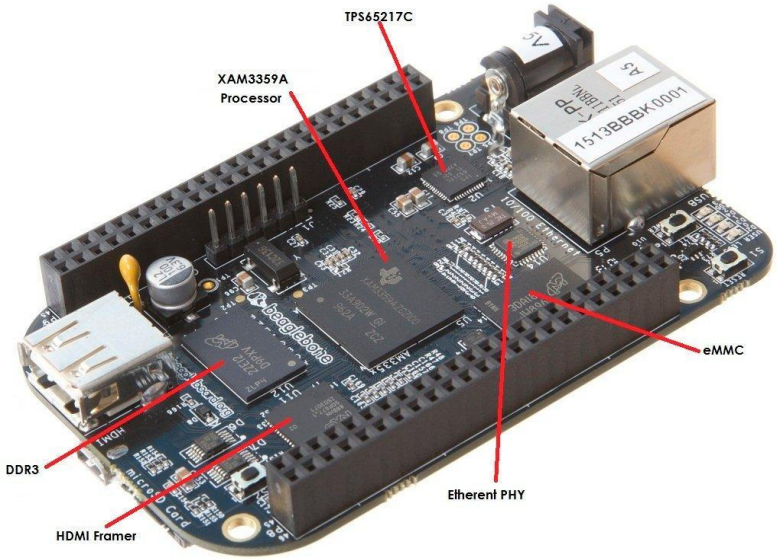

Figura 2. BeagleBone.

[Fuente: http://www.cnx-software.com]

1.2 Servidor central de control

Hasta ahora, se ha descrito la pasarela BatMP como un sistema aislado capaz de gestionar un conjunto de sistemas de un edificio mediante diferentes drivers domóticos, así como mediante la implementación de aplicaciones específicas que hacen uso de las tecnologías del edificio. De esta forma, el control de cada edificio sería independiente de cualquier otro. Uno de los objetivos a largo plazo de la pasarela BatMP es poder desarrollar un servidor central que permita realizar determinadas acciones de gestión sobre un conjunto de pasarelas distribuidas. De este modo, se ha planteado la posibilidad de unificar la gestión de los módulos OSGi de cada pasarela a través de un servidor central. En este sentido, una de las funcionalidades que se quiere incorporar a dicho servidor central es la gestión dinámica de los componentes (bundles) de cada una de las pasarelas locales, con el objetivo de poder controlar la instalación y estado de ejecución de diferentes módulos en las mismas.

[6] http://www.raspberrypi.org/
[7] http://beagleboard.org/Products/BeagleBone
[8] http://www.arm.com/

Esta funcionalidad resulta de utilidad por ejemplo cuando está disponible una nueva versión de un determinado bundle, o cuando existe un nuevo módulo que extiende las funcionalidades del sistema. Al poder controlar de forma centralizada los bundles de cada una de las pasarelas locales, por un lado se busca ampliar el concepto de pasarela local a otro de "combinación de pasarelas" con el que poder gestionar entornos más grandes que un edificio y por otro se persigue ahorrar tiempo en la realización de acciones de configuración tales como la instalación de bundles desde repositorios remotos.

Finalmente, la implementación actual de la pasarela BatMP está realizada utilizando únicamente el entorno OSGi Equinox, sin la ayuda de ningún contenedor, por lo que la ejecución de la pasarela y sus aplicaciones en los servidores de desarrollo es controlada mediante una serie de scripts y archivos de configuración, lo cual hace que su utilización resulte complicada. Además, no se dispone de una interfaz gráfica amigable para interactuar con el entorno OSGi de Equinox, ya que sólo es posible utilizar una consola de comandos con funcionalidades limitadas.

Esta consola es accesible mediante el protocolo telnet, por lo que en el caso de querer establecer una conexión desde el exterior de la red donde se encuentra BatMP, habría que resolver posibles problemas de conectividad si existen firewalls entre los hosts. En este sentido, se hace necesario disponer de una interfaz gráfica web que permita gestionar aspectos de BatMP, y que al mismo tiempo, resulte ligera en términos de memoria, para permitir que todo el software se ejecute en dispositivos de reducidas prestaciones.

1.3 Objetivos

El objetivo que se pretende con el presente Proyecto Fin de Máster es el desarrollo de un módulo de control de los bundles instalados en el sistema que permita la realización de otras acciones de control sobre el entorno OSGi de BatMP, limitando en lo posible las necesidades de memoria de dicho módulo de control. Además, otro de los objetivos consiste en incorporar a la pasarela BatMP un mecanismo que saque un mayor partido a la modularidad de OSGi. Como se ha comentado en el apartado anterior, actualmente, el control de los bundles instalados en el sistema se realiza de forma estática, utilizando un script que es leído cada vez que se inicia una ejecución, por lo que se hace necesario la utilización de un contenedor OSGi que permita la inclusión de nuevas funcionalidades.

Como se ha comentado, la principal funcionalidad del módulo que se pretende desarrollar será la instalación de nuevos bundles en el sistema, resolviendo sus dependencias si fuera necesario, y que permita activar o desactivar dichos bundles de manera remota, es decir, desde un servidor central. Además, con el objetivo de evitar

problemas de conectividad, el módulo a desarrollar debe poseer una interfaz web que evite problemas con firewalls entre el servidor central y la pasarela BatMP que se esté controlando. La Figura 3 ilustra este concepto.

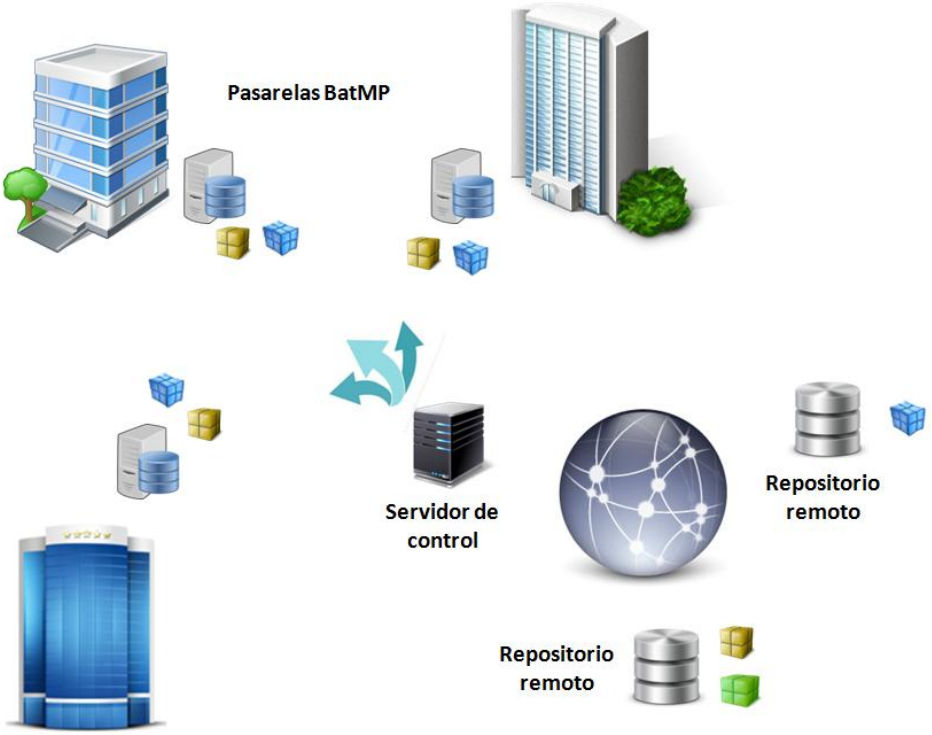

Figura 3. Servidor de control de pasarelas BatMP.

En los siguientes apartados de esta memoria se explican, en primer lugar, las tecnologías utilizadas para el desarrollo, haciendo especial hincapié en OSGi, dado que es el núcleo de la pasarela BatMP. A continuación se describen los contenedores OSGi que se han evaluado durante el trabajo, seguido de una explicación detallada del que finalmente se ha utilizado para el desarrollo, Apache Karaf. En los apartados sucesivos se describe el diseño y la implementación de la solución desarrollada, así como las pruebas que se han realizado, para finalizar con las conclusiones y trabajos futuros.

2 Estado del arte de las tecnologías utilizadas

En este apartado se describen las tecnologías que se han utilizado para el desarrollo de este trabajo. En primer lugar la especificación OSGi, así como sus principales implementaciones, seguido del gestor de proyectos Maven y las tecnologías de Java JMX y RMI. Finalmente, se describe el concepto de OSGi Bundle Repository (OBR) y sus principales implementaciones.

2.1 OSGi

La alianza OSGi nace en marzo de 1999 con el objetivo de generar una especificación abierta y colaborativa de java donde desarrollar plataformas compatibles que puedan proporcionar múltiples servicios. Algunos de sus miembros son Motorola, Nokia, Mitsubishi Electric Corporation, etc. En este sentido, se puede decir que OSGi es una especificación generada por una alianza entre empresas con el objetivo de modularizar el lenguaje Java.

2.1.1 Resumen de OSGi

Para conseguir entender la estructura y funcionamiento de OSGi es preciso tener en cuenta algunos conceptos de la máquina virtual de Java (Java Virtual Machine JVM) y de su funcionamiento. La JVM tiene probablemente mucha relación con el gran éxito que Java ha ido adquiriendo hasta nuestros días, puesto que permite hacer funcionar el mismo programa en diferentes máquinas con arquitecturas diferentes sin necesidad de realizar modificaciones en el mismo. Una vez se inicia una JVM, se crea o bien un sólo "ClassLoader" o bien un árbol jerárquico de éstos, en donde introducen todos los recursos y librerías de los que va a disponer la JVM para la ejecución. Si por ejemplo, dos de los diferentes archivos .JAR de los que consta la aplicación dependen de versiones distintas de la misma librería, la máquina virtual forzará la utilización de una sola de ellas, lo cual puede dar lugar a errores del tipo NoSuchMethodException cuando una clase de una .JAR interactúa con una clase incompatible de otra. Por lo tanto, una vez iniciada la máquina virtual, si posteriormente se quieren añadir nuevos módulos que hagan uso de librerías o recursos diferentes (y que por lo tanto, no han sido cargados), ésto no será posible, limitando la modularidad de la aplicación.

OSGi se basa en la utilización de módulos o *bundles* (se utilizará este término en lo sucesivo) cuya ejecución se controla de forma dinámica. Al iniciar un entorno OSGi, sus archivos .JAR son colocados en la JVM, creándose una instancia de la clase principal de OSGi, desde la cual se cargarán los diferentes archivos .JAR de los bundles. La principal ventaja de OSGi es que controla la creación de los objetos "ClassLoaders" de cada bundle, asignándole uno de ellos a cada bundle que se introduce en el sistema. Posteriormente mediante "puentes" entre ClassLoaders

consigue controlar el ciclo de vida de un nuevo recurso no inicializado previamente en la JVM. De este modo, una vez que se crea un bundle, se comprueba si sus *import* pueden satisfacerse en ese momento, es decir, si existe otro bundle en el sistema que exporta las librerías que necesita. Si es así, estos *imports* son "marcados" para que el bundle los resuelva del ClassLoader del bundle que los exporta. Existen otras formas para modularizar java pero OSGi es sin lugar a dudas la más utilizada, madura y respaldada que existe.

El framework de OSGi es el núcleo de las aplicaciones basadas en esta tecnología y nos proporciona una forma de desplegar, instalar, extender o eliminar las aplicaciones que están empaquetadas en bundles.

2.1.2 Estructura de OSGi

La funcionalidad de OSGi se divide en varias capas, ilustradas en la Figura 4.

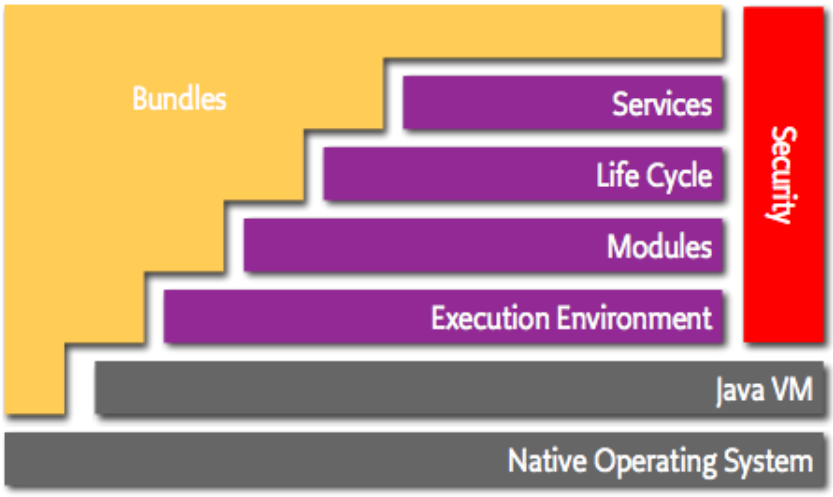

Figura 4. Estructura de OSGi.

[Fuente: http://www.osgi.org/Technology/WhatIsOSGi]

Security Layer

La capa de seguridad es una capa opcional de OSGi que está basada en la arquitectura de seguridad de Java 2 que es la seguridad estándar de Java y se base en usuarios, permisos, roles y grupos. Esta arquitectura no queda cerrada en la especificación por lo que es posible ampliarla o modificarla.

Module Layer

La capa de modularidad define el concepto OSGi de módulo, el bundle, que es básicamente un archivo .JAR con metadatos asociados. Estos metadatos se representan mediante el archivo MANIFEST.MF. Este archivo, compuesto por pares clave-valor, describe las propiedades principales de un bundle, tales como el nombre, la versión y los paquetes que importa y exporta. Las diferentes propiedades de este archivo, así como su descripción exhaustiva, pueden encontrarse en [3] y [4].

Como se comentará en la sección 3.1, a la hora de generar un bundle, con sus propiedades descritas en el archivo MANIFEST.MF, existen diferentes herramientas. Una de las más utilizadas es un plugin del software de construcción de proyectos Maven, del cual se habla en dicha sección.

Life Cycle Layer

La capa del ciclo de vida define cómo los bundles son dinámicamente instalados y gestionados en un entorno OSGi. Los estados en los que puede estar un bundle son los siguientes.

- **INSTALLED:** El bundle ha sido instalado correctamente.
- **RESOLVED:** Todas las clases java que el bundle necesita están disponibles. Este estado indica que el bundle está listo para ser arrancado o que el bundle ha sido detenido.
- **STARTING:** El bundle está arrancando. En caso de que hayamos establecido la propiedad Bundle-ActivationPolicy a lazy, se quedará en este estado hasta que se necesario su arranque.
- **ACTIVE**: El bundle ha sido correctamente activado y está en funcionamiento.
- **STOPPING**: El bundle ha sido detenido.
- **UNINSTALLED**: El bundle ha sido desinstalado. No se puede mover a otro estado.

La Figura 5 ilustra un diagrama con los estados en los que puede estar un bundle y las relaciones entre ellos.

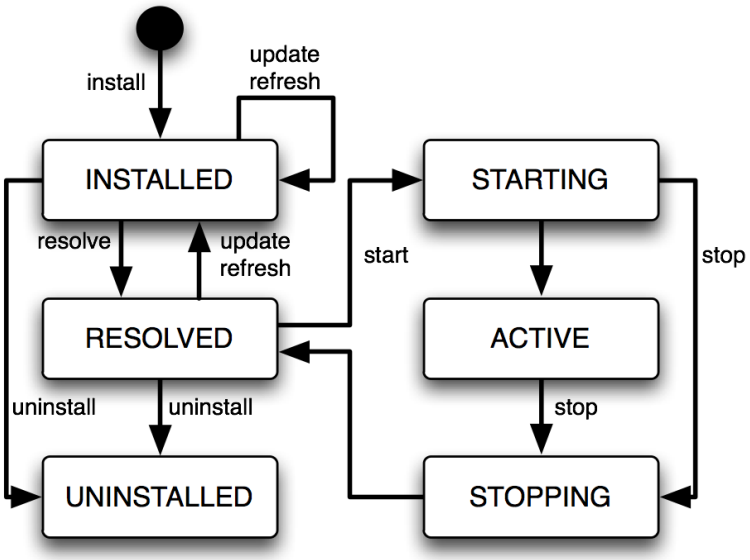

Figura 5. Ciclo de vida de un bundle.

[Fuente: http://zenit.senecac.on.ca/wiki/]

Service Layer

Proporciona una forma de crear o modificar servicios internos OSGi.

OSGI, además de ser un sistema modular, es también orientado a servicios. En este sentido, un servicio es un objeto java que es registrado bajo una o varias interfaces. Los bundles pueden registrar servicios, buscarlos o recibir notificaciones cuando el estado de un servicio ha sufrido alguna variación. Para comprender esta capa, se deben definir algunos conceptos relacionados.

- **Service**: Entendemos por servicio, un objeto registrado en la plataforma bajo una más interfaces. Dicho objeto registrado puede ser usado por otros bundles.
- **Service Registry**: Gestiona el registro de servicios.
- **Service Reference**: Referencia a un servicio registrado. Con este objeto podemos obtener las propiedades del servicio, no el propio servicio.
- **Service Listener**: Listener para escuchar los eventos de los servicios.
- **Service Event**: Evento producido al registrar, modificar o dar de baja un servicio.
- **Filter**: Objeto para filtrar servicios.

Por lo tanto, con OSGI se puede registrar una clase Java como un servicio, a través del contexto del bundle. De esta forma, con el Bundle Context es posible buscar servicios registrados en la plataforma, así como escuchar y filtrar los eventos producidos en los cambios de estado de un servicio.

Actual Services

Finalmente, en esta capa se encontrarían los propios bundles que componen las aplicaciones.

Los servicios, formados por tres tipos de bundles, los que definen interfaces de servicios, los que implementan y registran esos servicios y por último los bundles que hacen uso de los mismos.

2.1.3 Implementaciones de OSGi

A continuación se exponen las dos principales implementaciones de OSGi, Eclipse Equinox y Apache Felix.

Eclipse Equinox

Equinox[9] es una implementación de OSGi R4.x desarrollada por la Eclipse Foundation[10] y que entre otras cosas sirve de entorno de ejecución de los diferentes componentes del popular IDE Eclipse[11]. Es la implementación de OSGi utilizada en la pasarela BatMP.

El entorno de ejecución Equinox se puede controlar mediante una consola de comandos a la que se accede mediante el protocolo telnet. Utilizando esta consola se pueden realizar acciones sencillas como arrancar o detener un bundle, obtener información de éstos, o encontrar las dependencias no satisfechas algún componente.

Apache Félix

Apache Félix es una implementación de OSGi desarrollada por la comunidad de desarrolladores de código abierto Apache. El proyecto se inició como una donación del código de otro proyecto (Oscar project), y desde el año 2007 es un proyecto de primer nivel. Entre sus características se encuentran:

- Implementación del propio entorno de OSGi, con los archivos principales del núcleo de ejecución.

- Implementación de la especificación de servicios HTTP, que permite la comunicación de los bundles utilizando dicho protocolo, así como la utilización en el entorno OSGi de servicios que utilicen por ejemplo el paradigma REST o

[9] http://www.eclipse.org/equinox/
[10] http://www.eclipse.org/
[11] http://www.eclipse.org/ide/

la generación de contenido en XML o HTML. Esta implementación se proporciona mediante el servidor HTTP Eclipse Jetty. No obstante, Apache Felix proporciona una funcionalidad muy útil cuando se desea desplegar un servicio web en un contenedor de servlets distinto a Jetty. Esta funcionalidad se denomina servlet bridge, y permite a un contenedor web estándar (como por ejemplo Apache Tomcat) delegar las peticiones HTTP a otra aplicación (en este caso al entorno OSGi).

- Servicio de logging, que en el caso de OSGi se divide en dos componentes principales, el Log Service interface, utilizado por los propios bundles cuando quieren registrar algún mensaje de log (es decir, la interfaz que se utiliza en el propio código de las clases Java del bundle) y el Log Reader Service, que es el encargado de capturar los mensajes y enviarlos a su destino. Esta separación de los componentes del proceso de logging es habitual, y proporciona una abstracción entre la generación de los mensajes de log y su escritura en otros componentes (ya sea una base de datos o un fichero de texto).

- Consola de comandos gogo. A semejanza de Equinox, Felix incorpora una consola sencilla de comandos llamada Gogo[12], basada en el estándar OSGi RFC 147[13] que describe un conjunto estandarizado de comandos para entornos basados en OSGi. La consola Gogo utiliza nombres cualificados en la forma de <scope>:<name> con el objetivo de diferenciar comandos de diferentes dominios.

- Apache Felix proporciona un modelo de componentes llamado injected Plain Old Java Objects (iPOJO)[14]. Esta implementación se encarga de registrar los diferentes bundles del sistema y de resolver las dependencias entre ellos, sin necesidad de modificar las clases Java que se utilizan.

- Maven bundle plugin, que proporciona un conjunto de instrucciones para generar directamente bundles.

2.2 Maven

Apache Maven[3] es una herramienta de gestión de proyectos que permite definir un modelo de objetos del proyecto (del inglés POM, Project Object Management) así como su sistema de dependencias y la lógica de ejecución de las diferentes fases del ciclo de vida del proyecto.

[12] http://felix.apache.org/site/apache-felix-gogo.html
[13] http://wiki.osgi.org/wiki/RFC_147
[14] http://felix.apache.org/documentation/subprojects/apache-felix-ipojo.html

Mediante Maven, se puede gestionar la construcción de un proyecto Java programando las acciones a realizar mediante la utilización de plugins configurados por el desarrollador. En esencia, Maven realiza la ejecución de estos plugins para construir un proyecto completo, desde la compilación de las clases a bytecode hasta la ejecución de tests. Estas acciones de configuración se realizan a través del archivo POM.xml, el cual es interpretado por Maven para construir el proyecto. A grandes rasgos, la estructura de un archivo POM es la siguiente.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      http://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>

<!-- Información básica -->
<groupId>...</groupId>
<artifactId>...</artifactId>
<version>...</version>
<packaging>...</packaging>
<dependencies>...</dependencies>
<parent>...</parent>
<dependencyManagement>...</dependencyManagement>
<modules>...</modules>
<properties>...</properties>

<!-- Contrucción de proyecto y plugins -->
<build>
      <plugin>
      <artifactId>...</artifactId>
      <executions>
      <execution>
      <id>attach-javadocs</id>
      <goals>
      <goal>jar</goal>
      </goals>
      </execution>
      </executions>
      </plugin>
      <plugin>...<plugin>
</build>
</project>
```

Una de las características más interesantes de Maven es que da por supuestos determinadas propiedades de configuración del proyecto. Por ejemplo, se asume que el proyecto va a tener una determinada estructura de directorios en donde el código fuente se va a alojar en la carpeta ${proyecto}/src/main/java, y donde el fichero

.JARresultante se va a alojar en la carpeta ${proyecto}/target. Esto permite que, si se siguen las convenciones, la construcción de un proyecto resulte mucho más homogénea y fácil de implementar.

Finalmente, el sistema de dependencias de Maven hace que la gestión de las mismas se realice de una forma estandarizada. Cuando se especifica que un proyecto tiene una determinada dependencia, en primer lugar Maven tratará de resolverla en el repositorio local (el cual se especifica en el archivo de configuración settings.xml de la instalación) y en caso de no poderla resolver, tratará de localizarla en el repositorio central de Maven[15] para su descarga e incorporación al proyecto. La Figura 6 representa las diferentes fases y plugins básicos de Maven.

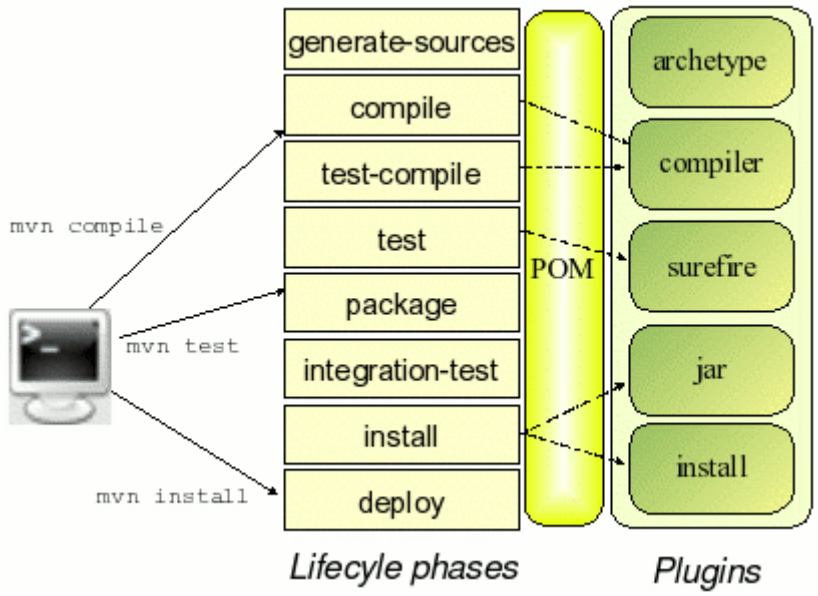

Figura 6. Estructura de las fases y plugins de Maven.

[Fuente: http://www.javaworld.com/article/2072203/]

A la hora de generar proyectos basados en OSGi, existe un plugin específico de Maven que facilita la creación de bundles, el Bundle Plugin for Maven[16]. Para generar un nuevo bundle es necesario incluir la configuración siguiente en la sección de plugins del archivo POM:

```
...
<plugin>
<groupId>org.apache.felix</groupId>
<artifactId>maven-bundle-plugin</artifactId>
<version>2.4.0</version>
<extensions>true</extensions>
```

[15] http://search.maven.org/#browse
[16] http://felix.apache.org/site/apache-felix-maven-bundle-plugin-bnd.html

```
<configuration>
<instructions>
<Export-Package>es.upm.cedint.api.*</Export-Package>
<Private-Package>es.upm.cedint.util.*</Private-Package>
<Bundle-Activator>es.upm.cedint.core.Activator</Bundle-Activator>
</instructions>
</configuration>
</plugin>
...
```

Los componentes marcados en negrita son los paquetes que exportará el bundle al resto del contexto de OSGi (en este caso los subpaquetes de api), mientras que los marcados como private no serán visibles desde el resto del contexto. Por último, es necesario especificar la implementación de la interfaz Activator[17]. Esta clase específica de OSGi especifica las funciones start y stop que se ejecutarán cada vez que el bundle sea arrancado por OSGi.

2.3 Java Servlets

De forma simplificada, un servlet es una clase o conjunto de clases Java que se ejecutan en el contexto de un servidor web, implementando la especificación Java Servlet API[18]. Funcionalmente, un servlet es un objeto que recibe peticiones (principalmente mediante el protocolo HTTP, aunque no necesariamente siempre), las procesa y genera la respuesta correspondiente.

2.4 Java Management Extension

Java Management Extensions (JMX) es un conjunto de especificaciones para la gestión de aplicaciones y elementos de red en un entorno Java EE. El concepto general de JMX reside en la creación de objetos Java (con una interfaz que define determinados atributos y métodos) que encapsulan otras aplicaciones, componentes software o elementos hardware y que exponen sus atributos y algunas operaciones de gestión en un entorno distribuido. Estos objetos Java se denominan MBeans (de Managed Beans) y se construyen como clases Java que cumplen determinados estándares dictados por la especificación JMX[19], y cuya interfaz está constituida por los atributos y operaciones que se exponen para la gestión de los mismos. A estos MBeans se accede mediante diferentes protocolos (como por ejemplo RMI o HTTP) utilizando conectores específicos que implementan dichos protocolos. En definitiva, JMX permite "envolver"

[17] http://www.osgi.org/javadoc/r4v43/core/org/osgi/framework/BundleActivator.html
[18] http://docs.oracle.com/javaee/6/api/javax/servlet/Servlet.html
[19] http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html

cualquier componente software (independientemente del lenguaje en el que esté implementado) y exponer sus atributos utilizando objetos Java.

Un servidor de MBeans es una clase Java que gestiona un grupo de MBeans, y representa el núcleo de la tecnología JMX, actuando como intermediario entre los MBeans y las aplicaciones que se comunican con ellos. Finalmente, un agente JMX (JMX agent) es un contenedor de un servidor de MBeans.

JMX está estructurado en tres capas:

- Capa de distribución (Distributed Layer)
  Esta capa es la más externa de una aplicación basada en la arquitectura JMX y es la responsable de que los agentes JMX sean visibles desde el exterior. A través de esta capa se producen las interacciones con el servidor de MBeans utilizando adaptadores específicos, bien a través de protocolos como HTTP o SNMP, o bien a través de otras tecnologías como Java RMI.

- Capa de agente (Agent Layer)
  El principal componente de esta capa es el agente JMX (JMX agent), que puede ejecutarse bien en la JVM que alberga los recursos, o bien localizada de forma remota. No requiere conocimiento de los recursos que está exponiendo, simplemente actúa gestionando y permitiendo la manipulación de los MBeans a través de un conjunto de protocolos utilizando los adaptadores correspondientes.

- Capa de instrumentación (Instrumentation Layer)
  Finalmente, la capa de instrumentación es la más cercana a los recursos gestionados a través de JMX. En esta capa se sitúan los MBeans que exponen parte de la configuración o funcionalidad de algún recurso, independientemente de su tecnología. Por ejemplo, acciones de control sobre una base de datos MySQL utilizando un driver JDBC o la gestión de un servidor web utilizando REST.

La Figura 7 representa la estructura de JMX.

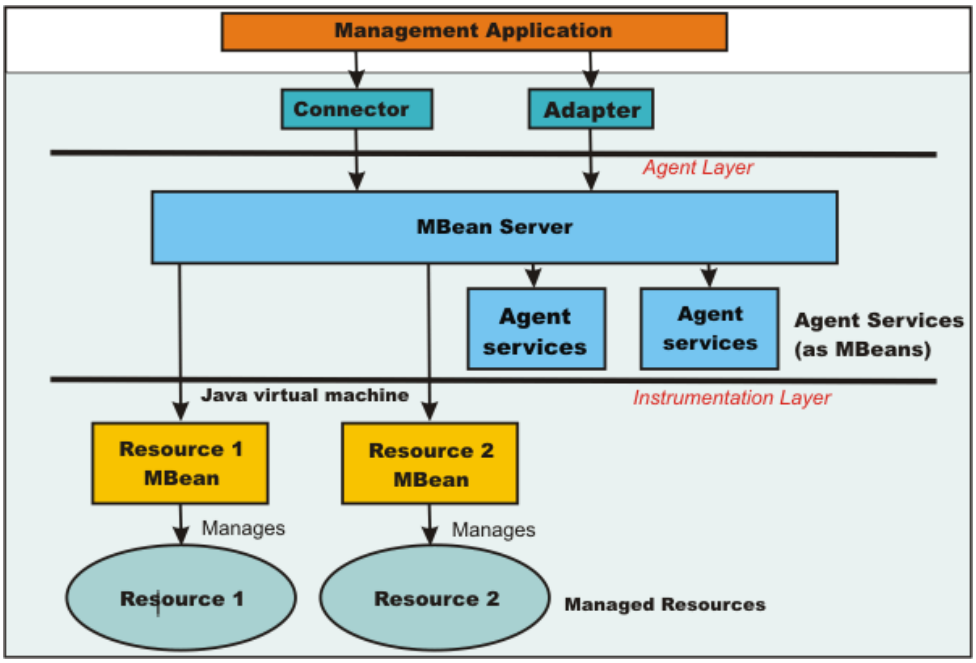

Figura 7. Estructura de JMX.

[Fuente: http://pic.dhe.ibm.com/infocenter/wasinfo/]

2.5 Java Remote Method Invocation (RMI)

La tecnología Java RMI es la implementación en Java del concepto de Remote Procedure Call (RPC). Se utiliza para poder invocar algún método localizado en la interfaz de un objeto remoto (externo a la máquina), y donde se utiliza la misma sintaxis que se utilizaría si el objeto fuera local. De este modo, RMI proporciona un mecanismo para la distribución de objetos Java como servicios, donde una petición a un servicio remoto es similar (desde el punto de vista del llamante) a una realizada localmente.

RMI emplea un mecanismo para la comunicación entre objetos remotos basado en *stubs* y *skeletons*. Un stub es un objeto local a la máquina en donde se va a invocar un método que actúa como representante local (o proxy) del objeto remoto. Este stub es el encargado de transportar la llamada al objeto remoto, serializando (marshalling) y transmitiendo la llamada por la red. Este proceso es transparente al objeto que realiza la llamada. En el servidor (donde está el objeto que recibe la llamada) se encuentra el skeleton, que es el encargado de recibir la llamada y enviarla al objeto real, así como de recibir el resultado de la ejecución, serializarlo de nuevo y enviarlo de vuelta al stub del cliente, igualmente de forma transparente. Finalmente, para que el cliente pueda localizar el objeto remoto, éste ha de estar localizado en el registro RMI (RMI Registry) del servidor. La Figura 8 ilustra la estructura general de la tecnología Java RMI.

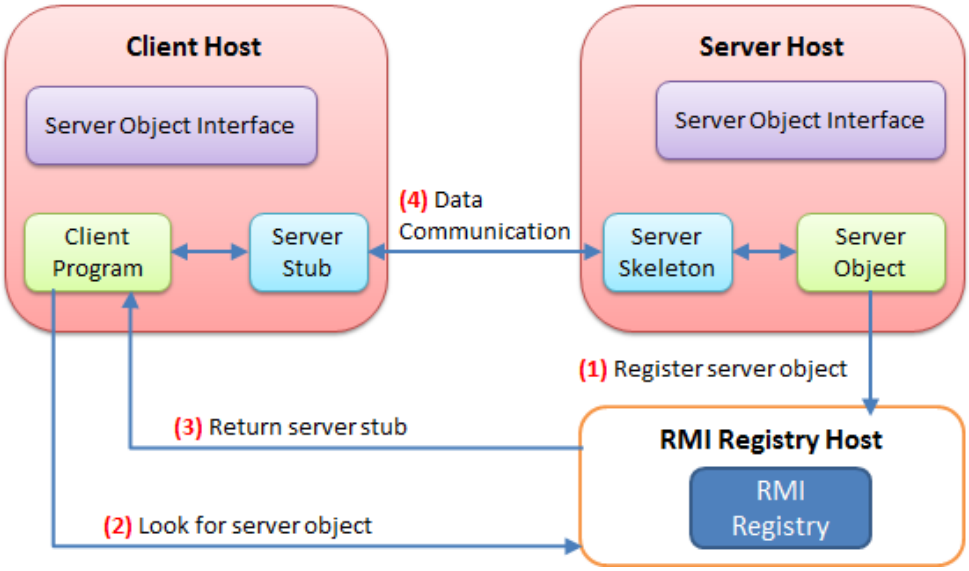

Figura 8. Diagrama de Java RMI.

[Fuente: http://lycog.com/distributed-systems/java-rmi-overview/]

2.6 OSGi Bundle Repository

OSGi Bundle Repository es desde 2012 la última especificación estándar de OSGi[20] para la distribución de bundles. Durante años había formado parte de la especificación de Oscar Bundle Repository, que estaba asociado con el Oscar OSGi framework, el cual finalmente se convirtió en el proyecto Apache Félix. Los objetivos principales de la especificación OBR son los siguientes:

- **Descubrimiento**. Es decir, permitir recopilar información sobre qué bundles están disponibles para ser descargados y desplegados en el sistema.

- **Gestión de dependencias.** Que se refiere a la resolución y obtención automática de las dependencias transitivas de los bundles.

Una implementación de un OSGi Bundle Repository (de ahora en adelante, denominado simplemente OBR) es básicamente una descripción genérica de un conjunto de recursos y sus dependencias. En este sentido, aunque un recurso se tratará fundamentalmente de un bundle, no es necesario que siempre sea así, ya que un recurso también se puede referir a otros elementos tales como archivos de configuración.

Un OBR funciona leyendo información de un fichero de metadatos en XML (generalmente denominado repository.xml) y utilizando esta información para construir la información sobre dependencias para desplegar y ejecutar los bundles en un determinado framework OSGi.

[20] http://www.infoq.com/news/2012/06/osgi-r5

La estructura de un archivo de descripción de repositorio es la siguiente:

```
<repository lastmodified=... name=...>
      <resource id=... symbolicname=... presentationname=... uri=...>
            <size>...</size>
(1)         <capability (2) name=’bundle’>
                  <p n='symbolicname' v='...'/>
                  <p n='presentationname' v='...'/>
                  <p n='version' t='version' v='...'/>
                  <p n='manifestversion' v='...'/>
            </capability>
            <capability (3) name=’package’>
                  <p n='symbolicname' v='...'/>
                  <p n='presentationname' v='...'/>
                  <p n='version' t='version' v='...'/>
                  <p n='manifestversion' v='...'/>
            </capability>
(1)         <capability>...</capability>
(4)         <require name='package'
filter='(&amp;(package=org.osgi.framework)(version&gt;=1.0.0))'
extend='false' multiple='false' optional='false'>Import package
org.osgi.framework;version=1.0.0</require>
(4)         <require>...</require>
(4)         <require>...</require>
      </resource>
      <resource>
            ...
      </resource>
      <resource>
            ...
      </resource>
</repository>
```

Los dos elementos principales en la descripción de un recurso son los *requirements* y los *capabilities*. Una capability (1) representa los diferentes elementos que el recurso proporciona al entorno de OSGi en donde va a desplegarse (con el objetivo de que otros elementos de OSGi puedan hacer uso de él). El tipo de elemento del que se trata se identifica con la etiqueta *name*, y describe el tipo de objeto que se está exportando a OSGi. Por ejemplo, en (2) se van a describir las propiedades del bundle exportado y en (3) se describen las diferentes propiedades de un paquete de dicho bundle. Dentro de una capability están contenidos los elementos identificados con la etiqueta <p>, que describen el nombre de la propiedad (n), la versión (v) y el tipo (t).

Un requirement (4) representa los paquetes o servicios que el recurso necesita (por ejemplo, los paquetes que debe importar para poder funcionar o los servicios que OSGi le debe proporcionar). La descripción del requerimiento se realiza a través de un filtro

utilizando el formato de LDAP. En el ejemplo de arriba, el primer requirement especifica que el recurso necesita el paquete org.osgi.framework, (uno de los paquetes básicos del entorno OSGi).

Una de las características interesantes de un OBR es que éste puede referenciar diferentes OBR, creando una federación entre ellos. Además, existen diferentes repositorios OSGi libremente accesibles como por ejemplo Apache Félix [6] o Apache Sling [7] (un entorno de desarrollo web basado en REST que permite la inclusión de scripts en diferentes lenguajes).

A continuación se describen varias implementaciones disponibles de esta especificación:

2.6.1 Apache Félix OSGi Bundle Repository

La implementación de OBR de referencia de Apache (incluida por defecto en el entorno Karaf) permite la instalación y despliegue de bundles en un entorno OSGi, resolviendo automáticamente las dependencias necesarias para su ejecución. Las entidades que define son las siguientes (la Figura 9 representa dichas entidades):

- Repository Admin. Este elemento define una agrupación de diferentes repositorios relacionados entre sí, con el objetivo de crear una federación de los mismos.
- Repository. Un repositorio específico.
- Resource. Un recurso es cualquier elemento que puede ser accedido y resuelto por el OBR.
- Capability y Requirement. Como ya se ha explicado, son los elementos exportados e importados por el recurso.
- Resolver. El encargado de resolver las dependencias.
- Repository file. Como se ha explicado, es el fichero que agrupa todos los metadatos del repositorio o conjunto de repositorios.

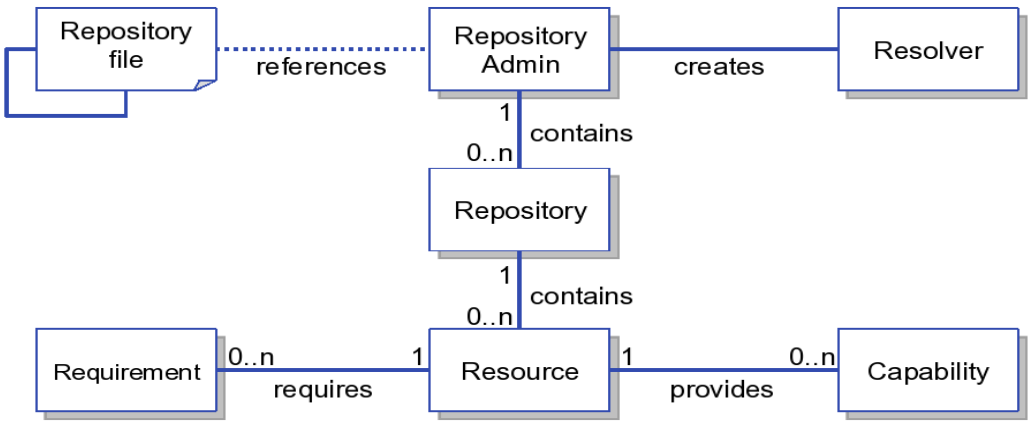

Figura 9. Entidades del OBR de Apache Felix.

[Fuente: http://felix.apache.org/site/apache-felix-osgi-bundle-repository.html]

La implementación de Apache Félix proporciona una consola de ejecución para su control. A continuación se describirán los más importantes. El resto de comandos, así como documentación de desarrollo puede consultarse en [1].

- **obr add-url**
  Este comando recibe la ubicación de un repository file y procesa la metainformación contenida, con el objetivo de ponerla a disposición del entorno OSGi.

- **obr list**
  Una vez añadida la información de los repositorios, este comando permite consultar los bundles que están disponibles para ser desplegados en el sistema.

- **obr deploy**
  Este comando trata de desplegar el bundle especificado, resolviendo sus dependencias por defecto y dejando el bundle en el estado de "Installed".

- **obr start**
  El comando start realiza lo mismo que el anterior, solo que una vez resueltas las dependencias, trata de poner el bundle en ejecución (estado "Started").

- **obr source y obr javadoc**
  Finalmente, estos comandos se utilizan para descargar tanto el código fuente como los documentos JavaDocs a una carpeta local.

Si bien las funcionalidades de Apache Félix OBR son suficientes para su utilización conjunta con la pasarela BatMP, con la idea de cumplir los objetivos enunciados en la sección 1.3 se hace necesario realizar las siguientes acciones:

- Evaluar de un contenedor de OSGi que permita la integración de otros servicios y componentes.
- Implementar una capa de abstracción sobre el OBR que permita su acceso remoto a través de una interfaz web.
- Implementar un entorno de pruebas que permita la ejecución simultánea de diferentes contenedores OSGi que simulen diferentes pasarelas locales.
- Implementar un servicio web externo a OSGi que permita reducir la memoria necesaria para la ejecución de dicho proxy

A continuación se describirán brevemente otras implementaciones de la especificación OBR, Eclipse Orbit y Enterprise Bundle Repository.

2.6.2 Eclipse Orbit

Eclipse Orbit[21] es un proyecto diseñado para ser un repositorio de librerías distribuidas como bundles y que han sido aprobadas para su utilización dentro de proyectos Eclipse (como por ejemplo la creación de un nuevo plugin). Funciona como un proyecto colaborativo en donde se intentan unificar el nombre, la estructura y las variaciones de versión de los bundles de librerías de terceras partes con el objetivo de que estén disponibles para su uso por parte de la comunidad de desarrolladores.

2.6.3 Enterprise Bundle Repository

SpringSource Bundle Repository[22] es un repositorio de bundles OSGi que actualmente se encuentra en fase de transición al proyecto Eclipse Bundle Recipes[23]. Contiene numerosos bundles empleados en el desarrollo de aplicaciones utilizando el framework de desarrollo Spring.

[21] http://wiki.eclipse.org/Main_Page
[22] http://ebr.springsource.com/repository/app/
[23] http://www.eclipse.org/ebr/

3 Contenedores de OSGi

Con el objetivo de desarrollar aplicaciones modulares, aunque no es necesario, sí es recomendable la utilización de contenedores OSGi. Mientras que hasta este punto sólo se ha descrito la arquitectura del propio framework de OSGi, para facilitar el desarrollo de aplicaciones más complejas, la utilización de un contenedor que permita integrar diferentes componentes resulta de utilidad. A continuación se describen algunos contenedores OSGi, empezando por el utilizado en la pasarela BatMP (Pax Runner) y algunas de sus limitaciones. De esta forma, se han evaluado las diferentes posibilidades que ofrecen los contenedores OSGi disponibles con el objetivo de utilizar uno de ellos para el desarrollo de este trabajo.

3.1 Pax Runner

Pax Runner [8] es un contenedor ligero de OSGi que mediante la utilización de una serie de scripts, permite iniciar una ejecución de OSGi. Estos scripts admiten diferentes opciones, especificadas con la sintaxis "--name=value", donde se puede definir la implementación OSGi que se quiere utilizar, la versión de Java del sistema o el nivel de log con el que va a arrancar. Además, Pax Runner posee un plugin del IDE Eclipse para crear diferentes configuraciones de ejecución (launch configurations) sobre diferentes frameworks (aparte de Equinox).

Una de sus características es que estas ejecuciones pueden controlarse especificando "grupos de bundles"que van a ser cargados directamente en el entorno. Estos grupos se denominan profiles, y son básicamente archivos de texto que especifican localizaciones desde donde descargar los diferentes bundles.

Aunque Pax Runner es descrita como una herramienta de provisión de bundles (provisioning tool), no es capaz de proporcionar éstos de una forma verdaderamente dinámica (es decir, en ejecución); es necesario especificar uno por uno cada uno de los bundles que se quieren poner en ejecución previamente a iniciarlo (hot deployment), utilizando diferentes scripts para cada "entorno" que se quiere utilizar, ya sea de test o de producción. Estas son algunas de las principales limitaciones de la pasarela BatMP y las cuales se pretenden abordar mediante la utilización de otro contenedor OSGi así como de un OBR.

3.2 Apache ACE

Apache ACE [2] es un framework de distribución de OSGi que permite controlar de manera dinámica la gestión de dependencias y despliegue de bundles en un entorno distribuido. Funciona con una arquitectura cliente-servidor en donde el servidor es capaz de controlar la distribución de bundles OSGi en una serie de clientes (que son básicamente entornos OSGi con un agente ACE ejecutándose en ellos).

Apache ACE permite agrupar los bundles que van a ser instalados en un determinado sistema en las llamadas features. Dichas features pueden agruparse a su vez en las llamadas distribuciones.

Apache ACE tiene un modo de operación completamente distinto al de Apache Félix OBR. En el modo de funcionamiento de OBR, es el propio cliente el que activamente intenta buscar los bundles necesarios para poner en ejecución otros bundles. En Apache ACE por el contrario, se seleccionan un conjunto de bundles que deben ser instalados en un conjunto de clientes remotos y, según éstos estén disponibles o sean accesibles, ACE trata de cargar dichos bundles en los clientes. La principal ventaja de Apache ACE es la posibilidad de controlar varios entornos OSGi simultáneamente, si bien su sistema de resolución no es tan flexible como el de OBR.

3.3 Apache Karaf

Apache Karaf es un entorno de ejecución de OSGi que proporciona un contenedor ligero donde desplegar diferentes componentes y aplicaciones. Permite integrar bien la implementación de OSGi Apache Félix o bien Eclipse Equinox, así como proporcionar diferentes servicios al entorno. Sus principales características son:

- Configuración dinámica
  Permite configurar dinámicamente aspectos de configuración de las instancias. Todos los ficheros de configuración (localizados en la carpeta /etc de la instalación) son interpretados dinámicamente.

- Gestión remota
  Para poder gestionar remotamente las instancias de Karaf, éste permite la gestión de diversos parámetros mediante JMX así como mediante un cliente SSH.

- Consola de ejecución
  A semejanza de entornos OSGi como Equinox, Karaf incorpora una consola de ejecución, agrupando los comandos en módulos que proporcionan determinadas funcionalidades. Estas funcionalidades pueden extenderse mediante la instalación de *features*.

- Despliegue dinámico.
  Karaf despliega automáticamente bundles o aplicaciones web de Java (proyectos con la extensión .war). Esta funcionalidad es similar a la que se realiza mediante la clase de OSGi ServiceTracker[24], que permite detectar cuándo

[24] http://www.osgi.org/javadoc/r4v42/org/osgi/util/tracker/ServiceTracker.html

hay disponible un nuevo servicio en el contexto de OSGi para ponerlo a disposición del resto de componentes.

- Sistema de log
  Karaf proporciona soporte de diferentes entornos de Logging, centralizando la configuración del entorno. Los entornos de logging (Java Logging Framework) son una característica muy útil de Java, que permite separar el proceso de generar mensajes de log del propio formateo del mensaje y su escritura en cualquier medio (ya sea consola, fichero de texto o base de datos).

  Por defecto, Karaf utiliza Pax Logging[25], un proyecto de Open Participation Software 4 Java (OPS4J)[26]. El archivo de configuración principal para el logger es etc/org.ops4j.pax.logging.cfg, en el cual es posible, entre otras cosas, modificar el nivel de log de las instancias de Karaf (modificando la línea log4j.rootLogger=INFO, out, osgi:* con el nivel de log deseado (FATAL, ERROR, WARN, INFO, DEBUG o TRACE), así como crear diferentes loggers dedicados a determinados componentes.

- Seguridad
  Karaf utiliza Java Authentication and Authorization Service (JAAS).Esta tecnología, a grandes rasgos permite separar las acciones genéricas de la autenticación (como por ejemplo el login de un usuario) y autorización (como por ejemplo la gestión de acceso a un determinado recurso en base a los permisos) de las implementaciones específicas subyacentes. De este modo, utilizando JAAS es posible, mediante la misma interfaz, la autenticación utilizando diferentes métodos e implementaciones de forma transparente al proceso que invoca las acciones de seguridad.

  Los elementos básicos de JAAS son, en primer lugar, un cliente (JAAS Client), que es la capa de abstracción sobre la cual se realizan las acciones de seguridad, en segundo lugar el servicio de autenticación (JAAS-compliant authentication service), que consta de diferentes módulos de autenticación, independientes entre sí, lo cual permite diferentes combinaciones y finalmente un archivo de configuración (JAAS Configuration File), que utiliza el cliente para saber qué módulos de autenticación tiene que usar. La Figura 10 ilustra la estructura de JAAS.

[25] https://ops4j1.jira.com/wiki/display/paxlogging/Pax+Logging
[26] https://ops4j1.jira.com/wiki/display/ops4j/Open+Participation+Software+for+Java

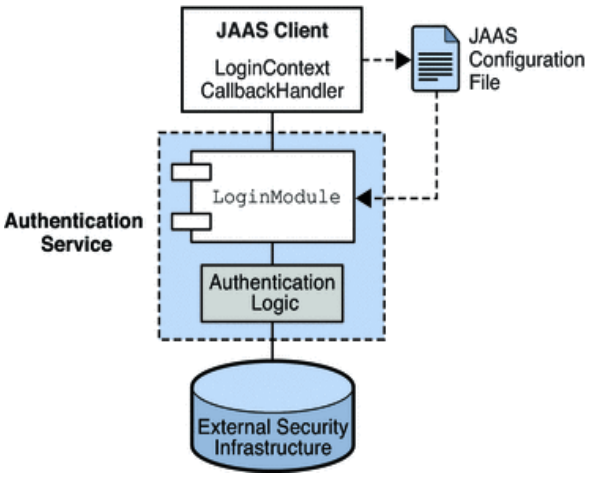

Figura 10. Estructura de JAAS.

[Fuente: http://half-wit4u.blogspot.com.es/]

- Sistema de instancias y features
  Karaf proporciona un sistema de creación de múltiples instancias dependientes de una instancia principal. Desde esta instancia principal (root) es posible crear instancias hijas de dos formas. Bien mediante creación simple, que genera una nueva instalación de Karaf con los componentes básicos o bien mediante el clonado, que crea una instancia copia de otra. En ambos casos, cada nueva instancia se ejecuta en su propia JVM, proporcionando de este modo aislamiento a nivel de proceso, evitando que una caída de una de las instancias afecte a las demás.

  Una de las características más interesantes de Karaf es su sistema de ampliación de funcionalidades, que permite descargar e instalar nuevas aplicaciones para ser ejecutadas en el entorno. Estas aplicaciones, llamadas features, contienen el nombre, la versión, la descripción y el conjunto de bundles que la implementan, así como el conjunto de dependencias de la misma (en el caso de que las requieran). Así, cuando se desea instalar una nueva feature, el propio entorno de Karaf se encarga de resolver las dependencias y de poner en funcionamiento la nueva feature (incluyendo los nuevos comandos que ésta proporciona).

La Figura 11 representa la estructura de Apache Karaf.

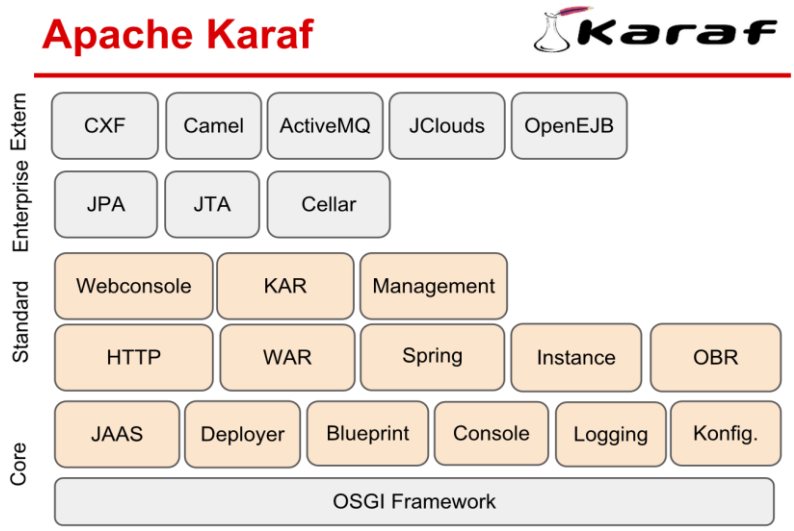

Figura 11. Estructura de Apache Karaf.

[Fuente: http://www.liquid-reality.de/]

En resumen, uno de los objetivos de este proyecto es intentar combinar las ventajas de ambos contenedores, por un lado utilizando la flexibilidad de Karaf y de OBR y por otro lado permitir algún tipo de control sobre varias instancias OSGi como permite Apache ACE. En este sentido, se ha optado por utilizar Apache Karaf para la realización de estos desarrollos con el objetivo a largo plazo de utilizar dicho contenedor en la pasarela BatMP y de este modo intentar sacar partido de algunas de sus características.

4 Apache Karaf en detalle

Dado que se ha decidido utilizar el framework Karaf para la realización de este trabajo, a continuación se explican en detalle el proceso de instalación y sus principales componentes.

4.1 Instalación de Apache Karaf

La instalación de Karaf en entornos Unix puede realizarse, bien construyendo el proyecto desde el código fuente, o bien descargando directamente los binarios desde la página principal del proyecto. El único requisito es tener la variable de entorno JAVA_HOME correctamente configurada. Los detalles particulares pueden consultarse en la página de instalación de proyecto[27]. Una vez instalado, ejecutando un script se arranca la instancia principal del entorno. Existe la opción de integrar la ejecución de Karaf en el sistema operativo, de forma análoga a la de un proceso *daemon* de Unix. Para instalar esta característica es necesario instalar una nueva feature denominada wrapper, y generar un script en la carpeta /etc/init.d.

Una vez en ejecución, una instancia básica de Karaf (sin añadir nuevas features aparte de las que vienen por defecto) requiere entre 300 y 400 MBytes de memoria. Aunque para una instalación básica esto es suficiente, para los desarrollos y pruebas llevados a cabo en este trabajo se ha optado por ampliar la memoria que puede ser utilizada por las instancias de Karaf. Estos valores pueden ser modificados en el archivo bin/setEnv. Los valores que se han utilizado son los siguientes:

```
export JAVA_MIN_MEM=128M # Minimum memory for the JVM
export JAVA_MAX_MEM=1024M # Maximum memory for the JVM
export JAVA_PERM_MEM=128M # Minimum perm memory for the JVM
export JAVA_MAX_PERM_MEM=256M # Maximum perm memory for the JVM
```

4.2 Arquitectura de Apache Karaf. Principales componentes

Los principales componentes de Apache Karaf son los siguientes:

4.2.1 Implementación OSGi

Como se ha comentado, la instalación de Karaf incluye por defecto la implementación OSGi de Apache Félix, aunque es posible utilizar la implementación Eclipse Equinox. Para la elaboración de este trabajo se ha utilizado la implementación por defecto, si

[27] http://karaf.apache.org/manual/latest/users-guide/installation.html

bien ésto se podría modificar mediante la edición del archivo config.properties. La parte que referencia a la implementación OSGi es la siguiente:

```
# Framework selection properties
#
karaf.framework=felix

#
# Location of the OSGi frameworks
#
karaf.framework.equinox=mvn\:org.eclipse/org.eclipse.osgi/3.8.2.v20130
124-134944
karaf.framework.felix=mvn\:org.apache.felix/org.apache.felix.framework
/4.2.1
```

Como se puede observar, a la localización del entorno OSGi se accede directamente a través de un repositorio Maven. Es decir, no es necesario descargar la versión deseada directamente en la máquina que ejecuta la instancia, ya que el plugin de Maven se encargará de descargar los .JAR necesarios y ponerlos en ejecución. Finalmente, es necesario que Karaf sepa dónde encontrar los repositorios de Maven, es decir, las URL donde acceder. Esto se consigue editando el archivo de configuración /etc/org.ops4j.pax.url.mvn.cfg, el cual deberá modificarse para especificar las URLs de los repositorios.

```
org.ops4j.pax.url.mvn.repositories= \
      http://repo1.maven.org/maven2@id=central, \
      http://repository.springsource.com/maven/bundles/release@id=spri
ng.ebr.release, \
      http://repository.springsource.com/maven/bundles/external@id=spr
ing.ebr.external, \
      file:${karaf.home}/${karaf.default.repository}@id=system.reposit
ory, \
      file:${karaf.data}/kar@id=kar.repository@multi
```

4.2.2 Jetty

Eclipse Jetty es un servidor HTTP basado en Java y un contenedor de servlets, desarrollado como un proyecto abierto de la Eclipse Foundation. Una de las características principales de Jetty[28] es que está diseñado para estar embebido en una aplicación Java, pudiendo instanciar y manipular los .JAR de éste de la misma forma que cualquier otro objeto.

[28] http://www.eclipse.org/jetty/

Para configurar adecuadamente el servidor Jetty dentro de la instalación de Karaf es necesario editar el archivo conf/jetty.xml donde principalmente se pueden configurar los puertos donde Jetty escuchará las conexiones básicas a través de HTTP, así como las que utilicen un protocolo seguro como SSL ("confidentialPort"). La parte relevante de dicho fichero es la siguiente:

```
...
<Set name="host">
      <Property name="jetty.host" />
</Set>
<Set name="port">
      <Property name="jetty.port" default="8181" />
</Set>
<Set name="maxIdleTime">300000</Set>
<Set name="Acceptors">2</Set>
<Set name="statsOn">false</Set>
<Set name="confidentialPort">8443</Set>
<Set name="lowResourcesConnections">20000</Set>
<Set name="lowResourcesMaxIdleTime">5000</Set>
...
```

4.2.3 Blueprint-Apache Aries

Blueprint[29] es una especificación del modelo de componentes OSGi que establece de qué forma van a crearse los componentes que publican o referencian servicios OSGi (es decir, principalmente los bundles). Esta información se indica mediante un archivo XML específico. Por ejemplo, el siguiente archivo localizado en la carpeta OSGI-INF/blueprint de un proyecto OSGi especifica uno de los componentes:

```
<components xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">
<description>
      Descripción del componente.
</description>
<component id="component" class="es.upm.cedint.component.Class">
</component>
...
</components>
```

Apache Aries[30] es una implementación de Blueprint que incluye además implementaciones de otros componentes Java, tales como JMX, Java Persistence API (JPA) o Java Naming Directory Specification (JNDI).

[29] http://wiki.osgi.org/wiki/Blueprint
[30] http://aries.apache.org/

Finalmente, en el entorno OSGi es común el modelo de componentes denominado Declarative Services (DS). La diferencia entre DS y Blueprint es que en el primero, los componentes no son instanciados hasta que todas sus dependencias pueden ser satisfechas (es decir, cuando todos los bundles de los que depende han sido creados y están disponibles en el entorno OSGi), mientras que en el segundo caso, los componentes son creados tan pronto como el bundle es cargado. Si en este caso, se llama a un componente que depende de un bundle que no ha sido cargado, se crea un proxy que retrasa la llamada (es decir, bloquea al bundle llamante) hasta que el servicio esté presente.

4.2.4 Features

Una de las características de Karaf es la posibilidad de agrupar diferentes bundles de una aplicación en una misma unidad lógica. De esta forma, es posible empaquetar todos los bundles, dependencias y archivos de configuración en una misma entidad. De esta forma, una feature describe una aplicación con los siguientes elementos:

- Nombre.
- Versión.
- Descripción de la feature.
- Conjunto de bundles.
- Conjunto de ficheros de configuración.
- Conjunto de dependencias.

La Figura 12 representa el concepto de feature.

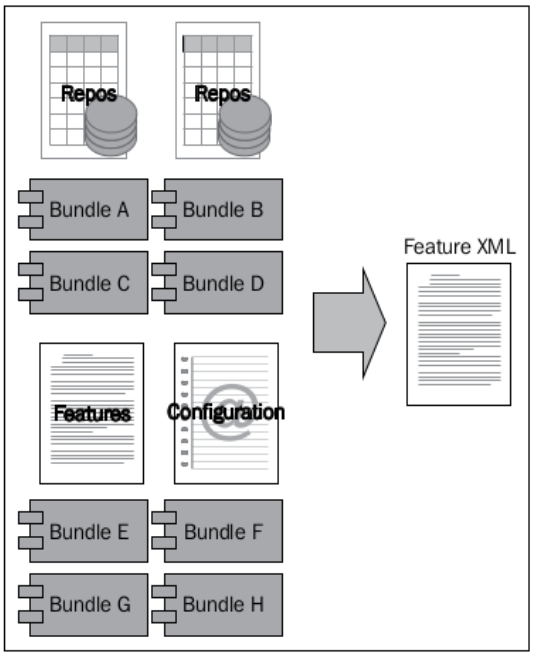

Figura 12. Features de Karaf.

4.2.5 JMX MBean server

Karaf proporciona un servidor JMX para monitorizar diferentes aspectos de las instancias en ejecución, ya que permite el acceso a un conjunto de MBeans dedicados a la monitorización y la gestión de los componentes. Por ello, utilizando un cliente JMX como por ejemplo jconsole[31], es posible visualizar el estado de los diferentes parámetros de las instancias activas.

La URL por defecto para conectarse al servidor JMX es:

```
service:jmx:rmi:///jndi/rmi://localhost:1099/karaf-root
```

El puerto por defecto del registro RMI es el 1099, aunque puede modificarse a través del fichero de configuración etc/org.apache.karaf.management.cfg. En dicho fichero, es posible modificar también el puerto del servidor RMI que recibe las conexiones.Este mecanismo es también utilizado en otras aplicaciones como por ejemplo la consulta por parte de un servlet de la configuración de un contenedor web.

Como se verá más adelante, el hecho de tener que utilizar estos puertos plantea algunos problemas cuando la instancia se encuentra detrás de un firewall (como es el caso de la pasarela BatMP instalada en CeDInt).

La Figura 13 muestra una captura de pantalla de jconsole conectado a la instancia principal "root" de Karaf. Se pueden ver las operaciones accesibles a través de dicho cliente, tanto las referidas a cambios de configuración como las orientadas al control de las instancias.

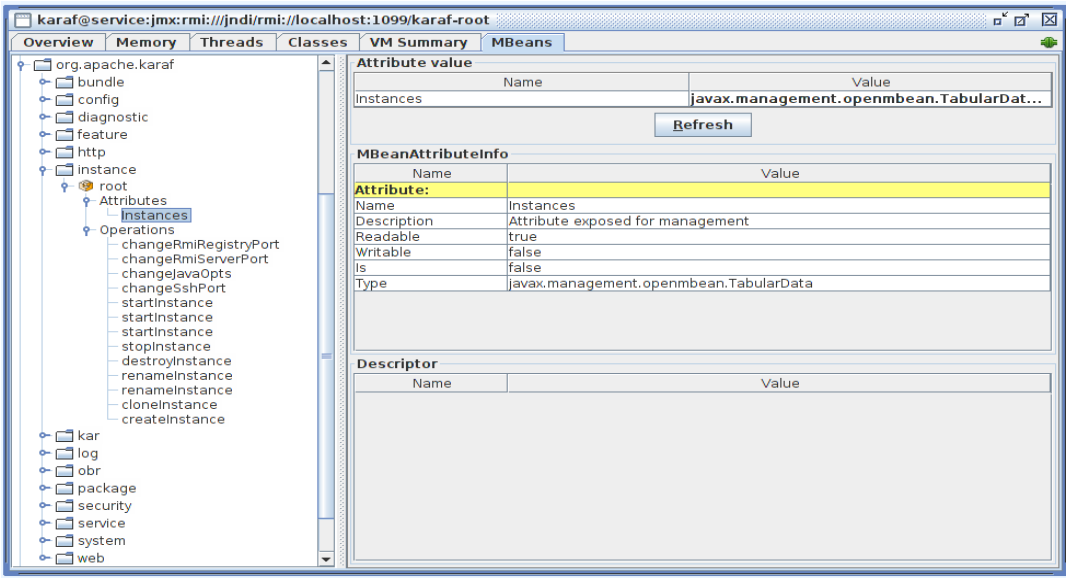

Figura 13. Cliente JMX jconsole conectado a la instancia "root" de Karaf.

[31] http://openjdk.java.net/tools/svc/jconsole/

4.2.6 Webconsole

Es posible instalar una feature opcional de Karaf que permite conectarse al servidor JMX de Karaf utilizando un navegador web. Esta consola proporciona una interfaz gráfica para controlar el estado del sistema, los artefactos y bundles desplegados, así como otros aspectos de configuración.

La Figura 14 ilustra el aspecto del webconsole.

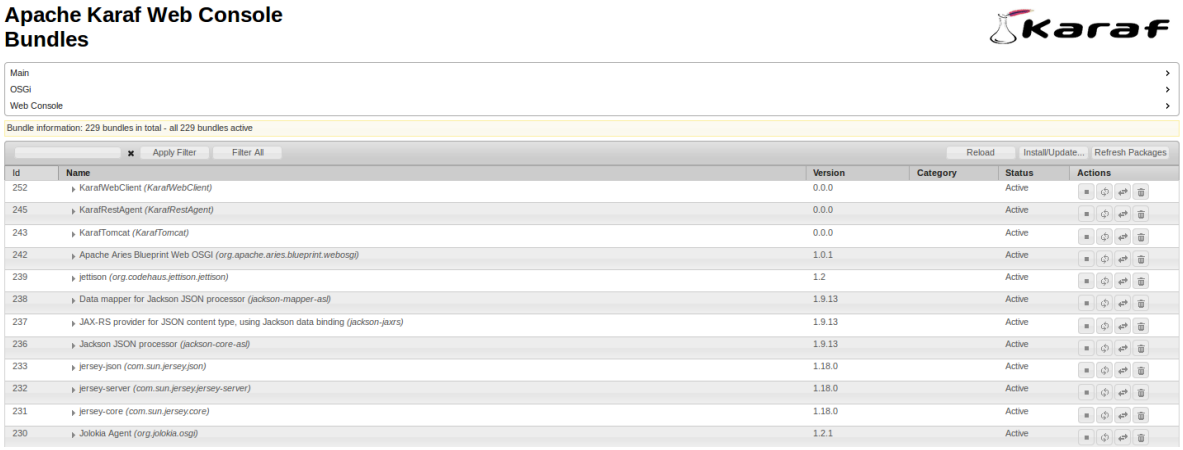

Figura 14. Webconsole de Karaf.

Si bien a través de la webconsole de Karaf es posible controlar muchos de los aspectos de la ejecución de Karaf, la utilización de webconsole en la pasarela BatMP plantea una serie de problemas:

- Dada la abundancia de opciones de control disponibles, su utilización no resulta intuitiva para alguien no familiarizado con un entorno OSGi. Por este motivo, se hace necesaria la utilización de una consola web de funcionalidades más reducidas y centradas en determinados aspectos de la gestión de Karaf.

- Cada instancia de webconsole permite controlar una única instancia de Karaf. Por lo tanto, si se desea controlar vía web simultáneamente más de una instancia, la única manera sería tener abiertas tantas webconsoles como instancias a controlar, por lo que esta solución resulta un tanto engorrosa.

Por estos motivos, se ha optado por desarrollar un cliente específico con las funcionalidades imprescindibles y que además permita visualizar simultáneamente el estado de más de una instancia Karaf.

4.2.7 Despliegue automático de aplicaciones web (WAR)

Como se ha comentado, Karaf permite el despliegue automático de diferentes tipos de archivos, entre ellos los Web ARchive (WAR) de Java. Estos archivos son similares a los JARs solo que contienen un conjunto de servlets Java, archivos XML, páginas HTML y archivos JavaScript, entre otros, con el objetivo de construir una aplicación web. Estos archivos WAR se pueden desplegar automáticamente en servidores como Glassfish[32] ó Tomcat. En este caso, Karaf permite el despliegue automático de este tipo de archivos mediante la copia del archivo WAR correspondiente en el directorio /deploy de la instalación de Karaf. Como se verá en el siguiente apartado, se ha hecho uso de esta característica para el despliegue de uno de los componentes desarrollados.

4.2.8 Despliegue de aplicaciones no compatibles con OSGi

Finalmente, una característica interesante de Karaf es la posibilidad de "envolver" (wrap) automáticamente un archivo JAR de forma que sea compatible con OSGi. Estas acciones en ocasiones pueden ser necesarias para integrar determinados componentes en un entorno OSGi que de otra forma no se podrían ejecutar. Si bien es posible hacer esto utilizando un plugin específico de Maven (maven bundle plugin), Karaf tiene la ventaja de que realiza automáticamente ese wrapper sin tener que recurrir a recompilar el JAR correspondiente. Esto se consigue, al igual que con los WAR, guardando directamente el archivo JAR en la carpeta /deploy de la instalación.
En el fondo, el "truco" que está realizando Karaf es simplemente la generación automática de un archivo MANIFES.MF que por defecto importa todos los paquetes de dicho JAR (import *). No obstante, dada la popularidad de OSGi, existen numerosas versiones convertidas a bundles OSGi de muchas librerías.

[32] https://glassfish.java.net/

5 Diseño de un External Karaf Web Proxy

A la hora de realizar una instalación de diferentes pasarelas BatMP controladas por un servidor central en un entorno como el Campus de Montegancedo, es necesario abordar el problema de las comunicaciones entre el servidor central y cada una de las mismas. Puesto que cada pasarela BatMP puede estar situada en una red interna de cada edificio, y por lo tanto detrás de un firewall corporativo, pueden aparecer dificultades a la hora de establecer conexiones con "puertos no estándar" entre el exterior de la red y la pasarela BatMP.

Por lo general, los firewalls que protegen estas redes suelen implementar la política de por defecto denegar una conexión que se sale de la "normalidad", es decir, toda conexión que no sea HTTP, HTTPS o perteneciente a algún servicio explícitamente permitido como el correo electrónico.

En este apartado se describe el diseño de una aplicación web ejecutada sobre Karaf para la gestión remota de las instancias Karaf y de los bundles que se ejecutan en ellas. En primer lugar se explica tanto el proceso de instalación de este framework como sus principales componentes. A continuación, se detallan las dos partes principales del desarrollo realizado. Al conjunto de desarrollos realizados se le ha denominado, de forma genérica, External Karaf Web Proxy. El motivo de esta denominación es que, aunque el principal objetivo de los desarrollos ha estado centrado en la gestión de los bundles de OSGi de forma remota, a lo largo de la elaboración del trabajo se ha generalizado este concepto para permitir el control de otro tipo de componentes, no únicamente bundles. De ahí que se haya utilizado el término más genérico Web Proxy.

El Web Proxy desarrollado consta de dos elementos principales:

- Karaf Web Client
  Este componente es un bundle OSGi que se ejecuta en cada instancia de Karaf dentro del servidor web Jetty, y se encarga de recibir y procesar las peticiones de gestión de los bundles.

- Central Server
  El Central Server es una aplicación web desarrollada en Java y Javascript que se ejecuta en cualquier servidor web y permite monitorizar dinámicamente el estado de diferentes instancias Karaf.

A continuación se describen ambos componentes en detalle:

5.1 Arquitectura de Karaf Web Client

El Karaf Web Client consta de tres paquetes principales:

- es.upm.kwebclient.core
  Este paquete posee las clases principales del servlet (WebAgent) y del objeto que interactúa mediante RMI con los MBeans de Karaf (Client).

- es.upm.kwebclient.elements
  Este paquete posee clases para representar bundles y repositorios. Además, en este paquete se encuentra la interfaz Provider que es la interfaz que se implementará en cada instancia de Karaf con el objetivo de controlar los diferentes parámetros de la instancia.

- es.upm.kwebclient.util
  Este paquete posee clases para la representación de mensajes. Estos mensajes tienen el mismo formato que los mensajes que se intercambian los diferentes componentes de BatMP, y se han incluido para su posterior integración con la pasarela.

La interfaz Provider consta de los siguientes métodos:

```
public interface Provider {
	public Message installRepository(String name, String uri);

	public Message uninstallRepository(String name);

	public Message installBundle(String name);

	public Message startBundle(String name);

	public Message stopBundle(String name);

	public Message uninstallBundle(String name);

	public boolean containsBundle(String name);
}
```

Estos métodos se implementan en la clase Client, la cual recibe las acciones a realizar del servlet WebAgent. Con el objetivo de controlar mediante JMX los diferentes parámetros de la instancia, Karaf Web Client utiliza las siguientes clases JMX:

- JMXServiceURL[33]
  Esta clase permite establecer una conexión con una instancia JMX mediante diferentes protocolos. Cada URL de conexión tiene el formato: service:jmx:protocol:address.

[33] http://docs.oracle.com/javase/7/docs/api/javax/management/remote/JMXServiceURL.html

En el caso de Karaf Web Client, la URL que se ha especificado para las pruebas en local es: service:jmx:rmi:///jndi/rmi://localhost:1099/karaf-root
El elemento service es una constante, seguida de jmx:rmi que especifica que el tipo de servicio es JMX y que el esquema de la URL es el empleado en RMI. A continuación, se especifica jndi para indicar que el cliente JMX puede encontrar la información de conexión utilizando JNDI. Finalmente, rmi://localhost:1099/karaf-rootindica que en la máquina localhost existe un servidor JMX que utiliza RMI y que es accesible a través de la clave karaf-root.

- JMXConnector[34]
  A través de este objeto se establece una conexión entre un cliente y un servidor JMX. En este contexto, el cliente JMX es cada instancia Karaf Web Client y el servidor JMX es la propia instancia Karaf que implementa un servidor JMX con los MBeans expuestos. Este objeto sirve también para establecer una comunicación segura entre las dos partes, ya que al establecer la conexión es necesario facilitar las credenciales (mapa con nombre y password de la instancia). Además, es posible especificar diferentes protocolos de cifrado, como por ejemplo SSL.

La Figura 15 muestra la estructura de Karaf Web Client en el IDE Eclipse:

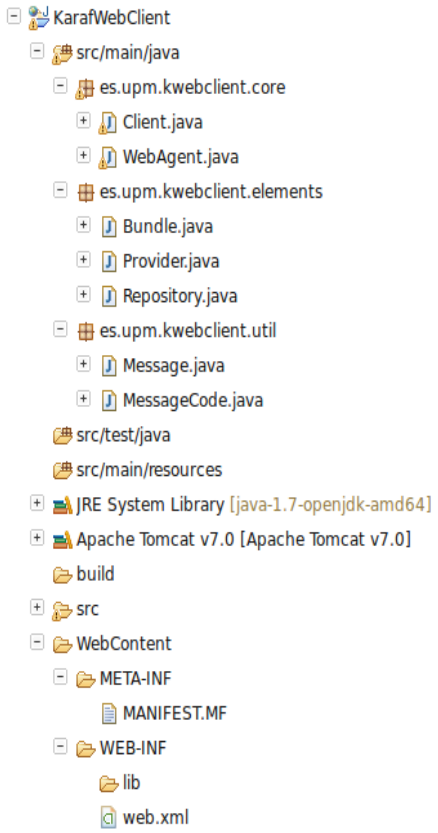

Figura 15. Estructura de Karaf Web Client.

[34] http://docs.oracle.com/javase/7/docs/api/javax/management/remote/JMXConnector.html

Como se ha mencionado, la clase Client implementa los métodos de la interfaz Provider. Al instanciarse un nuevo objeto Client, lo primero que se intenta es establecer una conexión con el servidor JMX del host:

```
// Class contructor
public Client(){
this.connectToJMXService();
try {
       this.mbsc = this.jmxc.getMBeanServerConnection();
    } catch (IOException e) {
      e.printStackTrace();
    }
    this.bundleMap = new HashMap<String, Bundle>();
...

// Establishes a connection to the JMX server
private void connectToJMXService(){
try {
        this.url = new JMXServiceURL(serviceURL);
        this.env = new HashMap<String, String[]>();
this.env.put(JMXConnector.CREDENTIALS, creds);
        this.jmxc = JMXConnectorFactory.connect(url, env);
    } catch (IOException e) {
      echo(" Exception connecting to the server");
      e.printStackTrace();
    }
 }
```

Una vez establecida la conexión, los diferentes métodos de la interfaz Provider serán llamados desde el servlet.

El método installRepository se ejecuta cada vez que se recibe una llamada para incluir un nuevo repositorio en la instancia (por simplicidad, sólo se mostrará la implementación de este método, ya que la implementación de los demás es muy similar).

```
public Message installRepository(String repositoryName, String uri) {
   Message result = new Message();
result.setMessageCode(MessageCode.ERROR);
result.setDescription("Repositorio no instalado");
ObjectName name = null;
      try {
      if (!this.containsRepository(repositoryName)){
      name = new
ObjectName("org.apache.karaf:type=obr,name="+this.instanceName);
```

```
      mbsc.invoke(name, "addUrl", new Object[]{uri}, new
String[]{String.class.getName()});
            result.setMessageCode(MessageCode.SUCCESS);
            result.setDescription("Repositorio instalado
correctamente");
      // Create the new repository object in the map
      Repository newRepository = new Repository(repositoryName, uri);
      this.repositoryMap.put(repositoryName, newRepository);
            }

       } catch (MalformedObjectNameException e) {
            ...
       }
      return result;
}
```

En este método, tras comprobar si la instancia contiene el repositorio que se pretende añadir, se invoca el método addUrl del MBean OBR de la instancia de Karaf org.apache.karaf:type=obr,name=root. Los otros métodos de la interfaz Provider se implementan de forma muy similar. Los diferentes métodos que se invocan del MBean OBR se representan en la Figura 16.

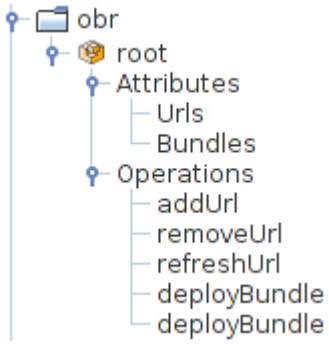

Figura16. Operaciones del MBean OBR.

La clase Message se utiliza para el intercambio de mensajes entre diferentes bundles al realizar llamadas entre ellos. Incluye un código del mensaje (ERROR, SUCCESS, WARNING) y una descripción del mismo.

```
public class Message implements Serializable{

private static final long serialVersionUID = 1L;
private MessageCode messageCode;
private String description;
…
}
```

La clase WebAgent es un servlet que recibe las peticiones HTTP del Central Server. Estas peticiones deben tener el formato http://<host>:<puerto>/WebAgent/<acción a realizar>. La raíz de la URL se especifica de dos formas. Por un lado, mediante una anotación Java[35] al principio de la clase WebAgent:

```
@WebServlet("/WebAgent")
public class WebAgent extends HttpServlet {
private static final long serialVersionUID = 1L;
…
```

Por otro lado, para que el servidor web en el que se está ejecutando el cliente sepa dónde dirigir las peticiones web que llegan a él, es necesario configurar adecuadamente el archivo web.xml de la siguiente forma:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd" id="WebApp_ID"
version="3.0">
<display-name>KarafWebClient</display-name>
<welcome-file-list>
      <welcome-file>index.html</welcome-file>
</welcome-file-list>
<servlet>
      <servlet-name>WebAgent</servlet-name>
      <servlet-class>es.upm.kwebclient.core.WebAgent</servlet-class>
</servlet>
<servlet-mapping>
      <servlet-name>WebAgent</servlet-name>
      <url-pattern>/WebAgent</url-pattern>
</servlet-mapping>
</web-app>
```

De esta forma, se indica la clase del servlet que procesará las peticiones utilizando el patrón URL KarafWebClient/WebAgent.

La clase WebAgent, al extender de la interfaz HttpServlet debe implementar los métodos doGet y doPost. Los comandos que se reciben en este servlet (procedentes del Central Server) codifican en la URL los parámetros de la llamada. De esta forma, cada llamada tiene la forma:

```
URL/KarafWebClient/WebAgent?nombre_metodo=parametro_metodo
```

[35] http://docs.oracle.com/javase/tutorial/java/annotations/

```
protected void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {

    Enumeration<String> parameterNames = request.getParameterNames();
while (parameterNames.hasMoreElements()) {
      if (paramName != null){
            String[] paramValues =
request.getParameterValues(paramName);
            if (paramName.equals("Contains")){
                  responseString +=
this.containsBundle(paramValues[0]);
}
...
```

Como su nombre indica, el método doGet es invocado cada vez que se recibe un mensaje Http GET. Desde este método se invocan aquellas llamadas que no provocan un cambio en la instancia Karaf, como por ejemplo Contains, para consultar si la instancia posee un bundle determinado, CheckBundleState, para comprobar el estado OSGi del bundle especificado, o CheckInstanceState, para comprobar si alguna instancia se encuentra activa. Este último método para permitir el control de varias instancias Karaf presentes en la misma máquina.

El método doPost por el contrario procesa las llamadas que acarrean alguna modificación en la instancia, como por ejemplo aquellas que instalan o desinstalan un bundle o lo arrancan o paran.

5.2 Arquitectura del Central Server

El Central Server es el programa desde el cual es posible controlar diferentes instancias de Karaf. Es una aplicación web desarrollada en JavaScript que permite visualizar en tiempo real el estado de las diferentes instancias que se hayan dado de alta en la propia aplicación. Para el desarrollo de este proyecto se han dado de alta diferentes instancias Karaf que emulan diferentes pasarelas BatMP que estarían instaladas en edificios del Campus de Montegancedo.

El nombre de cada instancia se corresponde con el de algún edificio de dicho campus (informática, cedint, imdea etc.), por lo que son estos nombres los que aparecen en la aplicación web. Además, con el objetivo de probar la aplicación con instancias que no se encuentren en la misma máquina, se ha creado una instancia Karaf en una máquina virtual utilizando VNX a la que también se accede desde dicha aplicación (la instancia se denomina vnx).

La estructura del proyecto se representa en la Figura 17.

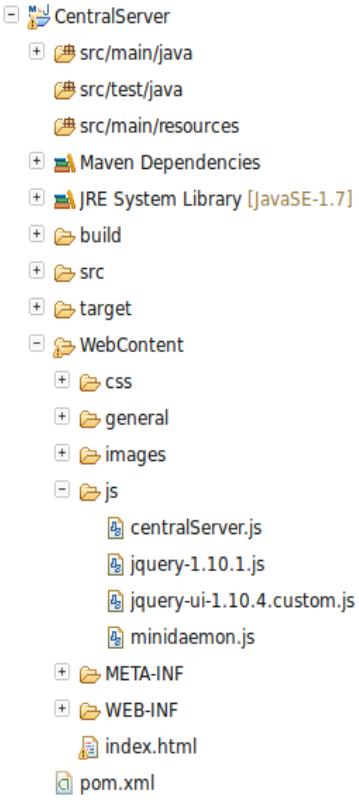

Figura 17. Estructura del proyecto Central Server.

El script centralServer.js utiliza la conocida librería jQuery[36] para la comunicación asíncrona con el objeto Karaf Web Client de cada instancia. Las funcionalidades del script son las siguientes:

- Refresco automático del estado de las instancias
  Una vez se accede a la URL donde está alojado y se carga la página en el navegador web, se inicia un contador que refresca cada pocos segundos los indicadores de estado de cada una de las instancias.
  En verde se especifica que la instancia está activa. En rojo, que está inactiva, y en amarillo que no se ha podido acceder al cliente Karaf Web Client de esa instancia.
  El refresco automático se realiza mediante la ejecución periódica de un script (miniademon.js) que ejecuta periódicamente una determinada función que se pasa como parámetro. Por ejemplo, el siguiente código programa la ejecución de la función checkInstancesStatus cada 5 segundos (5000 milisegundos). Más abajo se explica dicha función.

```
var demonio = new MiniDaemon(this, checkInstancesStatus, 5000);
demonio.start();
```

[36] http://jquery.com/

- Instalación y desinstalación de repositorios
  Especificando la instancia sobre la que se va a actuar, es posible instalar o desinstalar un repositorio nuevo para que dicha instancia pueda descargarse bundles de él. Una vez instalado el repositorio, todos los bundles que contiene están disponibles para su descarga e instalación.
  La URL de cada repositorio está precedida por un campo que indica el protocolo que Karaf va a utilizar para leer la información de dicho repositorio. Si dicha información se encuentra en un archivo local, la URL del archivo "repository.xml" se encuentra precedida de "file:". Por el contrario, si se trata de un repositorio de Maven, la URL va precedida de ":mvn".

- Instalación y desinstalación de bundle
  Una vez se ha dado de alta un nuevo repositorio es posible instalar o desinstalar un determinado bundle utilizando estas llamadas. Este proceso no sólo se limita a los bundles de los nuevos repositorios sino a cualquier bundle del entorno de Karaf.

- Arrancar y parar bundle
  Finalmente, mediante estas llamadas es posible iniciar o detener la ejecución de un determinado bundle, especificando éste con el atributo nombre de dicho bundle.

El aspecto de la página principal del Central Server se ilustra en la Figura 18. En esta Figura se observan dos instancias activas en ese momento (root y cedint), mientras que el resto de instancias no han sido desplegadas, por lo que aparecen en color rojo. Finalmente, la instancia identificada como vnx aparece en amarillo puesto que en el momento de la captura la máquina virtual se encontraba parada, por lo que no se ha podido establecer la comunicación.

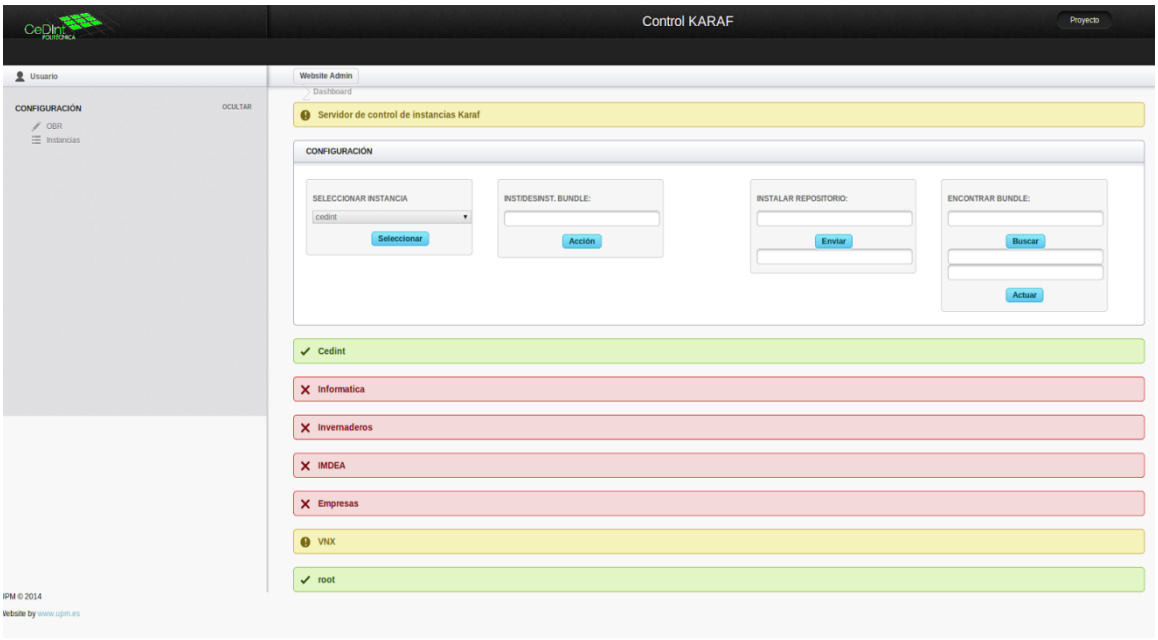

Figura 18. Aspecto inicial de la aplicación web desarrollada.

A la hora de localizar un bundle instalado en una instancia, se debe utilizar el nombre del bundle, mientras que para instalar un determinado repositorio, éste se especifica con el formato:

nombre;dirección

El campo "dirección" debe contener el prefijo del tipo de repositorio del que se trata (como por ejemplo "mvn:" para repositorios Maven o ":file" para repositorios especificados en un archivo local). En la Figura 19 se ilustra el resultado de instalar un repositorio local (y por lo tanto lleva el prefijo "file:" que contiene un bundle de prueba denominado BundleEquinox), e instalar dicho bundle.

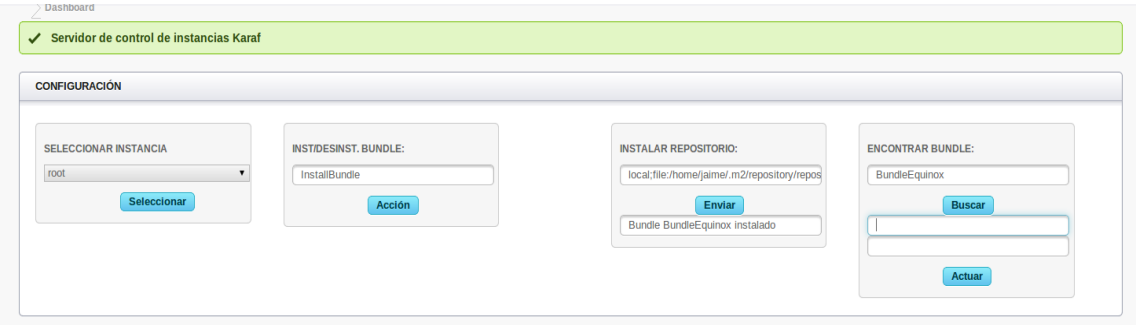

Figura 19. Instalación de un repositorio y un nuevo bundle en el sistema.

Una vez instalado, todos los bundles del repositorio pasan a estar en estado Installed, desde donde Karaf comprueba las dependencias de cada uno de dichos bundles. Si las dependencias están resueltas, los bundles pasan al estado de "Resolved", desde donde

pueden ser puestos en ejecución o detenidos desde la aplicación web mediante los comandos “StartBundle” y “StopBundle”.

Las llamadas al servlet de cada instancia se realizan utilizando Asynchronous JavaScript and XML (AJAX), un conjunto de técnicas utilizadas en los clientes web para crear aplicaciones asíncronas, es decir, aplicaciones que pueden interactuar con el servidor en cualquier momento, en segundo plano y sin interferir con la visualización normal de la página. Existen diferentes formas de implementar AJAX, si bien una explicación detallada de esta tecnología quedaría fuera del alcance de este trabajo. En el componente Central Server se ha utilizado una implementación disponible en la librería jQuery, la cual permite realizar llamadas asíncronas GET y POST de una forma sencilla. Todas las llamadas a los clientes de las instancias de Karaf se realizan utilizando esta librería.

La siguiente función ilustra las principales funcionalidades del script implementado:

```
function checkInstancesStatus(){
(1) var instances = ["cedint","invernaderos", "informatica", "imdea",
"empresas", "root"];
(2) var urlServer = centralServerURL +
"8181/KarafWebClient/WebAgent?CheckInstanceState=";
for (var i= 0; i < 7; i++){
(3)    $.ajax({
           type: 'GET',
           async: "false",
           url: urlServer + instances[i],
           localInstance : instances[i],
(4)        success: function (data){
                 console.log(" Devuelve: " + data);
                 if (data == 'Karaf HTTP Agent responses: Started')
document.getElementById(this.localInstance).className =
"alert_success";
                 else
document.getElementById(this.localInstance).className = "alert_error";
            },
           error: function(xhr, status, error){
                 console.log("Error: " + error);
           }
       });
    }
(5) var virtualServer = virtualServerURL +
"8181/KarafWebClient/WebAgent?CheckInstanceState=root";
    $.ajax({
      type: 'GET',
      async: "false",
      url: virtualServer,
```

```
      success: function (data){
            console.log(" Devuelve: " + data);
            if (data == 'Karaf HTTP Agent responses: Started')
document.getElementById("vnx").className = "alert_success";
            else document.getElementById(this.localInstance).className
= "alert_error";
       },
      error: function(xhr, status, error){
            console.log("Error: " + error);
       }
    });
}
```

Esta función es invocada cada vez que se actualiza el estado de las instancias en la página. En (1) se almacenan los nombres de las diferentes en un array. A continuación se construye la URL de la llamada para consultar el estado de cada instancia del array (2). Después, por cada elemento del array (es decir, por cada instancia) se construye la llamada AJAX que se comunicará con cada una de ellas (3) y si ésta tiene éxito (4) se altera el elemento correspondiente a dicha instancia. Como se puede observar, todas las llamadas se realizan sobre la misma instancia (instalada en localhost) que es la que posee las diferentes instancias de Karaf, si bien en un entorno real, cada llamada iría destinada a un host diferentes, como ocurre en (5), la cual es una llamada a la máquina virtual VNX que posee una instancia virtual de Karaf.

Para la construcción del Central Server se ha utilizado el gestor Maven, descrito en la sección 3.1 de este trabajo. El archivo pom.xml que se ha utilizado es el siguiente:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>
<groupId>CentralServer</groupId>
<artifactId>CentralServer</artifactId>
<version>0.0.1-SNAPSHOT</version>
<packaging>war</packaging>    (1)
<build>
      <plugins>
            <plugin>
            <artifactId>maven-compiler-plugin</artifactId> (2)
            <version>3.1</version>
            <configuration>
            <source>1.7</source>
            <target>1.7</target>
            </configuration>
            </plugin>
```

```
            <plugin>
            <artifactId>maven-war-plugin</artifactId>  (3)
            <version>2.3</version>
            <configuration>
            <warSourceDirectory>WebContent</warSourceDirectory>
            <failOnMissingWebXml>false</failOnMissingWebXml>
            </configuration>
            </plugin>
      </plugins>
</build>

<dependencies>
      <dependency>
            <groupId>com.thetransactioncompany</groupId>  (4)
            <artifactId>cors-filter</artifactId>
            <version>1.3.2</version>
      </dependency>
</dependencies>
</project>
```

En primer lugar, en este archivo se especifica que la forma de construir este proyecto es utilizando un formato WAR (1), por lo que el proyecto construido podrá ejecutarse en cualquier servidor web que pueda interpretar este tipo de archivo. Los dos plugins de Maven que se han utilizado para construir el proyecto son el compilador estándar de Maven (2) y el plugin para la construcción de archivos WAR (3). Finalmente, se ha especificado una dependencia (4) para la utilización de un filtro Cross-Origin Resource Sharing (CORS), cuya finalidad se explica en el apartado 6.2.

Finalmente, con el objetivo de dar un aspecto más estético a la página, se ha utilizado un CSS libremente disponible en Internet (bajo licencia GNU General Public License [11]).

6 Pruebas

Con el objetivo de probar el desarrollo realizado se ha creado un escenario de pruebas consistente en varias instancias Karaf, cada una de ellas con su propio Karaf Web Client instalado. Estas instancias se ejecutan, como se ha dicho, en diferentes máquinas virtuales Java y cada una de ellas tiene su propio conjunto de bundles, controlados desde el Central Server.

Además, con el objetivo de probar la aplicación de una forma más realista, se ha utilizado el software de virtualización Virtual Networks over linuX (VNX)[37] para simular una instancia Karaf remota, y de este modo probar la conexión con un host diferente.

6.1 Escenario de prueba

El escenario de prueba que se ha utilizado es una instalación con 6 instancias diferentes de Karaf en la misma máquina, así como una instalación en una máquina virtual utilizando VNX. Todas las instancias tienen instalado el Karaf Web Client.

Por otra parte, el componente Central Server se ha instalado también en la misma máquina pero sobre un servidor web Apache Tomcat.

6.2 Rendimiento

Uno de los objetivos de este desarrollo, aparte de poder controlar desde una misma aplicación las diferentes instancias de Karaf, es demostrar que los Karaf Web Client son más eficientes en consumo de memoria que la webconsole disponible en los repositorios oficiales de Karaf.

Antes de analizar el rendimiento, es necesario tener en mente las características de la memoria de la máquina virtual de Java. La JVM utiliza el Java Heap Space para crear nuevos objetos. Estos objetos son destruidos automáticamente por el Garbage Collector (GC) cada vez que las referencias a ellos se destruyen (es decir, cuando no pueden ser accesibles). Además, el GC puede separar de forma lógica diferentes espacios de memoria, con el fin de identificar más rápidamente los objetos que se pueden destruir. Cuando la máquina virtual agota la memoria que tiene asignada para el Heap, se produce una excepción.

[37] http://web.dit.upm.es/vnxwiki/index.php/Main_Page

Para intentar evaluar el rendimiento de Karaf Web Client en comparación con webconsole, se ha utilizado el cliente jconsole, ya que dispone de herramientas gráficas para la gestión de la memoria utilizada utilizando MBeans.

Para realizar la prueba, se ha arrancado una instancia de Karaf con webconsole y con Karaf Web Client, tras lo cual se ha esperado un tiempo de 10 minutos y se ha ejecutado manualmente el GC. Esto hace que los objetos Java creados durante el arranque de la instancia pero que ya no se utilicen (porque sus referencias hayan sido borradas) se hayan podido liberar y la memoria que ocupan esté disponible. Cada una de las instancias Karaf tiene asignada un máximo de 512 MB de memoria.

En primer lugar, se ha abierto en el navegador web la página de acceso a la webconsole de Karaf, y a continuación se ha comprobado la cantidad de memoria disponible en el Heap Space. La Figura 20 ilustra este momento.

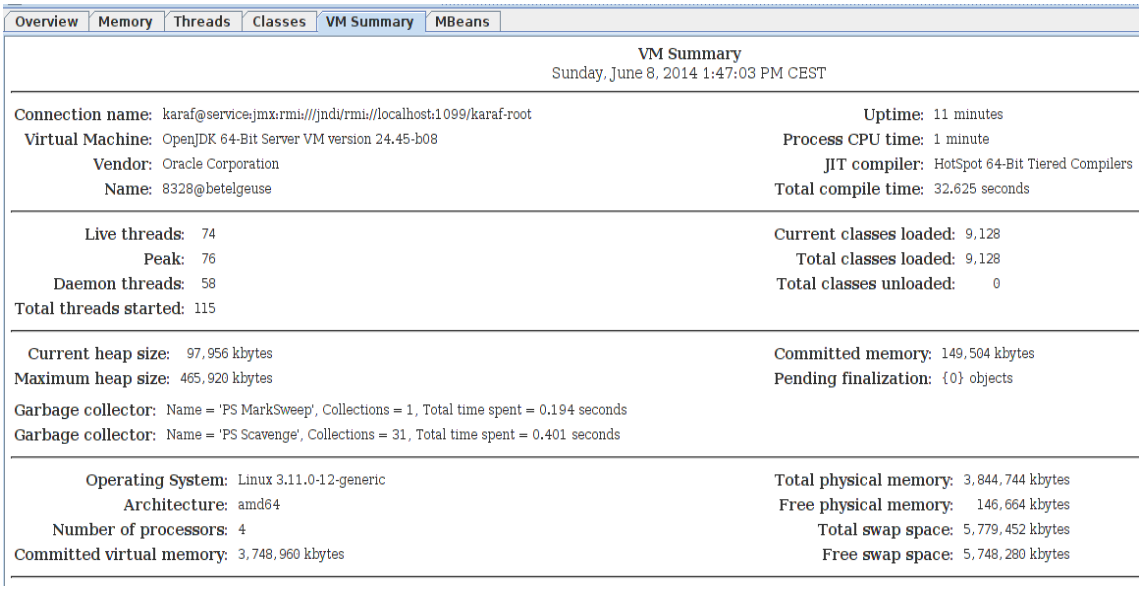

Figura 20. Máquina virtual al inicio.

En este momento, el dato que más interesa es el de Free physical memory, ya que siempre debe haber memoria disponible para la ejecución normal de los diferentes módulos. La cantidad de memoria libre en este momento es de alrededor de 145 MB. Para evaluar el consumo de memoria de la webconsole, se ha hecho la misma prueba pero cerrando ésta y esperando 5 minutos. El resultado se ilustra en la Figura 21:

VM Summary
Sunday, June 8, 2014 1:52:23 PM CEST

Connection name:	karaf@service:jmx:rmi:///jndi/rmi://localhost:1099/karaf-root	Uptime:	16 minutes
Virtual Machine:	OpenJDK 64-Bit Server VM version 24.45-b08	Process CPU time:	1 minute
Vendor:	Oracle Corporation	JIT compiler:	HotSpot 64-Bit Tiered Compilers
Name:	8328@betelgeuse	Total compile time:	33.293 seconds
Live threads:	75	Current classes loaded:	9,157
Peak:	77	Total classes loaded:	9,157
Daemon threads:	58	Total classes unloaded:	0
Total threads started:	121		
Current heap size:	75,792 kbytes	Committed memory:	127,488 kbytes
Maximum heap size:	465,920 kbytes	Pending finalization:	{0} objects
Garbage collector:	Name = 'PS MarkSweep', Collections = 1, Total time spent = 0.194 seconds		
Garbage collector:	Name = 'PS Scavenge', Collections = 49, Total time spent = 0.514 seconds		
Operating System:	Linux 3.11.0-12-generic	Total physical memory:	3,844,744 kbytes
Architecture:	amd64	Free physical memory:	187,320 kbytes
Number of processors:	4	Total swap space:	5,779,452 kbytes
Committed virtual memory:	3,749,988 kbytes	Free swap space:	5,748,280 kbytes

Figura 21. Máquina virtual sin webconsole.

En este momento, se ha incrementado la memoria disponible en alrededor de 30 MB. Una vez comprobado este dato, se ha procedido a arrancar el Central Server para consultar el estado de las instancias. Recordemos que el código JavaScript consulta cada 5 segundos el estado de las instancias, por lo que el número de threads en ejecución se va a incrementar. Estos hilos son los encargados de recibir y procesar las peticiones que llegan al servidor de Karaf. La Figura 22 ilustra la situación de la máquina virtual en el momento en el que se pone en marcha el Central Server.

VM Summary
Sunday, June 8, 2014 2:11:27 PM CEST

Connection name:	karaf@service:jmx:rmi:///jndi/rmi://localhost:1099/karaf-root	Uptime:	35 minutes
Virtual Machine:	OpenJDK 64-Bit Server VM version 24.45-b08	Process CPU time:	1 minute
Vendor:	Oracle Corporation	JIT compiler:	HotSpot 64-Bit Tiered Compilers
Name:	8328@betelgeuse	Total compile time:	36.894 seconds
Live threads:	85	Current classes loaded:	9,278
Peak:	85	Total classes loaded:	9,278
Daemon threads:	62	Total classes unloaded:	0
Total threads started:	144		
Current heap size:	82,814 kbytes	Committed memory:	136,704 kbytes
Maximum heap size:	465,920 kbytes	Pending finalization:	{0} objects
Garbage collector:	Name = 'PS MarkSweep', Collections = 1, Total time spent = 0.194 seconds		
Garbage collector:	Name = 'PS Scavenge', Collections = 167, Total time spent = 1.230 seconds		
Operating System:	Linux 3.11.0-12-generic	Total physical memory:	3,844,744 kbytes
Architecture:	amd64	Free physical memory:	151,708 kbytes
Number of processors:	4	Total swap space:	5,779,452 kbytes
Committed virtual memory:	3,033,824 kbytes	Free swap space:	5,712,276 kbytes

Figura 22. Máquina virtual sólo con Karaf Web Client.

Como se puede observar, el número de threads en ejecución (Live threads) se ha incrementado considerablemente. Esto es así porque el Central Server está preguntando cada pocos segundos a la instancia por su estado, lo cual hace que los hilos del servidor Jetty estén activos recibiendo las peticiones. De hecho, en la Figura

23se aprecia la variación en el número de threads en el momento en el que el Central Server comienza a realizar peticiones.

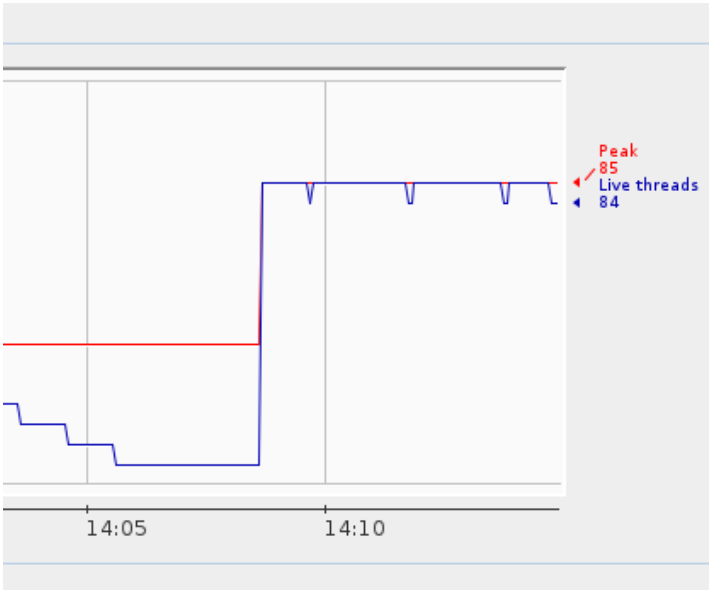

Figura 23. Variación en los threads producida por el Central Server.

Aunque el número de threads en ejecución ha aumentado, se puede observar cómo la memoria disponible ha permanecido prácticamente constante. Estas cifras permanecen en este rango a lo largo del tiempo, por lo que no se puede afirmar que la utilización de Karaf Web Client produzca una mejora significativa en la utilización de memoria de la máquina virtual con respecto a webconsole.

En el apartado 5.2 se ha mencionado la utilización de una dependencia de Maven para crear un filtro Cross-origin Resource Sharing (CORS)[38]. Uno de los mecanismos de seguridad que poseen los navegadores web es la prohibición de hacer peticiones a un dominio diferente del que se origina el recurso que realiza dichas peticiones. Por ejemplo, si un cliente se descarga un script desde un servidor (con un dominio http://dominio1.es), y ese script intenta acceder a un recurso en otro dominio (por ejemplo http://dominio2.es), el navegador impedirá dicho acceso, basándose en que el contenido del segundo dominio podría ser perjudicial para el host cliente. Esta política de seguridad se denomina same origin security policy. Con el objetivo de aceptar determinadas excepciones a dicha regla, se introdujo el mecanismo CORS que mediante la utilización de determinadas cabeceras HTTP, un servidor puede especificar qué dominios están permitidos para el acceso desde los scripts de dicho servidor.

Puesto que en un entorno real, el script centralServer.js hace llamadas a diferentes direcciones IP (y por lo tanto, diferentes dominios), es necesario utilizar CORS para permitir dichas llamadas. El servidor Apache Tomcat proporciona una implementación

[38] http://www.w3.org/TR/cors/

del filtro CORS[39] que especifica, entre otros parámetros, los dominios que se permiten en una redirección o los métodos HTTP que se pueden emplear. El siguiente código especifica el filtro CORS por defecto que se ha utilizado para permitir que el navegador acceda a las diferentes instancias. Este código se especificaría en el archivo web.xml del Central Server.

```
<filter>
<filter-name>CorsFilter</filter-name>
<filter-class>org.apache.catalina.filters.CorsFilter</filter-class>
<init-param>
            <param-name>cors.allowed.origins</param-name>
            <param-value>*</param-value>
</init-param>
<init-param>
            <param-name>cors.allowed.methods</param-name>
            <param-value>GET,POST,HEAD,OPTIONS,PUT</param-value>
</init-param>
<init-param>
            <param-name>cors.allowed.headers</param-name><param-
value>Content-Type,X-Requested-With,accept,Origin,Access-Control-
Request-Method,Access-Control-Request-Headers,Authorization</param-
value>
</init-param>
<init-param>
            <param-name>cors.exposed.headers</param-name><param-
value>Access-Control-Allow-Origin,Access-Control-Allow-
Credentials</param-value>
</init-param>
<init-param>
      <param-name>cors.support.credentials</param-name>
      <param-value>true</param-value>
</init-param>
<init-param>
      <param-name>cors.preflight.maxage</param-name>
      <param-value>10</param-value>
</init-param>
</filter>
<filter-mapping>
<filter-name>CorsFilter</filter-name>
<url-pattern>/*</url-pattern>
</filter-mapping>
```

[39] http://tomcat.apache.org/tomcat-7.0-doc/config/filter.html#CORS_Filter

6.3 Creación de bundles de prueba

Como se ha mencionado en la sección 3.1, el gestor de proyectos Maven facilita la creación de bundles a partir de un proyecto Java. Para generar diferentes bundles de prueba y poder comprobar que son instalados y ejecutados, se ha utilizado un plugin específico de Maven (Maven Bundle Plugin)[40]. La configuración mínima de este plugin para construir un bundle es la siguiente:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>
<groupId>BundlePrueba</groupId>
<artifactId>BundlePrueba</artifactId>  (1)
<version>0.0.1-SNAPSHOT</version>
<name>BundlePrueba</name>
<packaging>bundle</packaging>   (2)

<dependencies>
      <dependency>
      <groupId>org.osgi</groupId>
      <artifactId>org.osgi.core</artifactId>
      <version>4.3.0</version>
</dependency>

<dependency>   (3)
      <groupId>org.apache.felix</groupId>
      <artifactId>maven-bundle-plugin</artifactId>
      <version>2.4.0</version>
</dependency>

</dependencies>
<build>
      <plugins>
      <plugin>  (4)
      <groupId>org.apache.felix</groupId>
      <artifactId>maven-bundle-plugin</artifactId>
      <version>2.4.0</version>
      <extensions>true</extensions>
      <configuration>
      <instructions>
                        <Export-Package>bundleprueba</Export-Package>
(5)
```

[40] http://felix.apache.org/site/apache-felix-maven-bundle-plugin-bnd.html

```
                <Bundle-Activator>bundlehola.Activator</Bundle-
Activator>(6)
      </instructions>
      </configuration>
      </plugin>
      </plugins>
      </build>
</project>
```

En primer lugar, es necesario especificar el nombre, la versión o el identificador del bundle (1), seguido del tipo de artefacto generado, en este caso un bundle (2). A continuación hay que especificar la dependencia de ese proyecto en el propio plugin (3). Finalmente, se indica la configuración del plugin (4), especificando el paquete que se está exportando (5) y la localización de la clase Activator (6) dentro del proyecto.

6.4 Alternativas

Existen diferentes componentes que pueden ser ejecutados sobre un entorno Karaf que permiten obtener una funcionalidad similar a la desarrollada en este trabajo. En primer lugar, se describirá el conector de JMX Jolokia y a continuación la consola web Hawtio, que hace uso del primero para construir una interfaz web desde la cual controlar diversos aspectos de una máquina virtual de Java.

6.4.1 Jolokia

Jolokia[41] es un conector que, al ser ejecutado en una máquina virtual de Java (o entorno OSGi) con JMX, permite la invocación de MBeans a través de HTTP, pudiendo además utilizar el formato JSON para el envío de parámetros. Es decir, en esencia ofrece el mismo tipo de funcionalidad que el Karaf Web Client desarrollado, ya que permite el control de los MBeans sin tener que utilizar directamente RMI como protocolo de acceso, sino que éstos se acceden utilizando HTTP. La Figura 24 ilustra la estructura de Jolokia.

[41] http://www.jolokia.org/

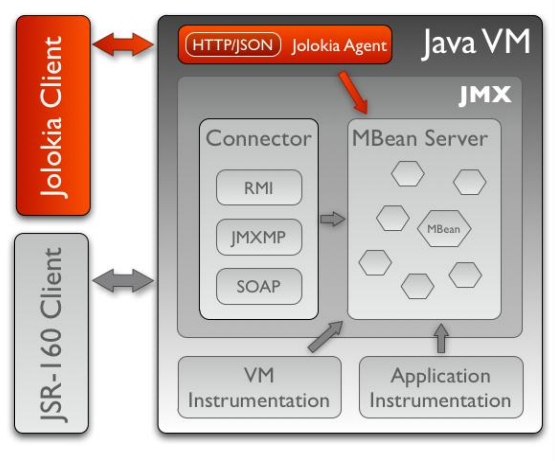

Figura 24. Estructura de Jolokia.

[Fuente: http://www.jolokia.org/reference/html/architecture.html]

Existen diferentes implementaciones de Jolokia (llamadas agents) preparadas para ser desplegadas en diferentes entornos, tales como un servidor JEE, un contenedor de OSGi o directamente sobre la máquina virtual de Java, además de tener un cliente desarrollado en JavaScript.

6.4.2 Hawtio

Hawtio[42] es una consola web modular para la máquina virtual de Java que permite el control de multitud de elementos tales como JMX, entorno OSGi en ejecución o servidores como Tomcat o Jetty entre otras características. Se puede instalar y desplegar en una instancia Karaf. La Figura 25 ilustra una captura de pantalla de Hawtio en un navegador web.

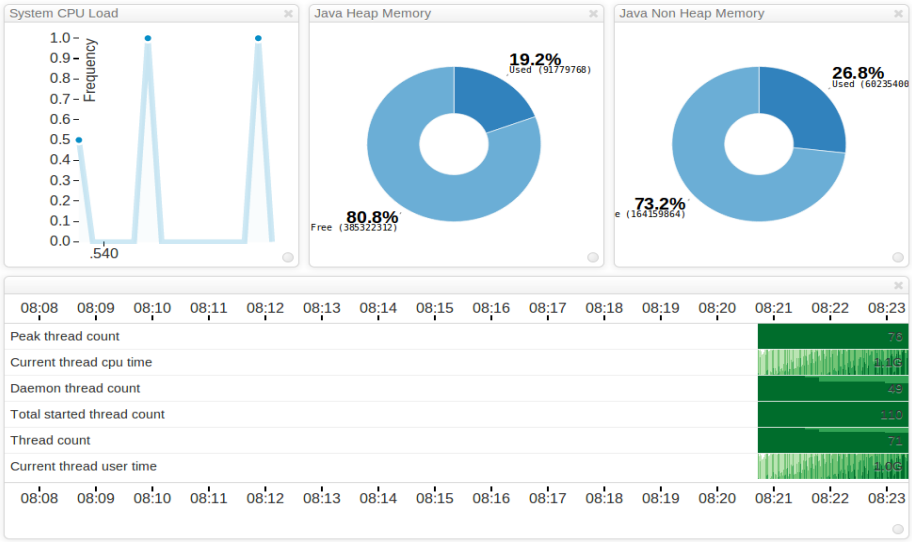

Figura 25. Hawtio.

[42] http://hawt.io/

En resumen, la combinación de Jolokia y Hawtio aporta funcionalidades similares a las que se consiguen con la webconsole de Karaf. Jolokia permite utilizar directamente el formato JSON para los mensajes, mientras que Hawtio proporciona unas funcionalidades más completas que las que se consiguen con webconsole. En cualquier caso, Hawtio no permite la visualización de múltiples instancias en diferentes máquinas.

7 Conclusiones y trabajos futuros

El planteamiento de este Trabajo Fin de Máster se debe a la necesidad de incorporar un mecanismo de control de bundles para la pasarela BatMP desarrollada en el CeDInt, además de permitir el control centralizado de la ejecución e instalación de dichos bundles en varias pasarelas remotas.Siguiendo estos objetivos, se ha realizado una búsqueda de diferentes contenedores OSGi sobre el que instalar la pasarela BatMP y que facilite realizar determinadas acciones de control sobre los repositorios y bundles del entorno OSGi que contiene. Estas acciones están encaminadas a especificar de forma remota las direcciones de los repositorios desde donde es posible descargar nuevos bundles y ponerlos en ejecución dinámicamente. Mediante este objetivo se pretende simplificar el proceso de configuración de la pasarela BatMP (principalmente para poder poner en ejecución drivers de diferentes tecnologías de control domótico). En este sentido, se han cumplido los principales objetivos del desarrollo propuesto, ya que no sólo es posible realizar un control dinámico de componentes OSGi, sino que además se ha adquirido una experiencia en los desarrollos sobre un contenedor como Apache Karaf que serán de utilidad a la hora de integrar la pasarela BatMP utilizando dicha tecnología.

Con respecto al proceso de desarrollo de este proyecto,enprimer lugar, ha sido necesario estudiar qué contenedores OSGi existen y de qué manera éstos pueden ser de utilidad para el desarrollo planteado. Debido al desconocimiento de Apache Karaf, la utilización de este contenedor no estaba planteada originalmente, y ha sido durante el desarrollo del trabajo cuando se ha llegado a la decisión de utilizarlo para el desarrollo. Las ventajas que ha aportado este entorno al desarrollo han sido varias. En primer lugar, la propia implementación de un OBR así como las diferentes features de las que dispone el entorno (agente JMX, servidor web Jetty, sistema de logging, etc.) ha resultado de utilidad a la hora realizar los desarrollos. En segundo lugar, la posibilidad de desplegar automáticamente en el entorno archivos .WAR ha facilitado el despliegue del Central Server. Finalmente, es necesario destacar las diferentes posibilidades de configuración de las que dispone Karaf así como la documentación disponible que, si bien no es muy extensa, es bastante concisa.

Con respecto a las dificultades encontradas, en primer lugar, el propio desconocimiento de los contenedores OSGi como Karaf o ACE, así como de otras tecnologías Java como por ejemplo JMX o Blueprint ha requerido bastante tiempo de documentación. En segundo lugar, la propia instalación de Karaf y la configuración de las diferentes instancias del escenario de pruebas, así como la configuración de la máquina virtual VNX ha supuesto también un tiempo de desarrollo. Además, aunque existe un plugin específico de Eclipse para generar distribuciones Karaf o features, el Eclipse Integration for Karaf (EIK), éste no se encuentra aún completamente operativo debido a la presencia de bugs que impiden su utilización, lo cual dificulta trabajar con este entorno. Finalmente, se ha invertido bastante tiempo en entender y resolver

diferentes problemas en el formato de los repositorios, así como en el acceso a los mismos.

7.1 Trabajos futuros

Los futuros pasos que se plantean son los siguientes:

- Integración de Karaf con BatMP
  El primer paso para probar el desarrollo realizado será la integración de los diferentes módulos de la pasarela BatMP utilizando Karaf. De esta forma se intentará sacar ventaja de las opciones que ofrece Karaf para la ejecución de la pasarela, como por ejemplo el control de los diferentes bundles sin depender de un script de configuración como ocurre ahora.

- Karaf Cellar
  Apache Cellar es una tecnología que permite manejar diferentes clusters de instancias Karaf, proporcionando sincronización entre las mismas. Con Cellar, cada instancia Karaf es un nodo que puede pertenecer a uno o más grupos de clusters, y donde cada nodo es descubierto automáticamente por otros nodos en su mismo grupo. De este modo, es posible distribuir en un grupo determinado de nodos, diferentes objetos tales como features, bundles o archivos de configuración, entre otros. Uno de los futuros pasos podría ser la utilización de Cellar para agrupar varias instancias Karaf y que éstas reaccionen de forma conjunta a un determinado evento, por ejemplo la presencia de una nueva versión de un bundle de BatMP.

- Integración con Artifactory
  Artifactory[43] es un gestor de repositorios Maven que permite, entre otras funcionalidades, actuar como proxy de repositorios remotos, almacenando los elementos de los diferentes repositorios para que éstos puedan ser accedidos localmente. La Figura 26 ilustra una instalación de Artifactory en una red local.

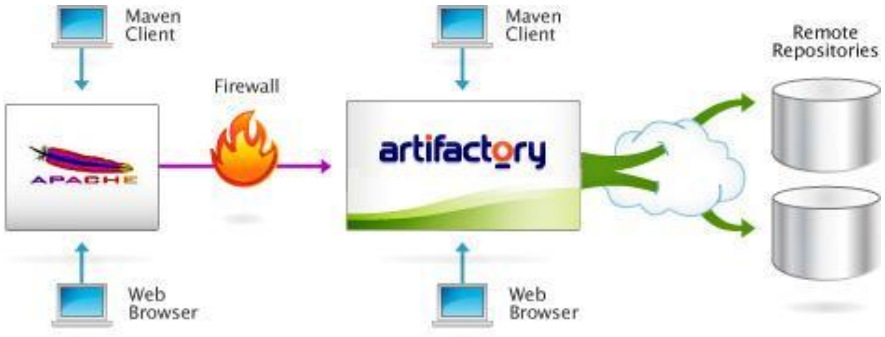

Figura 26. Instalación de Artifactory.

[43] http://www.jfrog.com/home/v_artifactory_opensource_overview

Uno de los futuros pasos a realizar será probar la integración de Karaf accediendo a Artifactory para descargar directamente los bundles, ya que en todos los desarrollos realizados hasta la fecha en el CeDInt se ha utilizado este gestor de repositorios.

- Ampliar funcionalidades
  En el cliente web las diferentes instancias se han creado *ad hoc* para las pruebas realizadas. En este sentido podría ampliarse la funcionalidad de Central Server para permitir dar de alta diferentes instancias. Por ejemplo, con esta funcionalidad se podría utilizar el cliente web para controlar el estado de los diferentes servidores de test de los que se dispone en CeDInt para el desarrollo (se dispone de un conjunto de máquinas virtuales Debian sobre las cuales se ejecutan diferentes configuraciones de la pasarela BatMP). Además, dada la variedad de opciones de control disponibles a través de JMX es posible ampliar las opciones de control sobre los bundles y que éstas no se limiten únicamente a los repositorios y bundles del entorno.

- Utilización de features de Karaf
  Como se ha comentado, una de las características más interesantes de Karaf son las features, que permiten agrupar en una unidad lógica un conjunto de bundles y archivos de configuración. Las tecnologías de control domótico de la pasarela BatMP (los drivers) están estructuradas como bundles OSGi con ciertos archivos de configuración (por ejemplo archivos que especifican los tiempos máximos de reenvío de mensajes, o las características de determinados dispositivos. En este sentido, dichos drivers pueden ser especificados como una feature, que contenga tanto los paquetes como los archivos de configuración asociados a dicho driver.

Bibliografía

[1] Apache Félix OBR Repository. http://felix.apache.org/site/apache-felix-osgi-bundle-repository.html
[2] Apache ACE. https://ace.apache.org/
[3] Apache Maven. http://maven.apache.org/
[4] Benjamin G. Sullins, Mark B. Whipple, "JMX in Action". Greenwich, Manning Ed, 2003.
[5] Richard S. Hall, Karl Pauls, Stuart McCulloch, David Savage, "OSGi in Action". Greenwich, Manning Ed, 2011.
[6] Apache Félix. http://felix.apache.org/
[7] Apache Sling. http://sling.apache.org/
[8] Pax Runner. https://ops4j1.jira.com/wiki/display/paxrunner/Pax+Runner
[9] Holly Cummings, Timothy Ward, "Enterprise OSGi in Action". Shelter Island, Manning Ed, 2013.
[10] Caffarel J., del Campo-Jimenez, G., Perandones, J.M.,Gomez-Otero, C.,Martinez, R.,Santamaria, A."Open multi-technology building Mangement System". 4th International Congress on Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT) 2012.
[11] GNU General Public License. http://www.gnu.org/copyleft/gpl.html