ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE
TELECOMUNICACIÓN
UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Trabajo Fin de Grado

Prueba de Plataformas para el Desarrollo de Aplicaciones de la Internet de las Cosas

AUTOR: José Luis Romero Gázquez
DIRECTORES: Juan Ángel Pastor Franco; Bárbara Álvarez Torres
TITULACIÓN: Grado en Ingeniería Telemática

Septiembre / 2015

Agradecimientos

A mis directores de proyecto, profesores y a Ramón Martínez por su incansable ayuda.

A mis compañeros de piso por los momentos compartidos.

Y sobre todo a mi familia y pareja por su aliento y confianza.

ÍNDICE

ÍNDICE DE FIGURAS

1. Motivación

1.1 Resumen del proyecto

La complejidad de las redes de sensores hace necesaria la utilización de plataformas de gestión de los dispositivos que formen parte de dicha estructura. Estas plataformas deben exhibir una serie de características, principalmente: (1) deben de permitir la interconexión de dispositivos heterogéneos, (2) deben ser escalables con el tamaño de la red, (3) deben permitir el registro y búsqueda de dispositivos y servicios, (4) deben ser tolerantes a fallos y (5) deben incluir utilidades para la estimación o interpolación de datos en aquellas áreas no cubiertas por los sensores [1].

Actualmente existen multitud de líneas de investigación y desarrollo abiertas con el objetivo de implementar eficazmente las características mencionadas anteriormente. Por este motivo, no existen estudios metódicos sobre utilidad y eficiencia de plataformas, ni éstas se encuentran implantadas comercialmente en su totalidad.

El presente proyecto de investigación es fruto del trabajo propuesto en una beca de colaboración con el departamento de Tecnologías de la Información y las Comunicaciones, cuyo **objetivo global** es la **prueba y evaluación de las plataformas de gestión de Internet de las Cosas actualmente disponibles, para su empleo en la agricultura de precisión**. Más concretamente, una de las actuaciones que se va a llevar a cabo para cumplir este objetivo es la definición de una **aplicación *software* genérica para la prueba de plataformas** de integración de Internet de las Cosas.

1.2 Objetivos iniciales y posteriores

El objetivo inicial de este trabajo fue la evaluación y comparación de diversas plataformas de Internet de las Cosas atendiendo a una serie de parámetros tales como facilidad de instalación, soporte, facilidad de desarrollo de aplicaciones, herramientas asociadas, estándares seguidos, rendimiento, etc. Para conseguir tal fin, se propuso desarrollar un *software* genérico (Test Bench) que permitiría evaluar los parámetros mencionados en cada una de las plataformas seleccionadas con el fin de comparar los resultados obtenidos de manera objetiva. Así, cuatro plataformas fueron elegidas para su estudio: CHOReOS, FIWARE, Nimbits y Amazon Web Services.

Tras iniciar los trabajos de investigación, se descartó el estudio de la plataforma CHOReOS [2] debido a la escasez y la mala calidad de la documentación proporcionada por la plataforma. Posteriormente, se ha desarrollado una aplicación que permite el envío de datos de M sensores virtuales a la plataforma Nimbits tal y como se describe en el Anexo I. de este documento. Sin embargo, las librerías que modelan el protocolo XMPP en Nimbits están en desuso por lo que no ha sido posible automatizar la implementación de la recepción de los datos de la platafoma. Por otro lado, los trabajos de investigación realizados acerca de la plataforma Amazon Web Services son descritos en el Anexo II. En este caso, se ha desarrollado una aplicación que permite el envío/recepción de los datos de M sensores virtuales a/desde la plataforma AWS. Sin embargo, no ha sido posible implementar la recepción de los datos sin utilizar la técnica de *polling* debido a que el servicio elegido no implementa el patrón de diseño descrito en el apartado 4 de esta memoria.

Por los motivos mencionados anteriormente, no ha sido posible la prueba y evaluación de las plataformas CHOReOS, Nimbits y Amazon Web Services. Como consecuencia, **el fin** de este trabajo de investigación pasa a ser **el estudio del rendimiento de la platafoma de gestión de Internet de las Cosas FI-WARE, para su empleo en la agricultura de precisión.**

Por este motivo, es necesario cumplir con los siguientes objetivos:

1. Estudio de los diferentes aspectos de los componentes que componen el capítulo *Data*/*Context Management* de FI-WARE.
2. Desarrollo de una aplicación *software* que permita la simulación del envío de información de una red de M sensores a la plataforma FI-WARE.
3. Desarrollo de un módulo de adquisición de los datos obtenidos a la salida de FI-WARE que permite la generación de un fichero de medidas.
4. Desarrollo de una aplicación de generación de resultados que genera medidas cuantitativas a partir del fichero de medidas creado en el punto anterior para poder evaluar el rendimiento de la plataforma.
5. Diseño de plan de pruebas que establezca diferentes escenarios de simulación de la red de sensores en función de parámetros tales como el número de sensores, el período de envío de datos, el *payload* y el número de nodos de la red.

1.3 Metodología

Para llevar a cabo los objetivos propuestos, se introducen los hitos del presente Trabajo Fin de Grado en orden cronológico:

1. Estudio de especificaciones técnicas de FI-WARE prestando especial atención al capítulo *Data*/*Context Management*.
2. Desplegar instancia GE *Context Broker* en la nube.
3. Desarrollo de una aplicación *software* inicial que permita transmitir datos a la instancia desplegada.
4. Persistencia de datos, configuración de inyector Cygnus. Creación cuenta de Cosmos e implementación de script para la descarga y eliminación de datos.
5. Configuración local de inyector Cygnus, modificación del código del inyector para incorporar módulo de adquisición de datos.
6. Desarrollo de la aplicación *software* definitiva, simulación con varios hilos.
7. Creación de servidor VPN, simulación con varios equipos.
8. Simulación de escenarios del protocolo de pruebas diseñado.
9. Desarrollo de una aplicación de generación de resultados cuantitativos.
10. Estudio de los resultados, conclusiones.

1.4 Organización de la memoria

El Trabajo Fin de Grado titulado "Prueba de Platafomas para el Desarrollo de Aplicaciones de la Internet de las Cosas" se estructura en los siguientes capítulos:

- **Capítulo 1:** resumen del proyecto realizado, se describen sus objetivos y se explica la metodología llevada a cabo para cumplir con el fin del trabajo de investigación.
- **Capítulo 2:** introduce los hitos históricos que dan lugar a la tecnología de Internet of Things (IoT). Además, presenta varias definiciones de este fenómeno así como los estándares utilizados por plataformas de gestión de dipositivos implementadas en el ámbito de la IoT. Finalmente, expone ejemplos de *hardware*/*software* disponibles en el mercado y de casos de uso desplegados con esta tecnología.
- **Capítulo 3:** describe el contexto de desarrollo del presente trabajo en el que se presentan las herramientas software y los lenguajes de programación que han sido utilizados durante los trabajos de investigación.
- **Capítulo 4:** explica las principales características y variaciones del patrón de diseño *Publish*/*Subscribe*.
- **Capítulo 5:** describe la plataforma FIWARE, específicamente los principales componentes del capítulo *Data*/*Context Management*: Context Broker, Cosmos y Cygnus.
- **Capítulo 6:** el propósito de este capítulo es describir los aspectos relativos al diseño, implementación, instalación, configuración, despliegue y simulación del prototipo del sistema estudiado en el presente Trabajo Fin de Grado.
- **Capítulo 7:** contiene tanto la descripción del protocolo de pruebas que ha condicionado las simulaciones de los prototipos estudiados en el capítulo 6, como la valoración de los resultados obtenidos.
- **Capítulo 8:** menciona posibles líneas futuras de investigación relacionadas con el presente trabajo.

Finalmente los Anexos I y II describen los trabajos realizados acerca de las plataformas Nimbits y Amazon así como los resultados obtenidos.

2 Introducción a Internet of Things (IoT)

2.1 Tres definiciones de IoT

Hay una gran cantidad de información acerca de Internet de las Cosas (en adelante IoT). Este fenómeno posee tres características que hacen muy difícil su definición que son: escasa madurez, rápido desarrollo y gran difusión. No obstante, en este apartado se citan tres posibles definiciones del mismo.

De acuerdo al IoT European Research Cluster [3], Internet de las Cosas (IdC, en inglés IoT) es: *"una infraestructura de red dinámica y global con capacidades de auto-configuración basada en protocolos de comunicación estándar y que operan entre sí donde las "cosas" físicas y virtuales tienen identidad, atributos físicos y personalidades virtuales utilizando interfaces inteligentes, siendo integradas a la perfección en la red de información".*

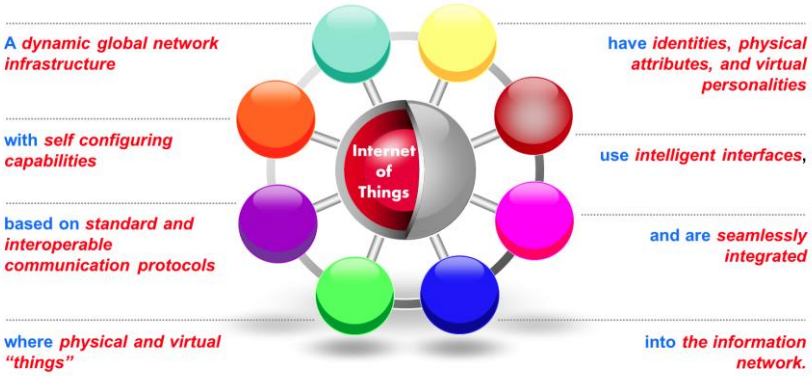


Figura 2.1. Definición de IoT

Otra definición extendida es: *"la IoT es una infraestructura global para la sociedad de la información, habilitando servicios avanzados al interconectar (física y virtualmente) cosas basadas en el mundo real con modernas tecnologías de la información y comunicaciones. A través de la identificación, captura de datos y capacidades de comunicación y procesado, la IoT permite un uso pleno de los datos para ofrecer servicios a aplicaciones, a la misma vez de dotar de los requisitos de seguridad y privacidad pertinentes. Desde una perspectiva más amplia, la IoT puede ser percibida como una nueva visión con implicaciones tanto tecnológicas como sociales".*

La IoT se basa en tres pilares básicos de conocimiento (Figura 2.2): el diseño de las cosas (hardware), el diseño de la infraestructura de comunicación (telemática) y la toma de decisiones (semántica y de razonamiento). Así, el fenómeno de la IoT se encontraría en la zona de intersección de estas tres áreas. La tercera definición proviene de una infografía [4], parcialmente mostrada en la Figura 2.3 que resume de una manera rápida y eficaz los conceptos básicos de esta tecnología: *"los Sistemas Inteligentes (Smart Systems) y la IoT son impulsados por una combinación de (1) sensores y actuadores, (2) conectividad y (3) personas y procesos".* De acuerdo con esta visión, los sensores y actuadores crean un sistema nervioso digital y, gracias a las conexiones, se habilita la disponibilidad de los datos que son transmitidos, almacenados, combinados y analizados entre sistemas que integran datos, personas, procesos y otros sistemas. Aunque menos formal, esta última es una definición simple y clara que aborda los tres principios fundamentales de la IoT: **las cosas, las conexiones y los procesos**.

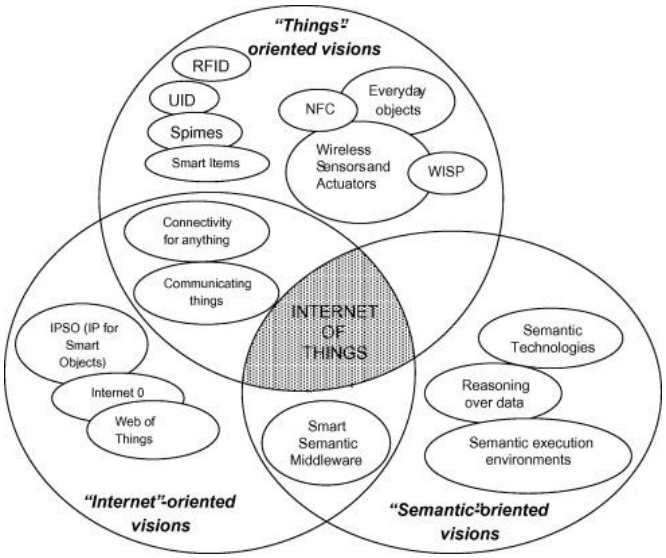


Figura 2.2. IoT como convergencia de diferentes visiones

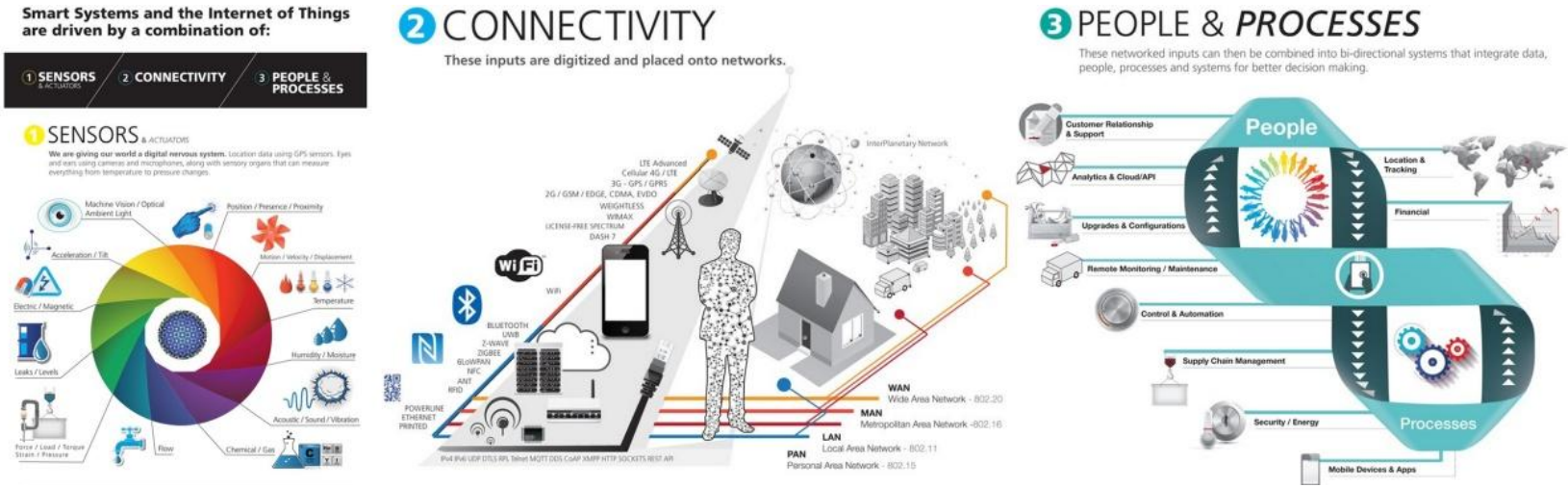


Figura 2.3 ¿Qué es la IoT?

2.2 Historia

El origen del Internet de las Cosas [5] data del año 1969 cuando la red ARPANET estableció comunicación entre las universidades de Stanford y Ucla, permitiendo no sólo el desarrollo de la IoT, sino de Internet propiamente dicho, la red de redes que actualmente conocemos y utilizamos diariamente.

En el año 1990 Jhon Romkey y Simon Hacket desarrollaron el primer objeto con conexión a Internet, la primera "cosa" con capacidad de manejo remoto. Esa "cosa" fue una tostadora inteligente que podía controlarse remotamente a través de cualquier ordenador, pudiendo encenderla, apagarla o incluso controlar el tiempo de tostado.

La primera aparición de la IoT como hoy se conoce procede del Instituto Tecnológico de Massachusetts (MIT), en concreto del trabajo del Auto-ID Center. Este grupo, fundado en 1999, realizaba investigaciones en el campo de la identificación por radiofrecuencia en red (RFID) y las tecnologías de sensores emergentes. Fue ahí donde se empezó a formar el actual concepto que tenemos de la IoT.

No obstante, tuvieron que pasar 10 años hasta que el británico Kevin Ashton, cofundador del Auto-ID Center nombrado anteriormente, acuñara por primera vez el nombre de Internet de las Cosas gracias a un artículo publicado en el RFID Journal en Julio de 2009. En dicho artículo Kevin introdujo el concepto de conectar todas las cosas que nos rodean con la finalidad de poder contarlas, posicionarlas, conocer su estado en cualquier momento así como aportarnos información sobre el entorno que les rodea.

Atendiendo a la Figura 2.4, se puede observar el crecimiento que ha seguido la cantidad de dispositivos conectados a Internet. En el año 2003, había aproximadamente 6,3 mil millones de personas en el mundo, con un número aproximado de 500 millones de dispositivos conectados a Internet. La relación entre estas variables era de 0,08 dispositivos por persona. El auge en el desarrollo y la mejora de las tecnologías portátiles e inalámbricas como smartphones o tablets elevó a 12,5 mil millones en 2010 la cantidad de dispositivos conectados a la red de redes, pasando la relación dispositivos/personas de 0,08 a 1,84, es decir, a partir de este año el número de dispositivos supera al de personas. Las predicciones apuntan a que en el año 2020 cerca de 50.000 millones de dispositivos electrónicos estarán conectados a Internet, cambiando radicalmente nuestro estilo de vida.

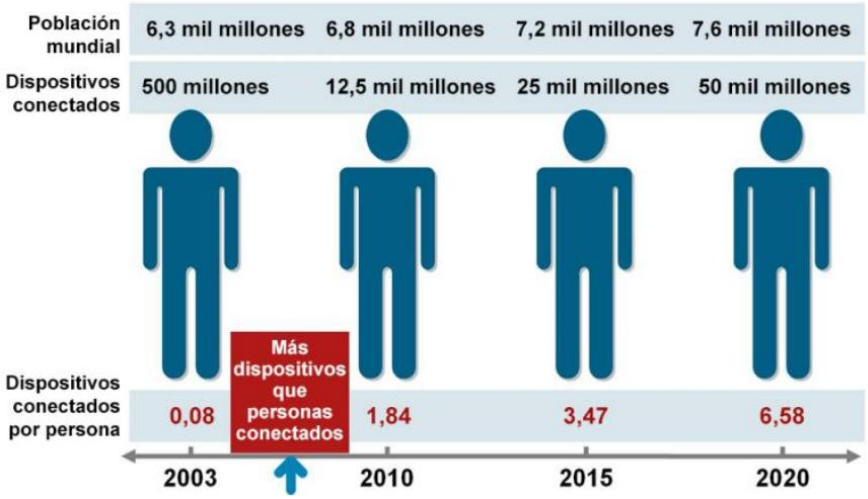


Figura 2.4. Evolución de los dispositivos conectados a Internet

2.3 Protocolos principales

Existe una amplia variedad de protocolos utilizados en el ámbito de Internet de las Cosas. Así, cada plataforma de IoT implementa una serie de protocolos u otros en función de los servicios que ofrece a los usuarios. Los protocolos más destacados son los siguientes [6]:

- **MQTT**: habilita el modelo *Publish*/*Subscribe* de una manera ligera. Permite la comunicación con servidores.
- **XMPP**: tecnología abierta para comunicación en tiempo real. Es el mejor protocolo para conectar dispositivos con personas. Se trata de un caso especial del patrón D2S (Devices to Servers).
- **DDS**: el primer estándar abierto e internacional que acoge el patrón *Publish*/*Subscribe* para comunicaciones en tiempo real de dispositivos embebidos. Se puede ver como un bus de datos que integra y comunica máquinas inteligentes.
- **AMQP**: sistema de colas diseñado para la conexión entre servidores. Sus características fundamentales son el encolamiento, enrutamiento, seguridad y fiabilidad.
- **IPv6**: versión del protocolo Internet Protocol y diseñada para reemplazar a IPv4, otorgando un mayor abanico de direcciones para la interconexión de dispositivos a través de Internet.
- **6LoWPAN**: acrónimo de *IPv6 over Low power Wireless Personal Area Networks*. Es una adaptación de IPv6 sobre IEEE802.15.4. Este protocolo opera solo en la banda de 2.4GHz de frecuencia.
- **UDP**: protocolo básico de la capa de transporte del modelo OSI usado en aplicaciones cliente/servidor basado en el protocolo IP. UDP es la principal alternativa a TCP y uno de los protocolos de red más antiguos, introducido en 1980. Se utiliza fundamentalmente para aplicaciones en tiempo real.
- **DTLS**: este protocolo provee de privacidad y seguridad en la comunicación a los protocolos situados en la capa de transporte, especialmente a los llamados *Datagram Protocols* como UDP.
- **CoAP**: protocolo de nivel de aplicación cuyo uso se da en dispositivos de Internet con recursos limitados, como nodos WSN. CoAP está diseñado para una fácil traducción a HTTP con el fin de simplificar la integración con la web, y al mismo tiempo satisfacer necesidades especializadas como soporte multicast, bajo coste y simplicidad.

Las Figuras 5 y 6 ilustran diferentes clasificaciones de los protocolos descritos.

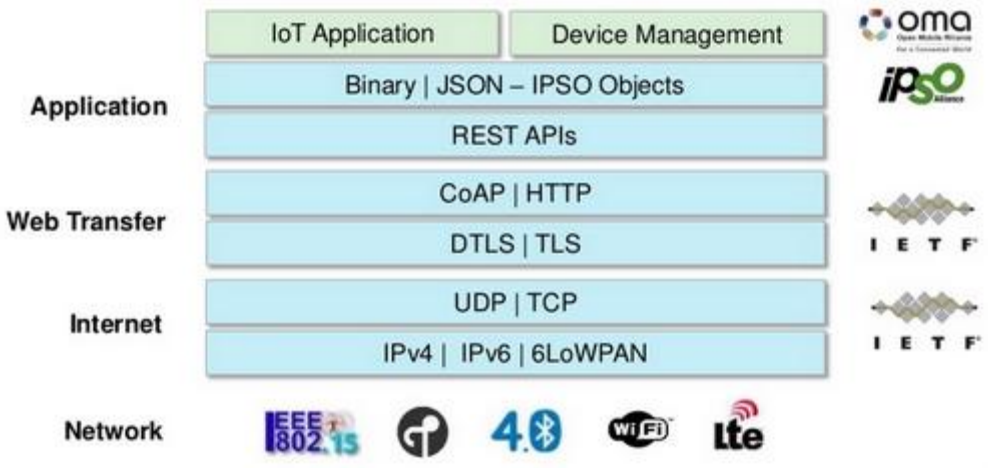


Figura 2.5. Protocolos principales IoT.

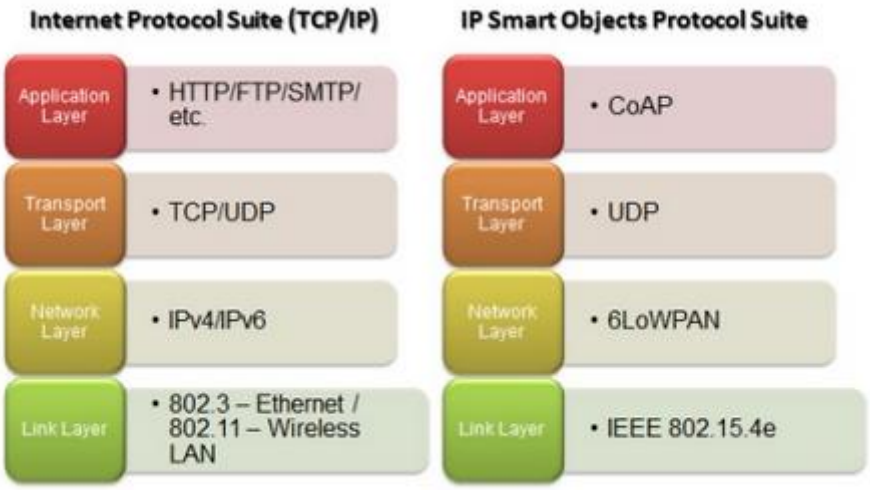


Figura 2.6. Pilas de protocolos TCP/IP y de objetos IP inteligente.

2.4 Ejemplos hardware y software

2.4.1 Hardware[1]

- **Arduino Uno**: Arduino es una plataforma de prototipos electrónicos de código abierto basado en hardware y software flexible y de fácil uso. Está destinado a cualquier persona interesada en crear entornos y objetos interactivos.

Figura 2.7. Arduino Uno

- **PanStamps**: pequeño módulo inalámbrico programable para IDE Arduino. Cada módulo contiene un MCU Atmega328p y una interfaz RF que provee de la conectividad y potencia necesaria para crear un dispositivo autónomo de conectividad inalámbrica.

Figura 2.8. PanStamp

- **Raspberry Pi**: es un ordenador de placa reducida o placa única desarrollado en UK por la Fundación Raspberry Pi. La Raspberry Pi tiene un tamaño de una tarjeta de crédito con posibilidad de conectarlo a la TV, teclados, ratones y otros periféricos.

Figura 2.9. Raspberry Pi

[1] Información extraída de [7].

- **Libelium Waspmote**: Waspmote es una plataforma de sensores inalámbricos de código abierto especialmente enfocada a la implementación de instancias de bajo consumo que permita a los nodos de sensores ser completamente autónomos, ofreciendo un tiempo de vida variable entre 1 y 5 años dependiendo de varios factores.

Figura 2.10. Waspmote

2.4.2 Software[2]

- **RIOT**: RIOT OS es un sistema operativo para dispositivos IoT. Está basado en un microkernel y diseñado para satisfacer aspectos como la eficiencia energética, desarrollo hardware independiente y un alto grado de modularidad.

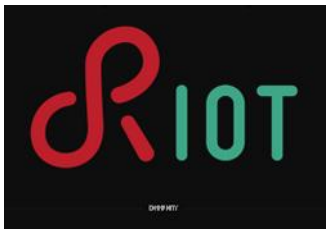


Figura 2.11. RIOT OS

- **OpenAlerts**: *software* de código libre y gratuito para el control y la monitorización de sensores a través de redes IP. Con openAlerts se pueden recibir alertas mediante correo electrónico y mensaje de texto así como lanzar comandos de control basándose en las condiciones del sensor.

Figura 2.12. OpenAlerts

- **Thingspeak**: aplicación IoT de código abierto con API para el almacenamiento y recuperación de datos de las *cosas* usando HTTP sobre Internet o a través de LAN. Con ThingSpeak se pueden crear aplicaciones de monitorización de sensores y redes sociales de *cosas* con actualización de estado.

Figura 2.13. ThingSpeak

[2] Información extraída de [8].

2.5 *Ejemplos de uso*

Figura 2.14. Ejemplos de uso IoT.

En la Figura 2.14 [9] se muestra un reducido número de la gran cantidad y variedad de usos actuales en los que la IoT está implicada. La Figura está estructurada en cuatro grupos o temáticas diferentes, como son aplicaciones para el cuerpo humano, el hogar, la ciudad y la industria.
En el ámbito del cuerpo humano, fundamentalmente se encuentran aplicaciones destinadas a la monitorización de las constantes vitales en cada momento. Un ejemplo de ello es la primera imagen, en la que se integra en el atuendo del bebé diferentes sensores que monitorizan su respiración, su temperatura corporal o la posición de su cuerpo. De la misma forma, actualmente en el mercado hay numerosos dispositivos *wearables* como pulseras deportivas que monitorizan las constantes vitales, calculan las calorías quemadas, pasos dados a lo largo del día así como los kilómetros que han sido realizados.

En cuanto a las aplicaciones del hogar, existen dispositivos como WeMo, de la compañía Belkin. Un dispositivo que hace de mediador entre la red eléctrica y los electrodomésticos y que permite, a través de una aplicación instalada en un smartphone, controlar el apagado y encendido de los electrodomésticos conectados. Otro *gadget* bastante útil actualmente en el mercado es el CobraTag, un llavero inteligente que se comunica con el smartphone a través de una aplicación que lo hace sonar. Esto permite encontrar de manera rápida y sencilla los objetos que estén cerca de este dispositivo tales como llaves o mochilas.

Cada vez se hace más fuerte el concepto de *Smart Cities* en nuestro entorno, siendo la IoT la gran culpable de este fenómeno. Tecnologías como las que se muestran son algunos ejemplos de ello, tales como sensores en las aceras que permiten conocer dónde hay aparcamiento libre sin necesidad de buscar o farolas inteligentes que regulan su iluminación en función de la luz natural y la estación del año, suponiendo un gran ahorro energético.

Por último, en el ámbito de la industria, se integran pequeños sensores en las máquinas que permiten localizar y notificar posibles errores de funcionamiento o averías en las mismas, siendo el servicio de mantenimiento automáticamente notificado, lo que favorece la productividad del proceso tanto de reparación como de producción al verse reducida la relación detección/reparación. La última aplicación que se indica es la utilización de sensores en la agricultura que permitan monitorizar y enviar datos de la cosecha tales como el nivel de humedad, abono o programación del riego, aplicación que precisamente se pretende estudiar en el presente trabajo de investigación a través de la plataforma FI-WARE.

2.6 *Plataformas IoT*

De la misma forma que hay numerosas aplicaciones basadas en la IoT, existen multitud de plataformas que permiten que el desarrollo de aplicaciones sea más amigable. A modo de introducción, se mencionan las características generales de cuatro plataformas de IoT.

- **Axeda Platform** [10]: plataforma con total integración de datos para aplicaciones IoT y M2M con una infraestructura basada en la nube. Más de 150 compañías utilizan Axeda para conectar y manejar sus productos con la nube con una completa seguridad, escalabilidad y flexibilidad.

Figura 2.15. Axeda

- **FI-WARE**: plataforma middleware Open-Source, financiada por la Unión Europea, para el desarrollo y despliegue de aplicaciones IoT. Ofrece un potente conjunto de servicios que facilita el desarrollo de aplicaciones inteligentes en múltiples sectores.

Figura 2.16. FI-WARE

- **Nimbits Platform**: Nimbits se define como una plataforma para conectar a personas, sensores y *software* con la nube y entre ellas. Se trata de una colección de componentes *software* diseñados para registrar series de datos en el tiempo, como pueden ser cambios de temperatura leídos por un sensor. En base a ellos, varios eventos pueden ser lanzados, como cálculos o alertas.

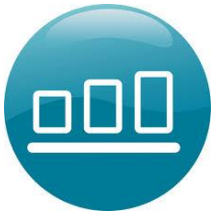


Figura 2.17. Nimbits

- **Amazon Web Services** [11]: colección de servicios de computación que en conjunto forman una plataforma de computación en la nube, ofrecida a través de Internet por Amazon.com. Es usado en aplicaciones populares como *Dropbox* o *Foursquare*.

Figura 2.18. Amazon Web Services

3 Contexto de desarrollo

3.1 Herramientas Software

3.1.1 Eclipse

La fundación Eclipse es una comunidad para particulares y organizaciones que desean colaborar con *software* abierto multiplataforma. El proyecto está enfocado en la creación de una plataforma abierta de desarrollo comprometida con *frameworks*, herramientas y tiempos de ejecución para crear, desarrollar y administrar el software en todo su ciclo de vida. Esta plataforma, típicamente ha sido utilizada para desarrollar entornos de desarrollo integrados (IDE), como el IDE de Java llamado Java Development Toolkit (JDT) y el compilador (ECJ) que se entrega como parte de Eclipse.

El proyecto Eclipse fue originalmente creado por IBM en Noviembre de 2001 y respaldado por un consorcio de proveedores *software*. La fundación Eclipse como tal fue creada en Enero de 2004 como una corporación independiente sin ánimo de lucro para actuar como administradora de la comunidad Eclipse. Esta corporación fue creada para permitir una comunidad abierta, transparente y ligada a Eclipse. Actualmente, la comunidad Eclipse consiste en un gran conjunto de particulares y organizaciones ligadas a la industria del *software* [12].

El *software* de Eclipse se compone por un conjunto de herramientas de programación de código abierto multiplataforma para desarrollar lo que denomina "Aplicaciones de Cliente Enriquecido". La arquitectura de esta plataforma la forman los siguientes componentes:

- **Plataforma principal**: inicio de Eclipse, ejecución de plugins.
- **OSGi**: plataforma para *bundling* estándar.
- **Standard Widget Toolkit (SWT)**: *widget toolkit* portable.
- **JFace**: manejo de archivos/texto y editores de texto.
- **Workbench de Eclipse:** vistas, editores, perspectivas y asistentes.
- **Analizador sintáctico**: compilación en tiempo real.

El entorno de desarrollo integrado (IDE) de Eclipse emplea módulos (*plug-in*) para proporcionar toda su funcionalidad al frente de la plataforma de cliente enriquecido. Esto permite a Eclipse extenderse usando otros lenguajes de programación como son C/C++ y Python [13].

3.1.2 PuTTY

PuTTY es un cliente SSH, Telnet, rlogin y TCP *raw*, desarrollado originalmente por Simon Tatham para plataformas Windows. Consiste en *software* libre disponible en código abierto desarrollado y respaldado por un conjunto de personas que trabajan de manera voluntaria [14]. Aunque originalmente fue creado de manera exclusiva para Windows, actualmente también está disponible en varias plataformas Unix, así como Mac OS clásico y Mac OS X.

El nombre **PuTTY** proviene de las siglas **Pu**: Port unique **TTY**: terminal type. Su traducción al español es: Puerto único de tipo terminal. Las características de PuTTY son [15]:

- Almacenamiento de hosts y preferencias para uso posterior.
- Control sobre la clave de cifrado SSH y la versión de protocolo.
- Clientes de línea de comandos SCP y SFTP, llamados "pscp" y "psftp" respectivamente.
- Control sobre el redireccionamiento de puertos con SSH.
- Emuladores completos de terminal xterm, VT102 y ECMA-48.
- Soporte IPv6.
- Soporte 3DES, AES, RC4 Blowfish, DES.
- Soporte de autenticación de clave pública.
- Soporte para conexiones de puerto serie local.

3.1.3 WinSCP

WinSCP es un cliente SFTP gráfico para Windows que emplea los protocolos SSH y SCP. Básicamente, se trata de una aplicación de *software* libre que tiene como función principal facilitar la transferencia segura de archivos entre dos sistemas: uno local y otro remoto que ofrece servicios SSHNewbie [16].

WinSCP es un proyecto liderado por Martin Prikryl, siendo la primera interfaz gráfica de transferencia de ficheros sobre SSH cuyas características principales son [17]:

- Interfaz gráfica de usuario: posibilidad de elegir entre dos interfaces gráficas diferentes.
    - *Commander Interface*: separa la ventana en dos paneles. El panel izquierdo muestra las carpetas locales y el panel derecho las remotas.
    - *Explorer Interface*: similar al explorador de Windows, por lo que sólo aparecen en pantalla los directorios remotos. Para realizar transferencias, basta con situarse en el directorio deseado y arrastrar los ficheros desde el directorio remoto.
- Traducido a multitud de lenguajes: incluyendo idiomas como el español, chino, checo, francés o alemán.
- Integración con Windows: funciones como arrastrar y soltar, URL, accesos directos, etc.
- Operaciones comunes con archivos: navegación, subida y descarga de archivos, manejo de sesiones, edición de ficheros, sincronización, etc.
- Soporte para protocolos SCP y SFTP sobre SSH.
- Sincronización de directorios de manera automática.
- Editor de texto integrado.

3.1.4 RStudio

RStudio es un entorno de desarrollo integrado (IDE) fundado en 2008 y diseñado para un lenguaje de programación de computación estadística denominado R. Existen ediciones comerciales y de código libre, disponibles para Windows, Mac y Linux [18].

El principal motor de inspiración de los creadores de RStudio fue el aumento de la utilización de R en ámbitos de investigación tales como la ciencia, la educación y la industria, apostando por el desarrollo de herramientas gratuitas y libres para la comunidad de R. Tras incrementarse su popularidad, RStudio incorporó productos destinados al ámbito profesional que facilitan el trabajo en equipo.

Los productos comerciales fueron introducidos a finales de 2013 y principios de 2014 para permitir la incorporación de los servicios en empresas. RStudio tiene su sede principal en el distrito de innovación de Boston y una segunda oficina en Seattle. El resto de los empleados se encuentran distribuidos a lo largo de los EEUU en lugares como Texas, California, Iowa y Minnesota [19]. Las principales características de RStudio son [20]:

- Multi-Plataforma.
- Integra las diferentes herramientas de R en un único entorno de trabajo.
- Potentes herramientas diseñadas para mejorar la productividad.
- Navegación entre ficheros y funciones, con facilidad para administrar y manejar múltiples directorios de trabajo.
- Integración de herramientas de representación de datos, con soporte de gráficas interactivas con Shiny y ggvis.
- Amplio manual de ayuda y documentación de R.
- Ejecución de código directamente desde el editor integrado.

3.1.5 OpenVPN

OpenVPN es una herramienta libre y gratuita que permite crear, configurar y gestionar redes virtuales privadas (SSL VPN) con todas sus funcionalidades. Ofrece conectividad punto a punto con validación jerárquica de usuarios y host conectados remotamente, soportando métodos de autenticación del cliente flexibles basados en certificados y credenciales usuario/contraseña, además de permitir a un grupo específico de usuarios acceder a la configuración de las reglas del *firewall* aplicadas en las interfaces VPN virtuales. Su implementación se basa en las capas de nivel 2 y 3 del modelo OSI, utilizando los protocolos SSL/TLS.

OpenVPN 2.0 expande las capacidades de OpenVPN1.x al ofrecer un modo escalable entre cliente/servidor, permitiendo que múltiples clientes se conecten a un único servidor OpenVPN sobre un único puerto TCP o UDP [21].
Las características destacables de OpenVPN son las siguientes [22]:

- Posibilidad de implementar dos modos básicos, en capa 2 o capa 3, lo que permite crear túneles capaces de enviar información en otros protocolos no IP o broadcast.
- Protección de los usuarios remotos.
- Las conexiones pueden ser realizadas a través de casi cualquier *firewall*. Basta con tener acceso a Internet y poder acceder a sitios HTTPS.
- Soporte para proxy.
- Sólo debe ser abierto un puerto en el firewall para permitir conexiones.
- Las interfaces virtuales (tun0, tun1, etc) permiten la implementación de reglas de *firewall* muy específicas.
- Instalación sencilla en cualquier plataforma.
- Permite dos tipos de configuración VPN: TUN y TAP
    - TUN: emula un dispositivo punto a punto. Utilizado para crear túneles virtuales operando con el protocolo IP. Las máquinas que queden detrás de cada uno de los extremos del enlace pertenecen a subredes diferentes.
    - TAP: simula una interfaz de red Ethernet, encapsulando directamente paquetes Ethernet. Las máquinas situadas detrás de cada uno de los extremos del enlace pertenecen a la misma subred.

3.1.6 Apache Maven

Maven se ideó originalmente como un intento de simplificar los procesos de construcción del proyecto Turbine. Apache Turbine es un framework basado en servlet que permite a los desarrolladores con experiencia en Java construir rápida y fácilmente aplicaciones web. El resultado es una herramienta que puede ser utilizada para construir y manejar cualquier proyecto basado en Java, facilitando tareas como el desarrollo o publicación de los contenidos, además de una vía para compartir JARs entre proyectos.

El objetivo principal de Apache Maven es facilitar el trabajo a los desarrolladores Java. Para ello, se apuesta por ciertos factores determinantes [23]:

- Facilitar el proceso de construcción.
- Proveer de un sistema uniforme de trabajo.
- Proveer de información de calidad del proyecto.
- Proporcionar directrices para el desarrollo prácticas óptimas de desarrollo.
- Permitir completa migración con nuevas características.

3.2 Lenguajes de programación

3.2.1 Java

Java es un lenguaje de programación que fue lanzado por Sun Microsystems en 1995. Se trata de un lenguaje de propósito general, concurrente y orientado a objetos que ha sido diseñado específicamente para minimizar el número de dependencias de implementación. Su objetivo principal es la universalización del código, de tal manera que los usuarios Java puedan ejecutar sus desarrollos en cualquier dispositivo. Así, el código ejecutado en una plataforma no tiene que ser recompilado para ejecutarse en otra. Por ello, las aplicaciones Java son generalmente compiladas a bytecode (clase Java) que puede ejecutarse en cualquier máquina virtual Java (JVM) sin depender de la arquitectura del computador.

El lenguaje de programación Java fue originalmente desarrollado por James Gosling de Sun Microsystems y publicado en 1995 como un componente fundamental de la plataforma Java de la misma compañía. Su sintaxis deriva en gran medida de C y C++, aunque tiene menos utilidades de bajo nivel [24]. En la creación de Java se persiguieron cinco objetivos principales:

- Debía ser simple, familiar y orientado a objetos.
- Debía ser robusto y seguro.
- Debía ser independiente de la arquitectura.
- Debía ser ejecutado con el máximo rendimiento.
- Debía poder ser interpretado, concurrente y dinámico.

3.2.2 R

R es un lenguaje destinado a computación estadística disponible para una amplia variedad de plataformas. Este lenguaje consiste en un proyecto GNU de la misma naturaleza del lenguaje S que fue desarrollado por los laboratorios Bell, principalmente por John Chambers. Así, R puede ser considerado como una implementación diferente de S ya que existen diferencias importantes. Sin embargo, la mayoría del código escrito en S es compatible con R.

R provee una amplia variedad de técnicas estadísticas (modelo lineal y no lineal, test estadísticos clásicos, análisis de series temporales, clasificación, *clustering*) y gráficas, siendo altamente ampliable. El lenguaje S es comúnmente el elegido para llevar a cabo investigaciones estadísticas, siendo R el camino *open-source* de participación en esa actividad. Finalmente, uno de los fuertes del lenguaje R es la facilidad y calidad con la que se producen gráficas, incluyendo símbolos matemáticos y fórmulas donde son necesarias [25].

3.3 Tecnologías implicadas (intercambio/almacenamiento datos)

3.3.1 REST

El término REST se definió en el año 2000 en una tesis doctoral sobre la web escrita por Roy Fielding, coautor de la especificación HTTP [26]. REpresentational State Transfer es un tipo de arquitectura de desarrollo web que se apoya en el estándar HTTP, pudiendo interpretarse como un *framework* para construir servicios y aplicaciones web en base a este estándar, siendo más simple que otras alternativa anteriores tales como SOAP o XML.

Los sistemas que siguen los principios REST se denominan RESTful. REST afirma que la web ha disfrutado de escalabilidad como resultado de una serie de diseños clave [27]:

- Un protocolo cliente/servidor sin estado.
- Un conjunto de operaciones bien definidas, siendo las más importantes: POST, GET, PUT y DELETE.
- Una sintaxis universal para identificar los recursos.
- Uso de hiper-medios: la representación de este estado en un sistema REST es típicamente HTML o XML.

3.3.1.1 JSON

JSON (JavaScript Object Notation) es un formato de intercambio de datos. Consiste en un formato fácil de leer y escribir por los humanos así como de analizar y generar por las máquinas.

JSON se puede definir como un subconjunto del lenguaje de programación JavaScript. Aunque se trata de un formato de texto completamente independiente, utiliza nociones familiares para los programadores de lenguajes de la familia C, incluyendo C, C++ y C#, Java, JavaScript, Perl, Python y muchos otros. Estas propiedades convierten a JSON en un candidato ideal para compartir datos entre lenguajes.
JSON está basado en dos estructuras:

- Una colección de pares nombre/valor. En varios lenguajes, esto se realiza mediante un objeto, estructura, diccionario o array asociativo.
- Una lista ordenada de valores. En la mayoría de lenguajes, esto se realiza mediante un array, vector, lista o secuencia.

Estas estructuras son universales ya que virtualmente todos los lenguajes de programación modernos las soportan de una u otra manera. En JSON, adquieren estas formas:

- Un objeto es un par desordenado de nombre/valor. Un objeto comienza con { y termina con }. Cada nombre es seguido por : y el par nombre/valor es separado por coma.
- Un array es una colección ordenada de valores. Un array comienza con [y termina con]. Los valores son separados por comas.
- Un valor puede ser un *string* con dobles comillas, un *número*, *verdadero* o *falso* o *null*, un *objeto* o un *array*. Estas estructuras pueden ser anidadas.

- Un string es una secuencia de cero o más caracteres unicode, envueltos en dobles comillas, usando escapes de barra invertida. Un carácter es representado como un *string* simple. Estas estructuras son muy parecidas a los *strings* de Java o C.
- Un número es similar a Java o C, con la salvedad de que los formatos octal y hexadecimal no son usados [28].

3.3.2 Apache Hadoop

En el proyecto Apache Hadoop se desarrolla *software* de código abierto para computación escalable, distribuida y de confianza. La librería *software* Apache Hadoop es un *framework* que permite el procesamiento de grandes conjuntos usando modelos de programación simples. Está diseñado para tener una escalabilidad que puede ir desde un simple ordenador hasta cientos de máquinas, ofreciendo cada una computación y almacenamiento local. Los módulos que incluye el proyecto son los siguientes [29]:

- **Hadoop Common**: utilidades básicas para soportar el resto de los módulos.
- **Hadoop Distributed File System (HDFS)**: sistema distribuido de ficheros que proporcionan un alto *throughput* de acceso a los datos de aplicación.
- **Hadoop YARN**: *framework* para la planificación de tareas y gestión de recursos de *cluster*.
- **Hadoop MapReduce**: sistema basado en YARN que permite el procesado en paralelo para grandes conjuntos de datos.
- **Hive**: infraestructura de almacenamiento de datos que proporciona la realización de consultas ad hoc.

3.3.2.1 Apache Hive

El *software* de almacenamiento de datos Apache Hive facilita la consulta y gestión de grandes conjuntos de datos que residen en almacenamiento distribuido. Está construido sobre Apache Hadoop, proporcionando [30]:

- Herramientas para facilitar la extracción, transformación o carga de datos (ETL).
- Un mecanismo para establecer la estructura en una variedad de formatos de datos.
- Acceso a los ficheros almacenados, ya sean en Apache HDFS o en otro sistema de almacenamiento de datos como Apache HBase.
- Ejecución de consultas a través de MapReduce.

Hive define un lenguaje simple de consultas como SQL llamado QL, que permite a los usuarios familiarizados con SQL consultar los datos.

3.3.3 Apache Flume

Apache Flume [31] es un proyecto de alto nivel en la fundación Apache Software. Consiste en un servicio distribuido, seguro y disponible que permite recoger, agregar y mover de manera eficiente grandes cantidades de datos desde muchas fuentes diferentes hasta un almacén de datos centralizado. Su principal objetivo es entregar los datos de aplicaciones a Apache Hadoop y tiene una arquitectura simple y flexible basada en el *streaming* de flujos de datos. Además, es robusto y tolerante a fallos con multitud de mecanismos de seguridad y de recuperación y utiliza un modelo de datos simple y extensible que permite aplicaciones analíticas en línea.

El uso de Apache Flume no sólo se limita a la agregación de datos. Desde que las fuentes de datos son configurables, es posible utilizar Flume para el transporte de grandes cantidades de datos (eventos), incluyendo pero no limitado a datos de tráfico de red, datos generados por los medios sociales, mensajes de correo electrónico y prácticamente cualquier fuente posible.

4 Patrón de diseño Publish/Subscribe

4.1 Introducción

Internet ha cambiado considerablemente la escala de los sistemas distribuídos. Actualmente dichos sistemas abarcan cientos de entidades cuyas localizaciones y comportamientos pueden variar significativamente. Estas restricciones hacen necesario modelos y sistemas de comunicación más flexibles que sean capaces de reflejar el dinamismo y el desacople de las características entre aplicaciones. Como consecuencia directa, el paradigma de comunicación *Publish/Subscribe* [32] está recibiendo progresivamente mayor atención. En los sistemas basados en los esquemas de interacción *Publish/Subscribe*, los **suscriptores** registran su interés en un evento o en un grupo de eventos, siendo notificados posteriormente de los eventos generados por los **publicadores**. Así, han aparecido una gran cantidad de variaciones basadas en esta idea, cada una adaptada a unos requisitos o características específicas del modelo de aplicación o red.

Básicamente, los suscriptores tienen la capacidad de expresar su interés en un evento o eventos de características determinadas, siendo posteriormente notificados por cualquier evento cuyas características cumplan los requisitos de la suscripción previa.

Un evento se propaga de manera asíncrona a todos los suscriptores que han registrado un interés en él. La potencia de este estilo de interacción basado en eventos radica en el desacople total en **tiempo**, **espacio** y **sincronización** entre publicadores y suscriptores. Muchos sistemas industriales apoyan este estilo de interacción, existiendo actualmente un notable auge en investigaciones basadas en los esquemas de interacción *Publish/Subscribe*.

4.2 Esquema básico de interacción

En resumen, los productores publican información en un bus *software* (**manejador de eventos**) y los consumidores se suscriben a la información que ellos quieren recibir de ese bus. Esta información típicamente se denota con el término de **evento**, mientras que el acto de entregarlo se denomina **notificación**.

El modelo básico de interacción *Publish/Subscribe* (Figura 4.1) se basa en un servicio de notificación de eventos proporcionando almacenamiento y gestión de las suscripciones así como una eficiente entrega de los eventos.

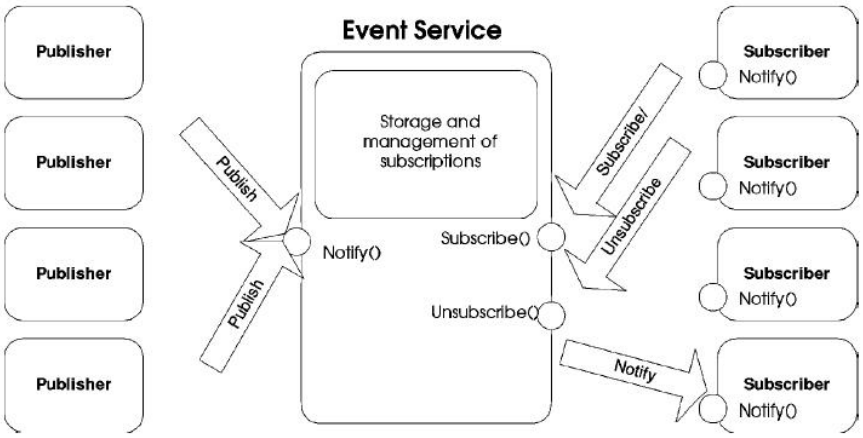


Figura 4.1. Esquema básico *Publish/Subscribe.*

Los **suscriptores** normalmente registran su interés por un evento llamando al método `subscribe()` del mismo, sin conocimiento alguno de la fuente de ese evento. Esta información de suscripción se queda almacenada en el **servicio de eventos** y no se envía a los **publicadores**. La operación contraria `unsubscribe()` finaliza la suscripción al evento.

Para generar un evento, el publicador llama a la operación `publish()`. El **servicio de eventos** propaga el evento a todos los suscriptores, por lo que de alguna manera puede ser visto como un proxy de los suscriptores.

Los **publicadores** también poseen la capacidad de advertir la naturaleza de eventos futuros a través de la operación `advertise()`. Esta información puede llegar a ser realmente útil para el **servicio de eventos** avisándole y preparándole ante futuras llegadas así como para los **suscriptores** para preveer cuando una nueva información estará disponible.

El desacople o desincronización que proporciona el **servicio de eventos** entre el **publicador** y el **suscriptor** puede ser descompuesto en tres dimensiones diferentes (Figura 4.2):

- *Space Decoupling*: los extremos no necesitan conocerse uno al otro. El **publicador** publica eventos a través del **servicio de eventos** de la misma forma que los **suscriptores** reciben los eventos de él. Los **publicadores** no tienen conocimiento de cuantos **suscriptores** consumirán esos eventos, al igual que estos últimos desconocen el número de **publicadores** que inyecta dicha información.
- *Time Decoupling*: los extremos no necesitan estar activos simultáneamente para realizar un intercambio de información satisfactorio. Así, el **publicador** puede publicar varios eventos mientras que el **suscriptor** está desconectado. De la misma forma, el **suscriptor** puede consumir los eventos mientras que el **publicador** esté desconectado.
- *Synchronization Decoupling*: Los **publicadores** no están bloqueados mientras producen eventos, y los **suscriptores** obtienen las notificaciones de dichos eventos de manera asíncrona mientras realizan cualquier otra actividad de manera concurrente. La producción y consumición de eventos no ocurren en el flujo de control principal de los extremos, y por tanto no suceden de manera síncrona.

La eliminación de las dependencias entre los extremos aumenta la escalabilidad además de reducir la coordinación entre las entidades. La infraestructura de comunicación que se obtiene es la idónea para entornos asíncronos por naturaleza, como por ejemplo los entornos móviles.

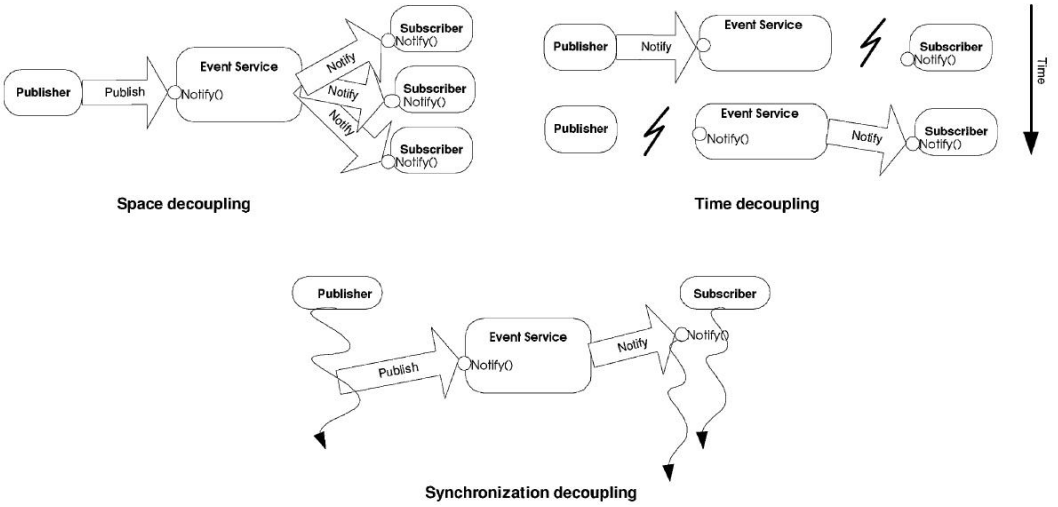


Figura 4.2. Desacople en espacio, tiempo y sincronización

4.3 Variaciones de Publish/Subscribe

Las diferentes formas en las que un suscriptor informa de los eventos que le interesan han llevado a desarrollar varios esquemas de suscripción. A continuación, se describen los esquemas basados en tema, contenido y tipo.

4.3.1 Publish/Subscribe basado en tema

Los primeros esquemas *Publish/Subscribe* se basan en temas, siendo implementados por muchas soluciones industriales. Este enfoque extiende la noción de los canales, utilizados para agrupar la comunicación entre usuarios, con métodos para caracterizar y clasificar el contenido del evento. Concretamente, se pueden realizar publicaciones y suscripciones a eventos basados en temas específicos, que son identificados por palabras clave o *keywords*.

El hecho de suscribirse a un tema T puede verse como ser miembro de un grupo T, y todo evento publicado en el tema T será enviado a todos los miembros pertenecientes al grupo T. En la práctica, cada tema puede ser visto como un canal de comunicación diferente. De esta forma, cada tema presenta la misma arquitectura que la discutida en la sección 4.2, donde el nombre del tema es especificado como argumento en la inicialización de la suscripción.

Así, cada tema es visto como un **servicio de eventos** en sí mismo, identificado por un nombre único, con una interfaz que ofrece las operaciones *publish()* y *subscribe().* Como virtudes de este esquema de funcionamiento, cabe destacar la organización jerárquica de los temas, permitiendo a los programadores anidar temas dentro de los temas. De esta manera, cuando un suscriptor realiza la suscripción a un tema, se suscribirá automáticamente a todos los subtemas que lo formen. Por ello, podemos decir que la suscripción se realiza por especificidad.

A continuación se indica un ejemplo basado en las cotizaciones en bolsa (StockQuotes) de unos valores determinados (Stock) en el mercado de Londres (LondonStockMarket). En la cabeza jerárquica estará el tema LondonStockMarket, siendo Stock un subtema del primero y StockQuotes un subtema del segundo. Los eventos de StockQuotes contienen cinco atributos: (1) identificador global, (2) nombre de la compañía, (3) precio, (4) cantidad, (5) identificador del vendedor. Se desea comprar acciones de la compañía con nombre TELCO cuyo precio sea menor de 100$. Para ello, se debe realizar una suscripción al subtema StockQuotes, donde se permita recibir todos los eventos de ese tema. Una vez recibido el evento, se tiene la posibilidad de analizar cada uno de los atributos para proceder o no a la compra. La Figura 4.3 muestra el pseudocódigo para realizar las suscripciones, mientras que la Figura 4.5 muestra el resultado de las interacciones, donde m1 y m2 son mensajes del tema StockQuotes.

```
public class StockQuote implements Serializable {
  public String id, company, trader;
  public float price;
  public int amount;
}
public class StockQuoteSubscriber implements Subscriber {
  public void notify(Object o) {
    if (((StockQuote)o).company == 'TELCO' && ((StockQuote)o).price < 100)
      buy();
  }
}
// ...
Topic quotes = EventService.connect("/LondonStockMarket/Stock/StockQuotes");
Subscriber sub = new StockQuoteSubscriber();
quotes.subscribe(sub);
```

Figura 4.3. Código de ejemplo de suscripción basada en tema.

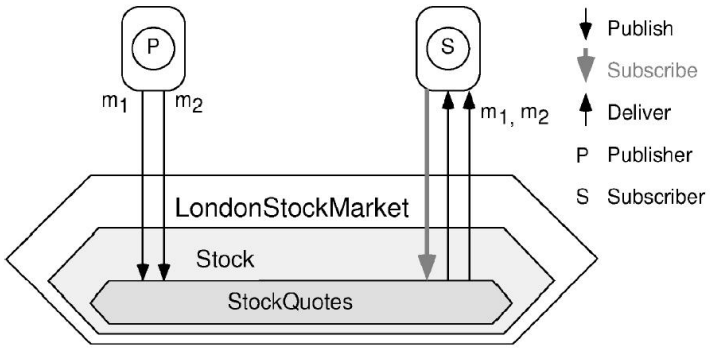


Figura 4.4. Interacciones de suscripción basada en tema.

4.3.2 Publish/Subscribe basado en contenido

La variante de *Publish/Subscribe* basada en contenido mejora el enfoque anterior al introducir un esquema de suscripción basado en el contenido actual de los eventos considerados. En otras palabras, los eventos no son clasificados de acuerdo a criterios externos predefinidos, sino de acuerdo a las propiedades de los eventos propiamente dichas. Estas propiedades pueden ser atributos o meta-datos asociados a los eventos.

Específicamente, los consumidores se suscriben a una selección de eventos de acuerdo a un filtro definido mediante un lenguaje de suscripción. El filtro define las restricciones que identifican los eventos válidos. Estas restricciones normalmente se definen mediante parejas nombre-valor y con operadores básicos de comparación como menor que (<), igual que (=) o mayor que (>). Además, estas restricciones pueden ser combinadas mediante operadores lógicos (AND, OR, XOR, etc). Para la suscripción se utiliza una variante de la operación `subscribe()`, al incluir un argumento adicional que represente al patrón de suscripción. Existen varios medios para representar estos patrones:

- *String*: Es la manera más frecuente de expresar el patrón de suscripción. Se utiliza una gramática determinada en base a un lenguaje determinado e inteligible por el **servicio de eventos** para poder leerlo y realizar la suscripción.
- Plantilla de objeto: Con este tipo, el suscriptor provee al **servicio de eventos** de una plantilla del tipo de eventos que demanda. Así pues, cuando se realiza una suscripción, el suscriptor envía un objeto T, que indica que está interesado en cualquier evento que sea conforme al tipo de T y cuyos atributos coincidan con los de T.
- Código ejecutable: El suscriptor proporciona un objeto capaz de filtrar eventos en tiempo de ejecución. Esta implementación se suele dejar al desarrollador de la aplicación. Este tipo no es usado debido a que los filtros resultantes son difíciles de optimizar.

Las Figuras 23 y 24 ejemplifican el mismo caso explicado en el apartado anterior. En este caso, para obtener la misma funcionalidad que con *Publish/Subscribe* basado en tema, el suscriptor debe establecer el filtrado de eventos en la propia suscripción, mientras que con el primer tipo se recibían todos los eventos basados en el tema especificado y una vez recibidos por parte del suscriptor se realizaba el filtrado de ellos, pudiéndolos descartar o no.

```
public class StockQuote implements Serializable {
   public String id, company, trader;
   public float price;
   public int amount;
}
public class StockQuoteSubscriber implements Subscriber {
   public void notify(Object o) {
      buy();       // company == 'TELCO' and price < 100
   }
}
// ...
String criteria = ("company == 'TELCO' and price < 100");
Subscriber sub = new StockQuoteSubscriber();
EventService.subscribe(sub, criteria);
```

Figura 4.5. Código de suscripción basada en contenido.

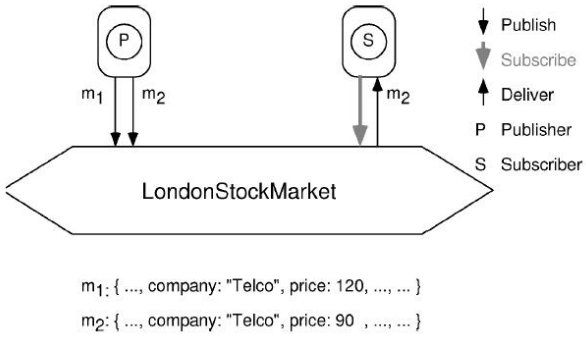


Figura 4.6. Interacciones de suscripción basada en contenido.

4.3.3 Publish/Subscribe basado en tipo

El esquema *Publish/Subscribe* basado en tipo hace uso de mecanismos adicionales para clasificar mejor los recursos y optimizar los servicios de notificación en el servicio de eventos. Además de permitir una perfecta integración entre middleware y lenguaje, el esquema basado en tipo tiene varias ventajas sobre los estilos vistos hasta ahora, habilitando a las aplicaciones describir en tiempo de ejecución las propiedades de los mensajes que desean recibir. Este esquema introduce el concepto de usar **tipos** en el sistema, permitiendo a los eventos clasificados previamente por nombre bajo una determinada jerarquía en árbol, ser clasificados por tipos siguiendo esa misma jerarquía.

La ventaja fundamental de este sistema con respecto al basado en temas reside en que ahora no necesitamos especificar toda la ruta del árbol jerárquico para realizar una suscripción, sino que, esos nombres, son clasificados como tipos de datos, realizando ahora suscripciones a tipos determinados de datos que no deben por qué cumplir una jerarquía determinada.

En el ejemplo descrito en el apartado 4.3.1, se realizaba una suscripción a StockQuotes alojada en LondonStockMarket/Stock, por lo que se limitaba la notificación de los eventos a todos aquellos producidos por esa ruta. Ahora con el esquema basado en tipo, se pueden realizar suscripciones a StockQuotes independientemente de la estructura jerárquica que haya detrás de ella, recibiendo eventos de todos los StockQuotes que se publiquen, independientemente de si es del LondonStockMarket o si es cualquier otro.

Otra nueva característica de este esquema es que asegura la seguridad del tipo (*type-safety*) en tiempo de compilación parametrizando la interfaz resultante con el tipo correspondiente del evento.

Así pues, se puede observar como este enfoque combina técnicas de sus predecesores. Por una parte se basa en la filosofía de suscripción de temas para extrapolarlo a la suscripción en tipo, mientas que por otro lado, la inclusión de técnicas como el *type-safety* permite una mejor y más eficiente coincidencia en la búsqueda de eventos que cumplan los requisitos demandados, mejorando sustancialmente los esquemas de *Publish/Subscribe* basado en contenido. Las Figuras 25 y 26 muestran los ejemplos de código y flujo de interacciones para el escenario estudiado en los apartados anteriores.

```
public class LondonStockMarket implements Serializable {
   public String getId() {...}
}
public class Stock extends LondonStockMarket {
   public String getCompany() {...}
   public String getTrader() {...}
   public int getAmount() {...}
}
public class StockQuote extends Stock {
   public float getPrice() {...}
}
public class StockRequest extends Stock {
   public float getMinPrice() {...}
   public float getMaxPrice() {...}
}
public class StockSubscriber implements Subscriber<StockQuote> {
   public void notify(StockQuote s) {
      if (s.getCompany() == 'TELCO' && s.getPrice() < 100)
         buy();
   }
}
// ...
Subscriber<StockQuote> sub = new StockSubscriber();
EventService.subscribe<StockQuote>(sub);
```

Figura 4.7. Código de suscripción basada en tipo.

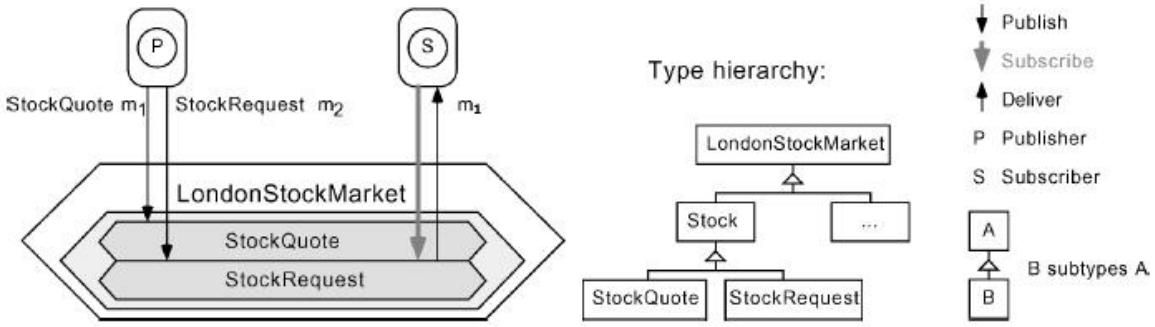


Figura 4.8. Interacciones de suscripción basada en tipo

5 FI-WARE

FI-WARE aspira a proporcionar un *framework* para el desarrollo de *smart applications* en el ámbito de Internet de las Cosas y está siendo desarrollado como parte del programa FI-PPP (Future Internet Public Private Partnership) lanzado por la Comisión Europea en colaboración con la industria de las TIC [33].

La plataforma FI-WARE es *open* y basada en elementos (a continuación llamados **Generic Enablers**, **GE**) que ofrecen funciones reutilizables y comúnmente compartidas al servicio de una multiplicidad de áreas de uso a través de diversos sectores. Así, su arquitectura se estructura en una serie de capítulos técnicos, a saber:

- **Cloud Hosting**
- **Data/Context Management**
- **Internet of Things (IoT) Services Enablement**
- **Applications/Services Ecosystem and Delivery Framework**
- **Security**
- **Interface to Networks and Devices (I2ND)**

Específicamente, en este apartado se estudia el capítulo *Data/Context Management* ya que provee de mecanismos para un acceso efectivo, procesado y análisis de flujos masivos de datos.

5.1 FI-WARE Data/Context Management

5.1.1 Introducción

Los GEs del capítulo Data/Context Management permiten:

- Generación, suscripción para recibir alertas y consulta de información de contexto proveniente de diferentes fuentes.
- Cambios en el modelo de contexto como eventos que pueden ser procesados para detectar situaciones complejas que conduzcan a la generación de acciones o de nueva información de contexto.
- Procesamiento de grandes cantidades de información de contexto de forma agregada, utilizando las técnicas **BigData** y **Map&Reduce**, para generar nuevo conocimiento.
- Procesamiento de flujos de datos (en particular, las secuencias de vídeo multimedia) procedentes de diversas fuentes con el fin de generar nuevos flujos de datos, así como información de contexto que puede ser explotada adicionalmente.
- Gestionar información de contexto, tales como información de ubicación, presencia, perfil de usuario o de terminal, etc, de una manera estándar.
- Administrar/Publicar datos abiertos, en particular como datos de contexto en tiempo real.
- Utilizar los datos y los medios existentes para enriquecer aplicaciones.

A continuación, se describen conceptos fundamentales dentro de este capítulo como: **Data**, **DataElement**, **Meta-data**, **Context**, **Context-Element** y **event**.

- **Data**, en FIWARE, se refiere a la información que es producida, generada, recogida u observada que puede ser relevante para el procesado. Tiene asociado un *DataType* y un *Value*.
- Un **Data Element** se refiere al dato cuyo valor está definido por una secuencia de uno o varios tripletes de atributos <*name*, *type*, *value*> denominados *Data Element attributes*.

Puede haber **Meta-data** vinculados a los atributos de un elemento de datos. Estas estructuras de datos se ven reflejadas en la Figura 5.1.

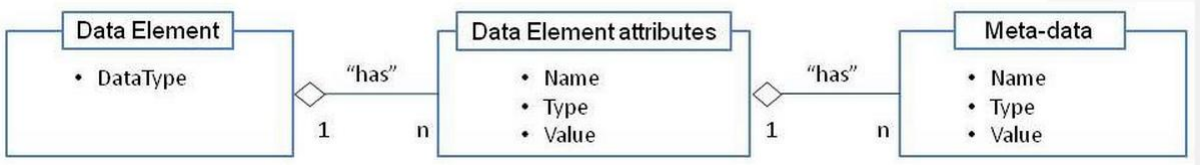


Figura 5.1. Estructura *Data Element.*

Lo interesante en FIWARE es que los *Data Elements* no están sujetos a un formato de representación de datos específico. Pueden ser representados en formato XML, almacenados en bases de datos relacionales, en un repositorio RDF o como entradas en una base de datos noSQL como MongoDB, adoptando un formato particular de almacenamiento que puede ser el mismo o diferente al formato usado en la transferencia de la información.

- **Context**, en FIWARE, está representado a través de los *Context Elements*.

Los *Context Elements* extienden el concepto de los *Data Elements* mediante la asociación de un *EntityId* y un *EntityType* a ellos, identificando de manera unívoca la entidad en el sistema FIWARE al que se refiere la información del elemento. Los *Context Elements* son creados conteniendo el valor de atributos que caracterizan a la *entity* (entidad) en un momento dado. Como ejemplo, un *Context Element* puede contener valores como la temperatura, metros cuadrados o humedad, atributos asociados a una habitación. La Figura 5.2 ilustra su estructura.

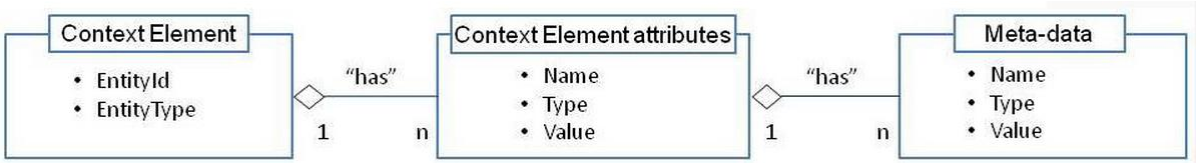


Figura 5.2. Estructura *ContextElement.*

- Un **event** (evento) es una ocurrencia dentro de un sistema o dominio particular.

Los eventos llevan normalmente a la creación de algún *Data* o *Context Element*, permitiendo así que la información que describen o relacionan sea tratada por aplicaciones diseñadas para tal fin como los Generic Enablers de FIWARE.

Finalmente, la palabra **event object** es usada en el sentido de una entidad de programación que representa una ocurrencia (evento) en un sistema de computación. En FIWARE, los *event objects* son creados internamente para algunos GEs como el **Publish/Subscribe Context Broker GE**.

5.1.2 Arquitectura Data/Context Management

Los siguientes diagramas muestran los principales componentes que comprenden la cuarta versión de la arquitectura del capítulo FIWARE *Data/Context* (Figura 5.3) y las integraciones más importantes entre Generic Enablers (Figura 5.4) [34].

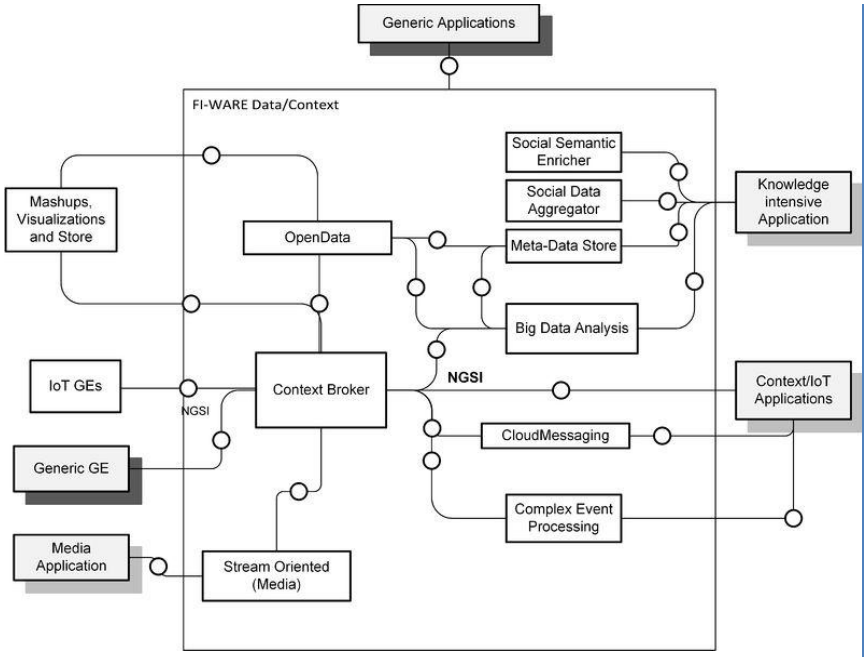


Figura 5.3. Arquitectura capítulo *Data/Context*.

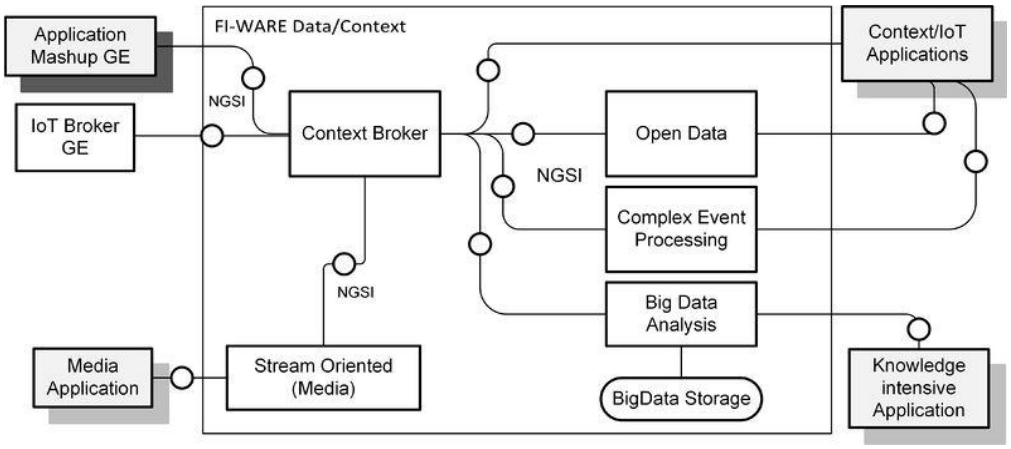


Figura 5.4. Integración de GEs *Data/Context*.

Cabe destacar los siguientes tipos de integración entre *Generic Enablers* (Figura 5.4):

1. **Context Broker (CB) – Context Event Processing (CEP)**: el CEP implementa adaptaciones específicas NGSI de los conectores REST para permitir la recepción y generación de notificaciones de *context events*. Esta integración se basa únicamente en notificaciones tipo *ONCHANGE*.
2. **Context Broker – Big Data Analysis (Cosmos)**: se ha implementado un adaptador para almacenar de forma automática en *Cosmos* todas las *context notifications* relacionadas con una *context entity*. Estos datos pueden usarse después para análisis *Map&Reduce* con carga en tablas externas (Hive).

3. **Context Broker - Open Data**: el GE Open Data puede catalogar y mostrar *context information*, definiendo recursos de conjunto de datos como *context queries*.
4. **Stream Oriented (Kurento)**: Kurento GE filtra y analiza contenido multimedia, entre otras posibilidades. Las aplicaciones desplegadas con este GE reciben información en tiempo real acerca del multimedia analizado. A través de componentes Java, es posible enviar NGSI *context notifications* al CB.
5. **IoT Broker (capítulo IoT)**: gracias a éste GE, el CB puede recibir notificaciones acerca de la información de contexto generada y publicada por los dispositivos IoT.
6. **Aplicaciones Mashups (Wirecloud):** a través de una librería *Javascript*, los *widgets* implementados en Wirecloud pueden interactuar con el CB tanto para la recepción como para la entrega de información de contexto.

La integración entre los *Generic Enablers* Context Broker y Wirecloud (6) ha sido estudiada en el Proyecto Fin de Carrera titulado "Contribución a la Infraestructura FI-WARE. Integración de Dispositivos Empotrados de Bajo Coste" [35]. **En el presente trabajo de investigación se aborda la integración (2)** mencionada anteriormente. Por este motivo, en los siguientes apartados se introducen los aspectos más relevantes de los *Generic Enablers* Context Broker y Cosmos.

5.1.3 Publish/Subscribe Context Broker GE - Orion Context Broker

Orion Context Broker es una implementación C++ del *Publish/Subscribe Context Broker* GE que proporciona las interfaces NGSI9 y NGSI10 [36]. Dichas interfaces son APIs de código abierto que pueden ser descargadas a través de la web de FIWARE e implementan la Transferencia de Estado Representacional (REST). Por este motivo, la comunicación se realiza mediante el protocolo HTTP para lo cual se pueden emplear dos posibles formatos de intercambio de datos: JSON y XML. Utilizando estas interfaces, se pueden realizar diferentes operaciones tales como:

- Registrar aplicaciones productoras de información de contexto.
- Actualizar información de contexto.
- Recibir notificaciones cuando se producen cambios en la información de contexto
- Consultar información de contexto.

5.1.3.1 Arquitectura Orion Context Broker

La Figura 5.5 muestra una arquitectura lógica [37] del *Generic Enabler Publish/Subscribe Context Broker* con sus principales componentes e interacciones con otros actores:

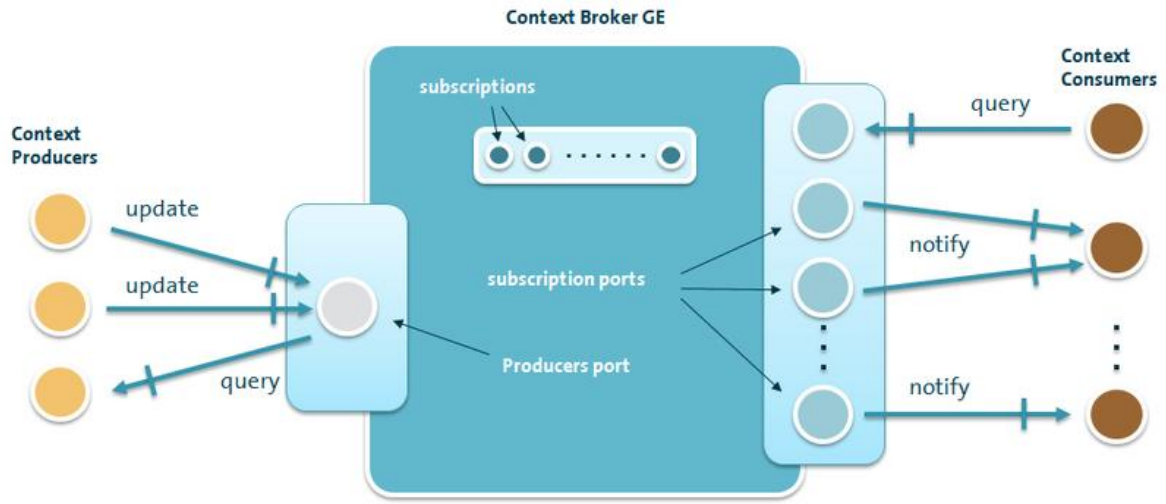


Figura 5.5. Arquitectura *Publish/Subscribe Context Broker*.

Los actores del modelo del *Generic Enabler Publish/Subscribe Context Broker* son:

- **Publish/Subscribe Context Broker**: es el principal componente de la arquitectura. Funciona como un manejador y agregador de datos de contexto y como una interfaz entre los actores de la arquitectura. Principalmente, el CB controla el flujo de contexto entre todos los actores. Por este motivo, el CB tiene que saber acerca de todos los *Context Providers* (CPr). Esta característica se realiza a través de un proceso de anuncio.
- **Context Producer:** un *Context Producer* (CP) es un actor capaz de generar contexto. El *Context Producer* básico es el que actualiza información de contexto de forma espontánea, sobre uno o más atributos de contexto de acuerdo con su lógica interna. Esta comunicación entre CP y CB se realiza en modo *push*, del CP al CB. Los CP también pueden trabajar en modo *pull*, en cuyo caso se denominan *Context Providers*.
- **Context Provider (CPr)**: es un tipo especializado de actor *Context Producer*, que proporciona información de contexto a demanda, en modo síncrono. Esto significa que el *Publish/Subscribe Context Broker* o incluso el *Context Consumer* puede invocar el CPr con el fin de adquirir la información de contexto. Un CPr proporciona datos de contexto sólo a raíz de una invocación específica. Por otra parte, un CPr puede producir nueva información de contexto a partir del cálculo de los parámetros de entrada; por lo tanto es muchas veces responsable del razonamiento en la información de contexto de alto nivel y de la fusión de datos del sensor. Cada CPr registra su disponibilidad y capacidades mediante el envío de los anuncios correspondientes al CB y expone interfaces para proporcionar información de contexto al CB y a los *Context Consumers*.
- **Context Consumer**: un *Context Consumer* (CC) es una entidad (por ejemplo, una aplicación basada en contexto) que explota la información de contexto. Un CC puede recuperar información de contexto enviando una solicitud al CB o invocando directamente al CPr a través de una interfaz específica. Otro camino para que el CC obtenga información es mediante la suscripción a actualizaciones de información de contexto que responden a ciertas condiciones (por ejemplo, están relacionados con cierto conjunto de entidades). El CC registra una operación *call-back* con la suscripción, por lo que el CB notifica al CC acerca de actualizaciones relevantes en el contexto invocando esta función *call-back*.
  Por último, una especie de CC puede exponer operaciones de actualización de contexto a ser invocadas por el CB. Esto está principalmente relacionado con las capacidades de actuación (por ejemplo, encender/apagar una lámpara).
- **Entity**: cada intercambio de datos de contexto se refiere a una entidad específica, que puede ser en su orden un grupo complejo de más de una entidad. Una *entity* es el tema (por ejemplo, usuario o grupo de usuarios, cosas o grupo de cosas, etc.) al que hacen referencia los datos de contexto. Se compone de dos partes: un tipo y un identificador. Cada *Context Provider* soporta uno o más tipos de entidad y esta información se publica al *Publish/Subscribe Context Broker* durante el proceso de anuncio.
  Un tipo es un objeto que clasifica un conjunto de entidades, por ejemplo dispositivos móviles – identificados por *imei*.

Así, el *Publish/Subscribe Context Broker GE* habilita la publicación de la información de contexto por parte de las *entities* (denominadas *Context Producers*), para que la información de contexto publicada esté disponible para otras *entities* (denominadas *Context Consumers*). En la plataforma FIWARE, aplicaciones o incluso otros GEs pueden desempeñar el rol de los *Context Producers*, *Context Consumers* o ambos.

El principio fundamental soportado por el *Publish/Subscribe Context Broker* es el de conseguir total disociación entre los *Context Producers* y los *Context Consumers*. Esto es posible gracias a la implementación del Patrón de Diseño *Publish/Subscribe* (capítulo 4) y más concretamente, a la variación basada en contenido descrita en el apartado 4.3.2. Este hecho permite que los *Context Producers* publiquen datos sin el conocimiento de quién, cuándo o dónde sean consumidos por los *Context Consumers*, por lo que no es necesaria una comunicación directa entre ellos. Por otro lado, los *Context Consumers* pueden consumir dicha información de su interés independientemente de las publicaciones que realicen los *Context Producers*. Esto es debido a que los *Context Consumers* están interesados en el tipo de información y no en quién publica dicha información. Como resultado, el *Publish/Subscribe Context Broker* es un excelente puente que permite a aplicaciones externas manejar eventos relacionados con el Internet de las Cosas de una manera simple.

5.1.3.2 Modos de Comunicación

El *Generic Enabler Publish/Subscribe Context Broker* puede proporcionar el acceso a la información de contexto por parte del *Context Consumer* cuando éste lo requiera (*on-request*), o cuando dicha información disponible (*on-subscription*).

- **Modo on-request:** extrae eventos como respuesta de peticiones enviadas por los *Context Consumers* a través del *Publish/Subscribe Context Broker*.
- **Modo on-subscription:** permite la configuración de las condiciones necesarias para que los eventos sean enviados a los *Context Consumers*. En caso de que el evento cumpla con las condiciones establecidas en la suscripción, el CB invoca la operación de notificación (*notify*) hacia el CC. Las suscripciones pueden ser configuradas por terceras aplicaciones y no necesariamente por los CC en sí mismos.

5.1.4 Big Data Analysis GE - Cosmos

El *Generic Enabler Big Data Analysis* está destinado a desplegar medios para analizar tanto *batch data* como *stream data*, con el fin de conseguir, al final, perspectivas sobre estos datos que revelan nueva información que estaba oculta. *Batch data* son almacenados por adelantado, y la latencia no es extremadamente importante cuando son procesados. *Stream data* son recibidos y procesados casi en tiempo real, y se espera que los resultados estén listos de inmediato, básicamente porque este tipo de datos no se almacena y las perspectivas deben ser tomadas al instante. Aunque el bloque de *Streaming* está en fase de desarrollo, el bloque de *Batch* ha sido ampliamente desarrollado a través de la adopción y/o la creación de las siguientes herramientas:

- **Hadoop As A Service (HAAS) engine**: ya sea el "oficial " que se basa en el *Openstack* de Sahara, ya sea la versión ligera basada en un *cluster* Hadoop compartido.
- **GUI Cosmos** y **OAuth2 Tokens Generator** para las APIs REST Cosmos.
- **Cygnus**, el conector de datos para *Orion Context Broker*.
- **Tidoop**, extensiones para Hadoop incluyendo medios para utilizar datos CKAN y algunos trabajos de propósito general MapReduce.

5.1.4.1 Stream Processing Block

Este bloque se encuentra en fase de desarrollo.

5.1.4.2 Batch Processing Block

Este bloque pretende ser una plataforma de servicio Big Data [38] en la parte superior de la infraestructura *Openstack*. Esta plataforma será la encargada de proporcionar, principalmente, *clusters* Apache Hadoop bajo demanda. También pueden ser consideradas otras soluciones de análisis basadas en *clusters* como Apache Spark.

Esta parte del GE diferencia claramente entre servicios de computación y almacenamiento:

- El servicio de computación permite crear y configurar un *cluster* coordinado de máquinas de una manera rápida mediante línea de comandos o API REST. Además, el GE permite seleccionar un subconjunto de una amplia gama de tecnologías preconfiguradas de procesamiento y coordinación (Hadoop, Hive, Oozie, HBase, Sqoop, PIG, etc).
- El servicio de almacenamiento permite nutrir a los *clusters* con una gran variedad de datos sin ningún adaptador o configuración especial. La tecnología utilizada en *Cosmos* permite un alto *throughout* debido a que no se realizan traducciones de los datos. Para almacenar datos en el servicio de almacenamiento del GE es tan fácil como usar la línea de comandos o la API REST.

El uso de cualquiera de los servicios (computación o almacenamiento) no implica usar la el otro. De esta manera, el almacenamiento de datos es independiente del ciclo de vida de datos asociado a su procesamiento. Además, este bloque enriquece el ecosistema Hadoop con herramientas específicas que permiten la integración de *Cosmos* con otros GE como el *Publish/Subscribe Context Broker* o el *Open Data Portal*.

5.1.4.3 Arquitectura de Cosmos

La Figura 5.6 ilustra la arquitectura de Cosmos. Este *Generic Enabler* es una expansión de Apache Hadoop que ha sido descrito en el apartado 3.3.2. En general, la instancia del *Big Data GE* consta de un *Master Node*, en el que se ejecuta un *software* de administración y que actúa como frontera con los usuarios finales. Normalmente, estos usuarios son aplicaciones que utilizan la API de administración para enviar solicitudes de creación de nuevos *clusteres* de almacenamiento y/o computación.

Por un lado, la creación de un *cluster* implica la creación de un *Head Node*, encargado de la gestión del almacenamiento, la computación y otros servicios como herramientas de análisis o inyectores de datos (*Data Injectors*). Por otro lado, son creados uno o más nodos esclavos (*Slave Nodes*), cuya misión es la ejecución de las tareas de análisis y del almacenamiento real de los datos.

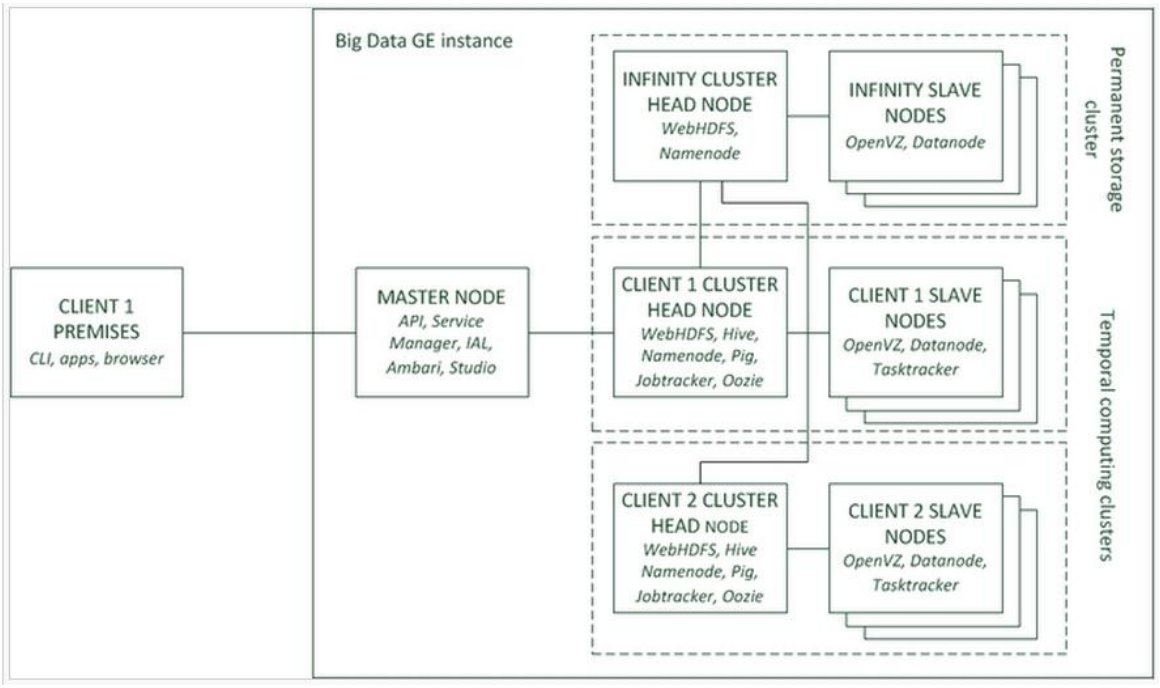


Figura 5.6. Arquitectura de Cosmos.

El enfoque de este GE se basa en el almacenamiento y el análisis de los datos. Específicamente, los usuarios solo tienen que pensar en desarrollar aplicaciones sin preocuparse por otros aspectos internos como la paralelización, distribución, tamaño o escalabilidad. Por este motivo, no importa tanto la arquitectura interna de este GE ni cómo manejen los componentes la información, sino la funcionalidad que puede aportar.

5.1.4.4 HDFS RESTful APIs

Las especificaciones utilizadas en el presente trabajo son WebHDFS y HttpFS:

- **WebHDFS:** existe una serie de comandos Hadoop para realizar operaciones de administración de los *clusteres* y acceso a la información. No obstante, en ciertas ocasiones es necesario acceder a los recursos HDFS mediante una entidad externa. Para conseguir esto, Hadoop proporciona una API HTTP Rest soportando una completa interfaz para HDFS. Esta interfaz permite la creación, el renombrado, la lectura de ficheros y carpetas, además de otras muchas operaciones con el sistema de ficheros. Está disponible en el puerto TCP 50070.
- **HttpFS:** se trata de otra implementación de la API de WebHDFS (puerto 14000 de TCP), aunque su comportamiento es diferente. Cuando se usa WebHDFS, tanto el *Head Node* (Namenode) como los *Slave Nodes* (*Datanode*) deben tener una IP pública. Esto es debido a que ambos tipos de nodos (*Namenode* y *Datanode*) deben de ser alcanzables. Este problema implica el uso de un alto número de direcciones IP públicas. Httpfs actúa como un gateway entre el cliente htpp y WebHDFS, y en vez de redireccionar hacia los *Slave Nodes*, vuelve a dirigir al propio *Head Node*. Ante tales facilidades de funcionamiento y utilización que encapsulan ciertos aspectos de infraestructura, esta es la especificación más utiliza [39].

5.1.4.5 Sistemas de consulta

- **Hive:** descrito en el apartado 3.3.2.1, proporciona mecanismos de consulta a los datos mediante un lenguaje similar a SQL llamado HiveQL. Este sistema de consulta ha sido utilizado en el desarrollo del proyecto.
- **Pig:** herramienta de consulta de datos similar a Hive. Apache Pig es una plataforma que consiste en la utilización de lenguajes de programación de alto nivel para la creación de *software* de análisis de datos. La propiedad más destacada de los programas Pig es que su estructura es compatible con la paralelización, permitiendo un mejor manejo de grandes cantidades de datos.

5.1.5 Cygnus

Cygnus [40] es un conector encargado de persistir datos de contexto procedentes del *Orion Context Broker* en almacenamientos de terceros, creando una vista histórica de tales datos. En otras palabras, el *Orion Context Broker* sólo almacena el último valor en relación con el atributo de una entidad, comportándose como una base de datos en tiempo real. Si se desea guardar los datos de manera histórica para su posterior análisis es necesaria una herramienta que funcione como puente de conexión y que envíe la información a otro almacenamiento que constituya una base de datos histórica. Cygnus es esta herramienta, un puente entre tecnologías que convierte la información de tiempo real en información histórica.

Además, Cygnus utiliza las operaciones de suscripción/notificación de datos del *Orion Context Broker*. Así, basta con realizar una suscripción en el *Orion Context Broker* de parte de Cygnus, detallando cuáles son las entidades de las que se quiere ser notificado cuando se produzca una actualización de cualquiera de sus atributos.

5.1.5.1 Arquitecturas de Cygnus

Internamente, Cygnus está basado en Apache Flume (apartado 3.3.3). De hecho, Cygnus es un agente Flume, cuya arquitectura básica mostrada en la Figura 5.7 se compone de una **source** (*HttpSource*) a cargo de recibir los datos del *Orion Context Broker*, un **channel** (*Memory Channel*) donde la *source* pone los datos una vez transformados en eventos Flume, y un **sink** (OrionHDFS Sink), que extrae los eventos Flume del canal para persistir los datos en un sistema de almacenamiento externo.

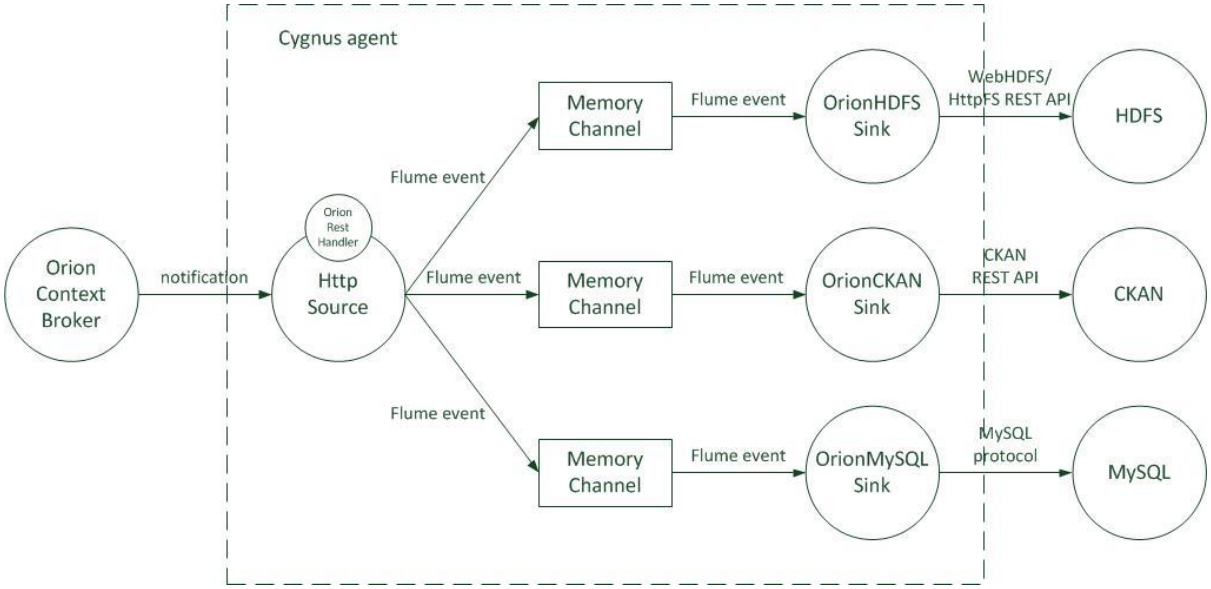


Figura 5.7. Arquitectura básica de Cygnus.

En este proyecto se ha utilizado la versión 0.7.1 de Cygnus que permite la persistencia de los datos del Orion Context Broker en los siguientes formatos:

- **HDFS**: sistema de ficheros distribuidos de Hadoop, explicado en apartado 3.3.2 y utilizado en el presente proyecto de investigación.
- **MySQL**: el gestor conocido de bases de datos relacionales.
- **CKAN**: plataforma Open Data.

Cuando se intenta mejorar el rendimiento de Cygnus aparecen arquitecturas avanzadas. Como se puede observar en la Figura 5.7, la configuración básica de Cygnus trata de una *source* escribiendo eventos Flume en un *channel* donde un *sink* consume estos eventos. Esto puede ser claramente trasladado a una configuración múltiple de *sinks* ejecutándose en paralelo donde aparece una variedad de nuevas configuraciones.

Una posible configuración es **multiple sinks, single channel**, sin embargo usualmente muestra una importante desventaja, especialmente si los eventos son consumidos por los *sinks* muy rápido: los *sinks* tienen que competir por el canal.

La desventaja mencionada anteriormente puede ser solucionada configurando un *channel* por cada *sink*, evitando la competición por el *channel*, es decir: **multiple sinks, multiple channels**. Sin embargo, cuando múltiples *channels* son utilizados para un mismo almacenamiento, entonces debe existir algún tipo de *dispatcher* que decida qué *channels* reciben una copia de los eventos. Además se establecen las siguientes condiciones:

- Se desea que los eventos sean replicados por cada tipo de almacenamiento configurado.
- Dentro de un tipo de almacenamiento, no se permite la replicación de los eventos Flume.
- Se desea que el criterio de dispatching esté basado en un comportamiento round-robin.

El selector *RoundRobinChannelSelector* cumple con estos requisitos. La Figura 5.8 ilustra la arquitectura avanzada descrita anteriormente.

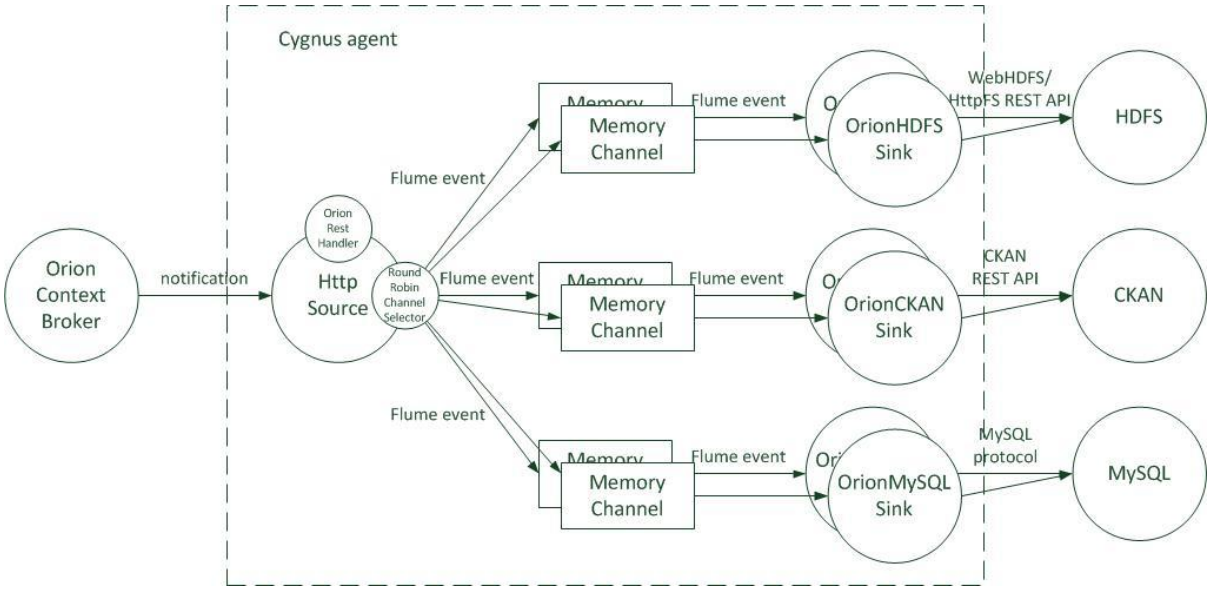


Figura 5.8. Arquitectura avanzada de Cygnus.

6 Prototipo de Laboratorio

El propósito de este capítulo es describir los aspectos relativos al diseño, implementación, instalación, configuración, despliegue y simulación del prototipo del sistema estudiado en este Trabajo Fin de Grado.

6.1 Diseño

La Figura 6.1 ilustra el diseño del prototipo inicial, el cual está compuesto por los *Generic Enablers* descritos en el apartado 5.1. FI-WARE Data/Context Management. El fin de este prototipo es aprender la metodología a seguir en la plataforma FI-WARE para desarrollar aplicaciones en el ámbito de Internet de las Cosas. Además, permite trabajar con tres componentes fundamentales del capítulo *Data/Context Management* de FIWARE así como desarrollar y evaluar nuevas vías para mejorar su eficiencia como la incorporación de varios canales en Cygnus junto con el algoritmo de selección Round Robin.

Más concretamente, el sistema propuesto simula el comportamiento de una red sensores por medio de la implementación de las operaciones provistas por el *Context Broker*. Dicha implementación se traduce en una aplicación (**`TestBench`**) que se ejecuta en un PC del laboratorio DSIE. Como resultado, M sensores virtuales (*Context Producers*) envían información de contexto (*context information*) a una instancia del *Context Broker* desplegada en FIWARE. Cada vez que se produce la actualización de la información de contexto asociada a una entidad (*entity* = virtual sensor) registrada en la base de datos del *Context Broker*, se envía una notificación a Cosmos. Esta notificación se realiza a través del inyector Cygnus. De esta forma, se genera un histórico de los datos del sistema.

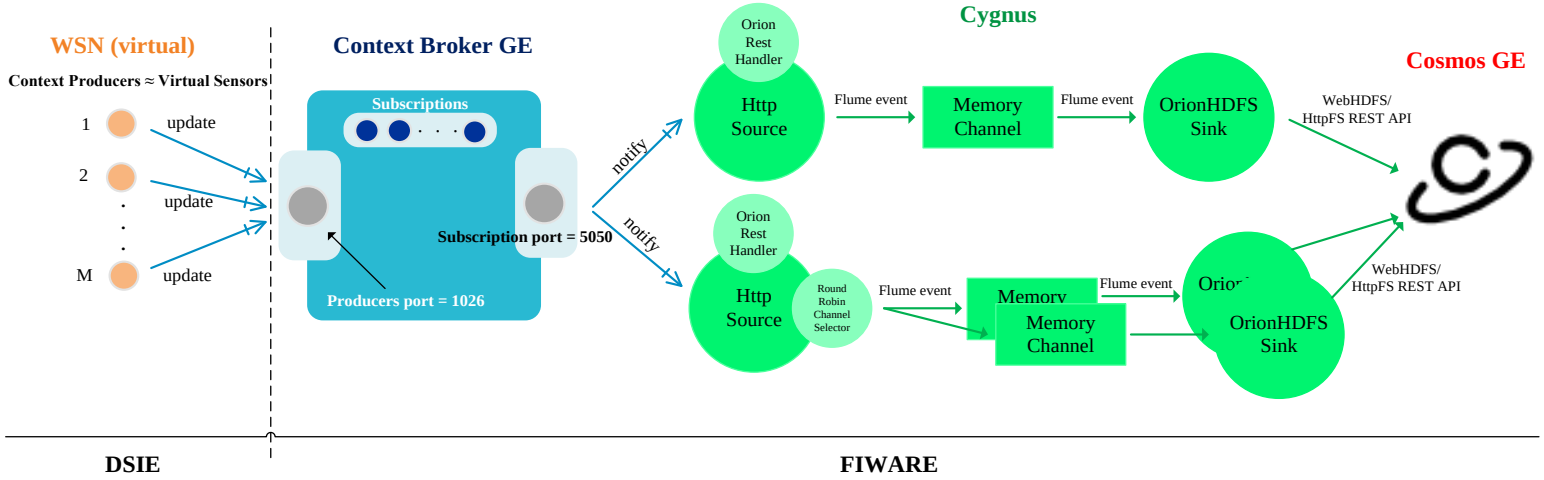


Figura 6.1. Arquitectura del Prototipo Inicial.

A continuación, se supone que el Prototipo diseñado solo simula un sensor para explicar el diagrama de secuencia (Figura 6.2). En el proceso de inicialización, la aplicación **`TestBench`** solicita al **`Broker`** tanto la creación de una entidad asociada al sensor simulado (operación **`append`**) como la suscripción a la información de contexto de a dicha entidad (operación **`subscribe`**). Posteriormente, la aplicación **`TestBench`** solicita al **`Broker`** la actualización de la información de contexto de la entidad creada anteriormente (operación **`update`**). Esta operación se realiza cada ΔT ms (parámetro establecido en **`TestBench`**).

El **`Broker`** envía una notificación al agente Cygnus (operación **`notify`**), específicamente a su **`HTTPSource`**. Esta *source*, a través de **`OrionRestHandler`**, analiza la notificación y produce un evento Flume que se almacena en el canal configurado (**`HdfsChannel`**). Respecto a la persistencia HDFS, el evento Flume es adquirido del *channel* y se crea un fichero HDFS con los datos del evento asociado a la entidad del sistema. Finalmente, se comprueba la existencia del directorio HDFS. Si existe, los datos del evento son añadidos. Si no existe, se crea el fichero junto a su directorio HDFS así como su respectiva tabla Hive.

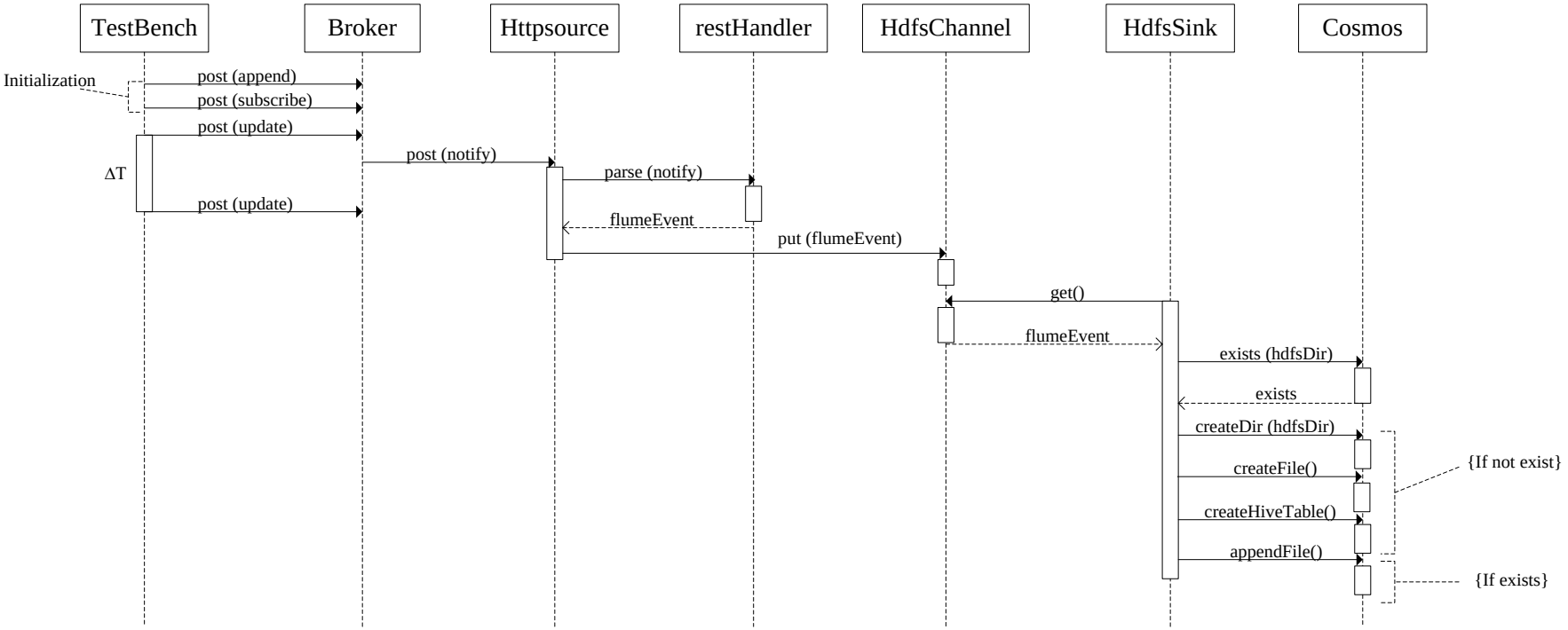


Figura 6.2. Diagrama de secuencia del Prototipo Inicial.

En resumen, el Prototipo de Pruebas diseñado permite evaluar el rendimiento del *Context Broker* debido a que produce las siguientes medidas cuantitativas: retardo entre el envío de datos por el simulador de sensores y su recepción a la salida del *Broker* (sensor virtual-*source* de Cygnus) y relación de paquetes enviados/recibidos por este componente que constituye el *core* del capítulo FIWARE *Data/Context Management*.

Tras observar el correcto funcionamiento del prototipo inicial, se realizan las siguientes modificaciones para estudiar el rendimiento del *Generic Enabler Context Broker*:

1. Equipo del laboratorio DSIE: (1) instalación de sistema operativo CentOS 6.5, de la versión 0.7.1 del inyector Cygnus y de un servidor VPN. (2) Configuración de Cygnus y del entorno de desarrollo Eclipe con proyecto **`TestBenchCl82ient`** que incorpora la interfaz gráfica del simulador de sensores y la clase **`Cliente.java`**. (3) Sincronización **`Cliente.java-Servidor.java`**.
2. Extensión a la red de ordenadores del laboratorio LSI-1 (12 equipos): (1) instalación de cliente VPN y configuración de proyecto **`TestBenchServer`** en el entorno de desarrollo Eclipse. (2) Sincronización **`Cliente.java-Servidor.java`**.
3. El GE Cosmos no es un componente de este prototipo.

La Figura 6.3 ilustra la arquitectura del prototipo de pruebas que incopora las modificaciones mencionadas anteriormente. En dicha arquitectura, la interconexión de las redes DSIE, FIWARE y LSI1 se realiza a través de Internet. Además, se crea una red VPN (equipos DSIE y LSI1) mediante el *software* OpenVPN.

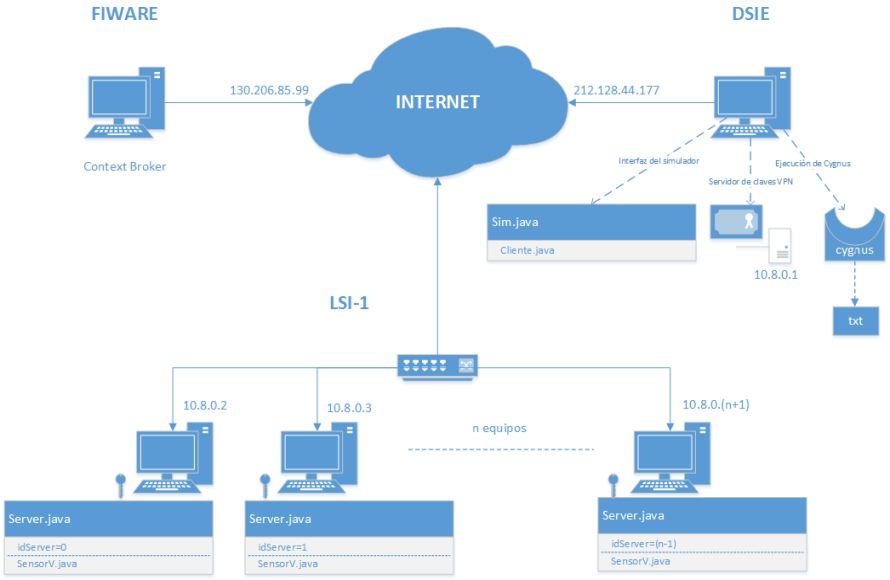


Figura 6.3. Arquitectura del Prototipo de Pruebas.

Inicialmente, en el equipo del DSIE se lanza la interfaz gráfica del simulador (**`Sim.java`**) en la que se establecen los diferentes parámetros de la simulación y se generan las solicitudes de creación y suscripción de/a entidades en el *Context Broker*. Posteriormente, la clase **`Cliente.java`** envía una solicitud de arranque de simulación a cada uno de los equipos que escuchan por el puerto 8080 y que pertenecen a la red VPN.

Para que los equipos de la red VPN (LSI1) puedan recibir los parámetros de simulación es necesario que la clase **`Server.java`** haya sido ejecutada ya que permite permanecer a la espera de solicitudes de simulación por parte de la máquina del DSIE. Así, cada instancia de la clase **`Server.java`** tiene un identificador único (**`idServer`**) que, a su vez, permite la actualización de ciertas entidades del *Context Broker*. Cuando reciben una solicitud, procesan dichos parámetros y pasan a simular el comportamiento de un número de sensores generando

y enviando datos de contexto al *Context Broker*.
Cabe mencionar que, cada equipo del LSI1 puede ejecutar uno o varios procesos. Se entiende por proceso una instancia de Eclipse con un identificador único (`idServer`). Gracias a esta característica, se pretende conseguir aumentar la simulación del número de dispositivos (*Context Producers*).

El *Context Broker* recibe los datos de contexto generados por la red virtual de sensores que actualizan las entidades del sistema. Además, se encarga de enviar a Cygnus la información de contexto anterior asociada a dichas entidades.

Finalmente, el agente Cygnus, y más concretamente la clase `HTTPSOURCE`, recibe las notificaciones del *Context Broker*. A continuación, la clase `OrionRestHandler` analiza dichas notificaciones y almacena la información relevante en un *buffer*. Cuando finaliza la simulación la información pasa a un fichero de texto formateado para su posterior estudio mediante la herramienta *software* RStudio.

Por lo dicho anteriormente, el diagrama de secuencia del Prototipo de Pruebas tiene la misma estructura que el de la Figura 6.2, pero finaliza con el análisis de la notificación por la clase `OrionRestHandler` para su almacenamiento en un *buffer* o, en un fichero de texto al final de la simulación.

6.2 Instalación, Configuración y Despliegue

6.2.1 Context Broker.

El *Generic Enabler Context Broker* es utilizado en los dos prototipos estudiados en este Trabajo Fin de Grado.

6.2.1.1 Procedimiento en FILAB

Tanto para desplegar una instancia del *Context Broker* como para utilizar cualquier servicio de la plataforma FIWARE, en primer lugar es necesario el registro en FILAB a través del siguiente enlace https://account.lab.fiware.org/.

A continuación, se ingresa en la cuenta FILAB creada al introducir las credenciales configuradas en el proceso de registro. Al hacer click en la pestaña *Cloud*, aparece la ventana de la Figura 6.4 en la que se puede lanzar, modificar y eliminar las diferentes instancias creadas en la plataforma.

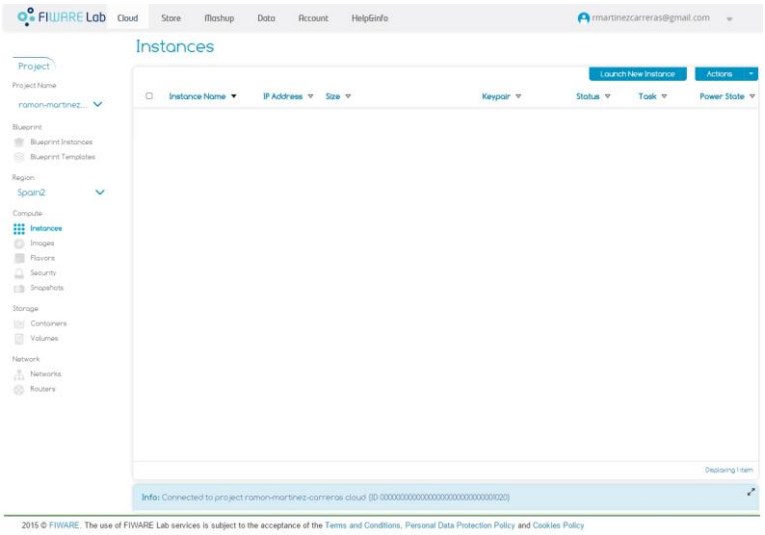


Figura 6.4. Ventana *Cloud* en FILAB.

Antes de lanzar la instancia del *Context Broker*, es necesario realizar las siguientes acciones: (1) creación de una clave en formato .pem, (2) creación de un grupo de seguridad que incorpore los puertos que utilizará la instancia a desplegar y (3) creación de una IP pública.

Para ello, en la ventana *Cloud*, hacer click sobre *Security* → *Compute* → *Keypairs*. A continuación, hacer click sobre *Create keypair* y añadir un nombre cualquiera con el fin de identificar la futura instancia asociada a dicha clave.
Tras ello, clickar en *Create Keypair* (Figura 6.5). Como resultado de esta acción aparece un cuadro de diálogo que invita a proceder a la descarga de la nueva clave generada en formato PEM. Esta descarga debe efectuarse para posteriormente, poder establecer comunicación con la instancia a través del *software* PuTTY.

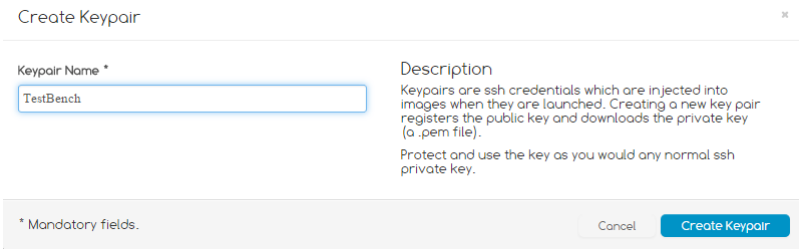


Figura 6.5. Creación de *Keypair* en FILAB.

Posteriormente, en la misma sección (*Security*) hacer click sobre *Security Groups* → *Create Security Group*. En la ventana emergente, especificar un nombre y una descripción cualquiera que identifique la instancia a crear. Seguidamente, pulsar sobre *Create Security Group*.

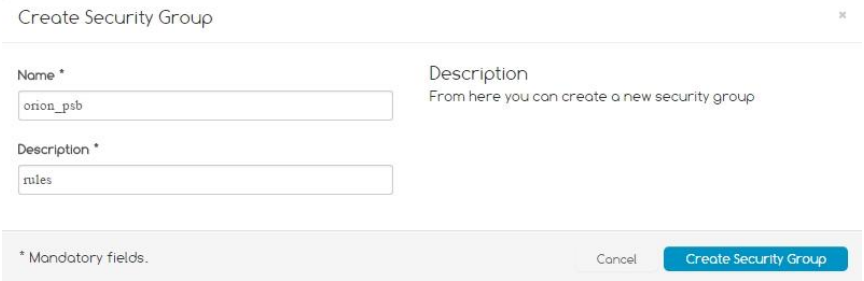


Figura 6.6. Creación de *Security Group* en FILAB.

Tras unos instantes, aparece el nuevo grupo de seguridad creado que permite especificar y abrir los puertos de la instancia a desplegar. Para realizar estas acciones, hay que seleccionar

el nuevo grupo de seguridad y, posteriormente hacer click sobre *Actions* → *Edit Rules*. Las reglas que deben ser añadidas son mostradas en la Figura 6.7.

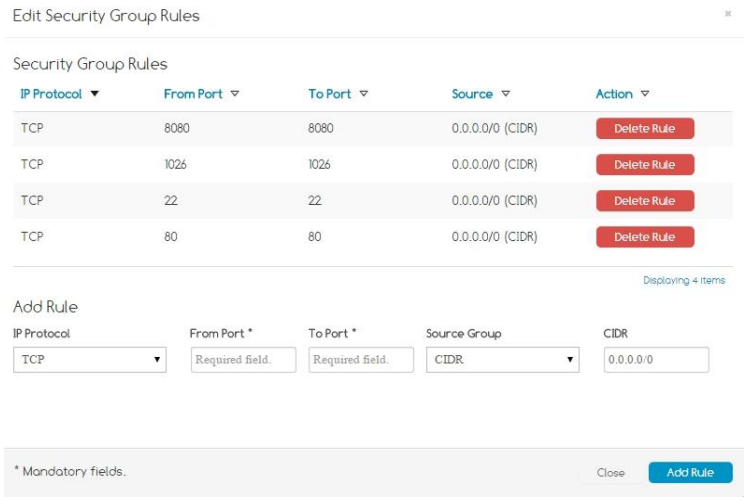


Figura 6.7. Configuración de *Security Group*.

Para finalizar en la sección *Security*, hacer click sobre la pestaña *Floating IPs* para crear una IP pública disponible que, posteriormente será asociada a la nueva instancia. A continuación, click sobre *Allocate IP to Project*. Finalmente, elegir una de las opciones contempladas en la pestaña *Pool* y click sobre *Allocate IP*.

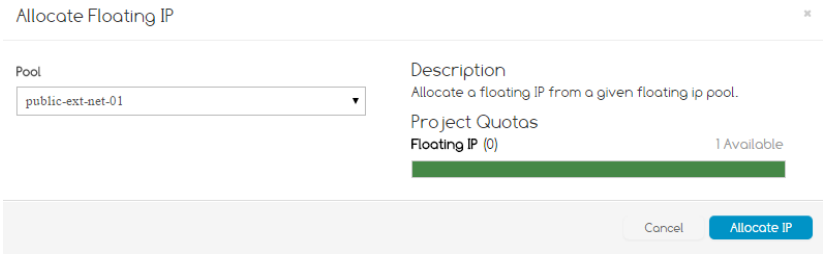


Figura 6.8. Creación de IP pública en FILAB.

La Figura 6.9 ilustra las configuraciones realizadas anteriormente en la sección *Security*.

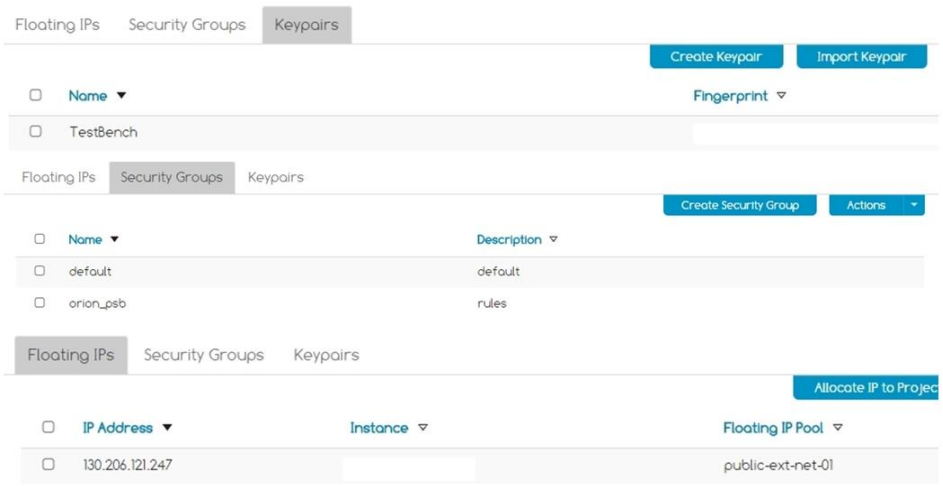


Figura 6.9. Configuración final de la sección *Security* en FILAB.

Una vez realizada dicha configuración, se procede a configurar y lanzar la instancia del *Context Broker*. Para ello, hacer click sobre la pestaña *Images* del apartado *Compute*, buscar la imagen denominada orion-psb-image-R4.2 y hacer click sobre *Launch*.

Aunque se trata de un proceso que se puede realizar de manera intuitiva, a continuación se describe la configuración de la instancia a desplegar. En el primer paso de este proceso, se especifica el nombre de la instancia así como su tamaño.

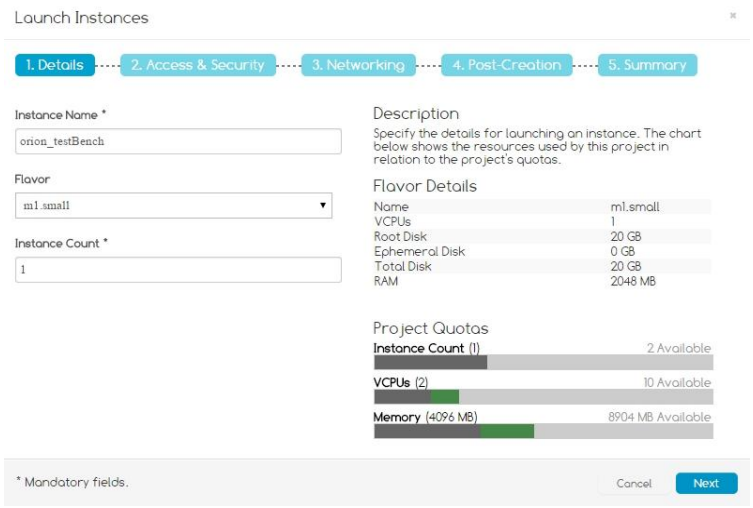


Figura 6.10. Configuración de la instancia (1).

En el segundo paso, se hace referencia al *Keypair* y al *Security Group* creados anteriormente.

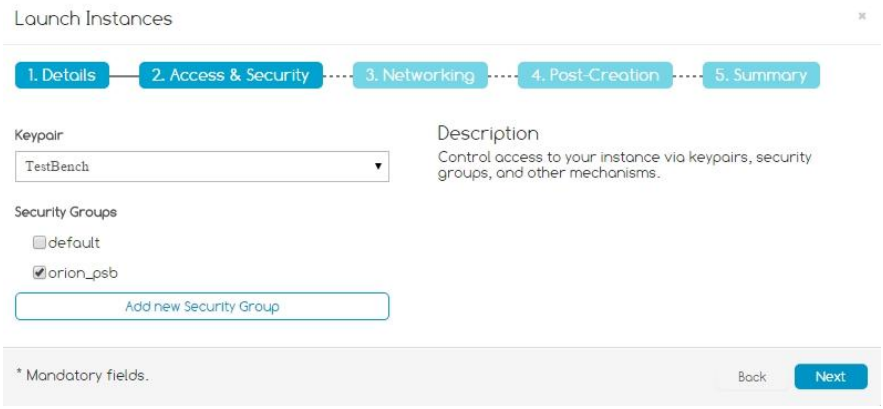


Figura 6.11. Configuración de la instancia (2).

En el tercer paso, hay que seleccionar la red de la instancia a desplegar. Para ello, arrastrar la opción *shared-net* desde la zona *Available Networks* hasta *Selected Networks*.

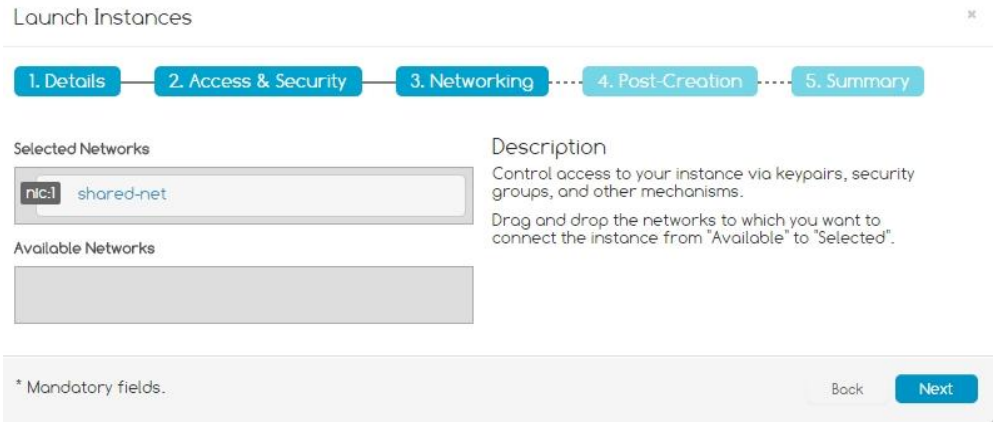


Figura 6.12. Configuración de la instancia (3).

Tras pulsar en Next, aparece un *script* con la configuración que ejecutará la máquina por defecto. Aunque se permite la modificación de dicha configuración, no se realiza ningún cambio en la misma.

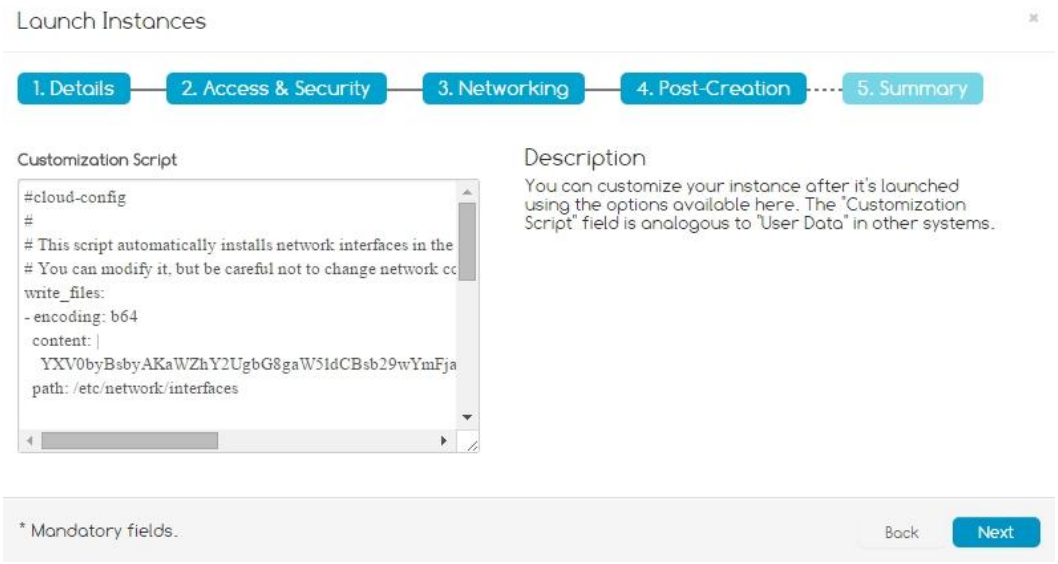


Figura 6.13. Configuración de la instancia (4)

Por último, aparece un resumen de los parámetros seleccionados en el proceso de configuración. Pulsar *Launch Instance* para terminar.

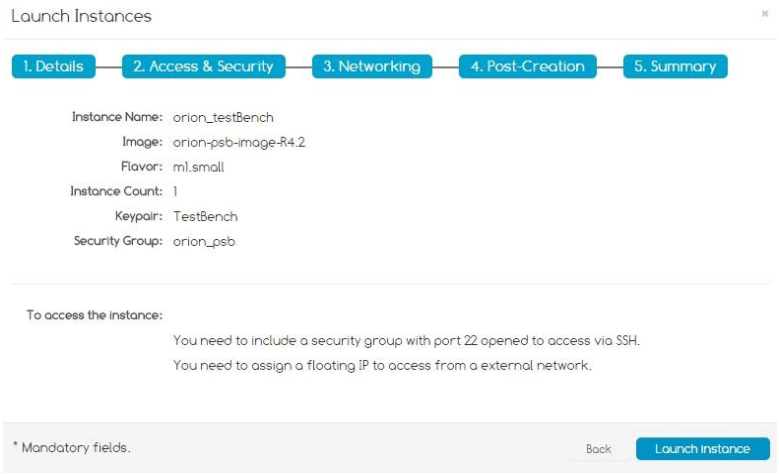


Figura 6.14. Configuración de la instancia (5).

Tras unos instantes, se inicia el proceso de creación de la instancia con los parámetros de configuración establecidos. Cuando este ha finalizado, se realiza el despliegue de la instancia, apareciendo en el apartado *Instances*, de manera similar a la mostrada por la Figura 6.15.

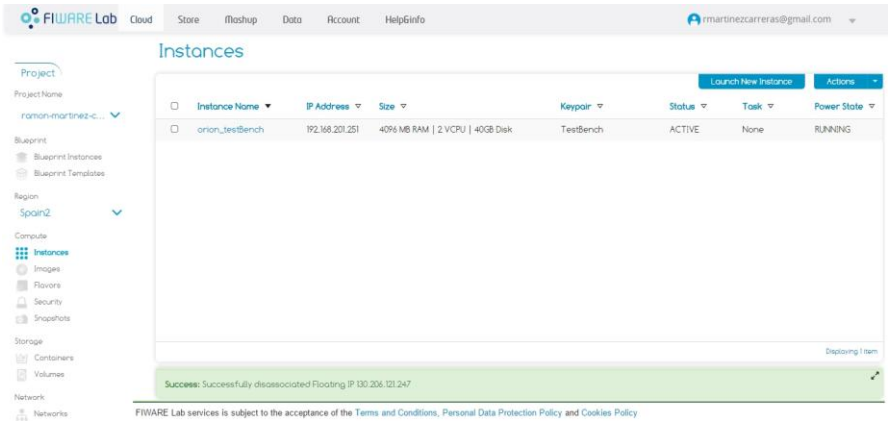


Figura 6.15. Instancia desplegada en FILAB.

Llegados a este punto solo resta asociar una IP pública a la instancia desplegada para poder establecer comunicación con la misma, ya que en estos momentos tiene asignada una IP privada, por lo que sería imposible su conexión remota.

Para ello, pulsar nuevamente sobre el apartado *Security* y en la pestaña *Floating IPs* seleccionar la IP pública creada previamente. Tras esto, pulsar sobre *Actions* → *Associate IP*. En la ventana emergente, seleccionar en el desplegable la instancia recientemente creada y su IP privada. Finalmente, pulsar sobre *Associate IP*.

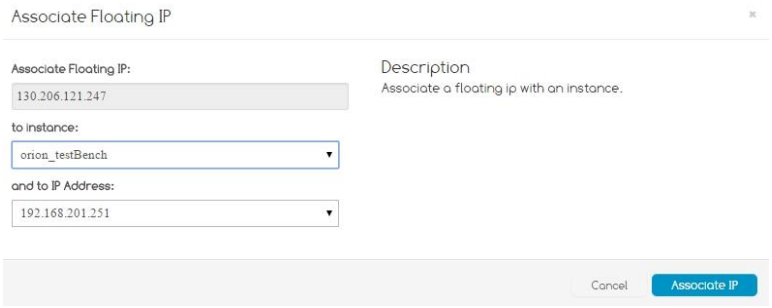


Figura 6.16. Asociación de IP pública.

6.2.1.2 Configuración de Herramienta PuTTY

PuTTY solo acepta como medio de seguridad para la conexión con máquinas remotas las claves generadas bajo el formato .ppk (*PuTTY Private Key*). La clave que se obtiene de la creación de la instancia en FILAB es de formato .pem. Por este motivo, es necesaria la utilización del *software* PuTTYgen para realizar la conversión entre formatos.

Al ejecutar PuTTYgen, aparece una ventana que permite la creación de claves desde cero o bien, la carga de claves desde un fichero. Así, pulsar sobre *Load* y seleccionar la clave descargada en el proceso de generación de Keypair explicado anteriormente.

Posteriormente, hay que guardar la clave en formato .ppk pulsando sobre *Save private key* y seleccionando una ruta. Esta clave es secreta, por lo que es conveniente almacenarla en un lugar seguro y no compartirla ni publicarla.

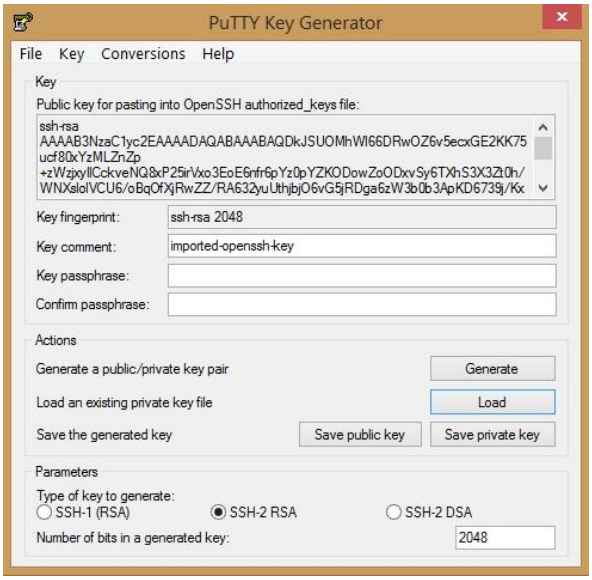


Figura 6.17. Conversión entre formatos con PuTTYgen.

Para realizar la conexión con la máquina virtual (Figura 6.18) de la instancia desplegada, hay que ejecutar PuTTY, especificando su IP pública en el apartado *Host Name* y su puerto (22).

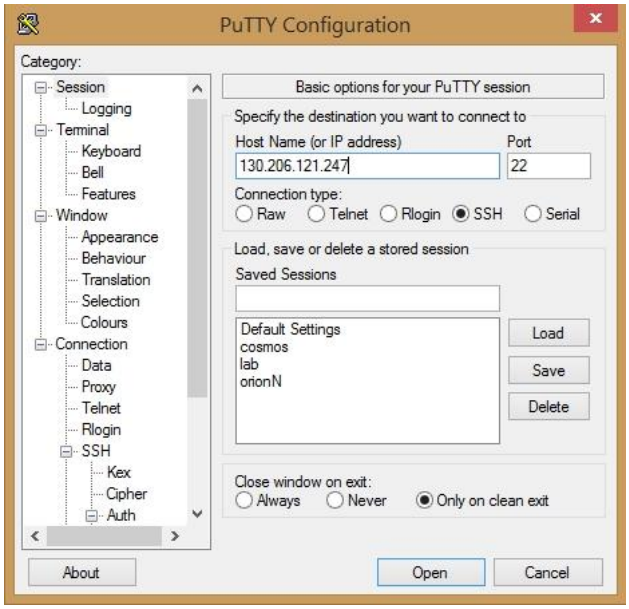


Figura 6.18. Conexión con PuTTY (1).

Finalmente, desplegar el menú *Connection* del panel *Category* y seleccionar el submenú SSH → Auth. En este submenú, proceder a la carga de la clave privada con formato .ppk generada mediante PuTTYgen y pulsar sobre *Open*.

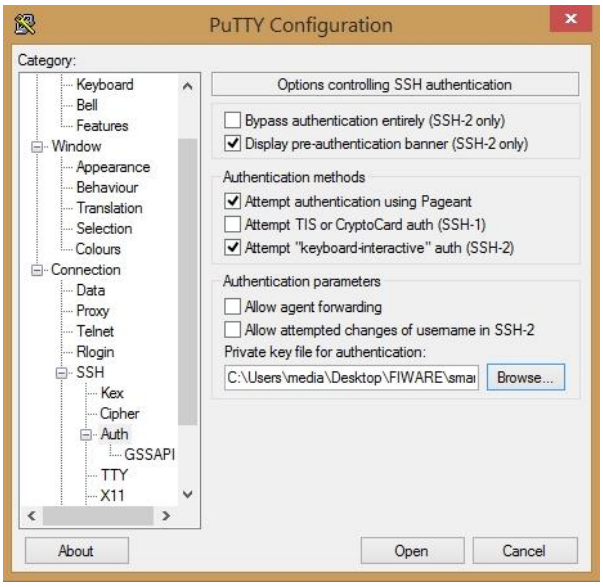


Figura 6.19. Conexión con PuTTY (2).

Una vez que se establece conexión, aparece un terminal en el que hay que identificarse como root. Posteriormente, aparece la línea de comandos del broker.

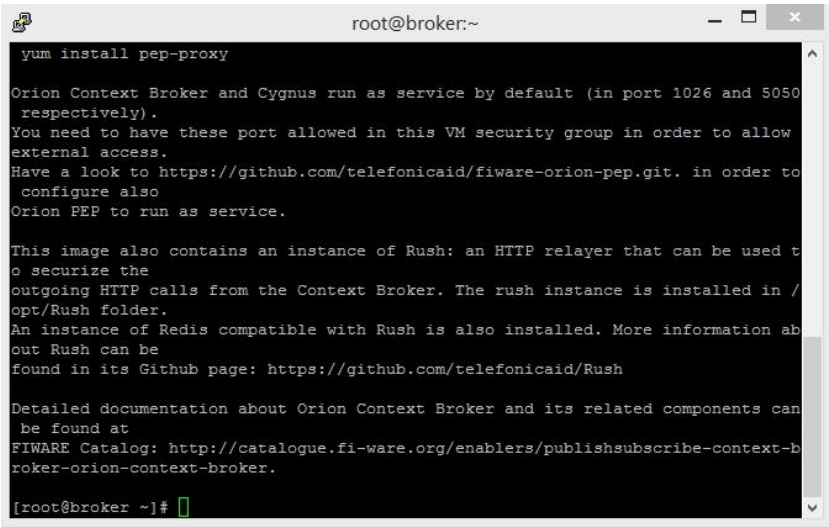


Figura 6.20. Terminal del *Context Broker*.

6.2.2 Cosmos

El GE Cosmos se ha empleado en el Prototipo Inicial. Cuando se crea una cuenta en FI-LAB, automáticamente se crea una cuenta en Cosmos. Para verificar la existencia de esta cuenta, es neceario acceder al siguiente enlace http://cosmos.lab.fi-ware.org/cosmos-gui/ en el que hay que introducir las credenciales de la cuenta FI-LAB (Figura 6.21).

Cabe mencionar que existen métodos alternativos de instalación de este GE en máquinas locales que permiten una administración completa de Cosmos, sin embargo no son estudiados en este trabajo de investigación.

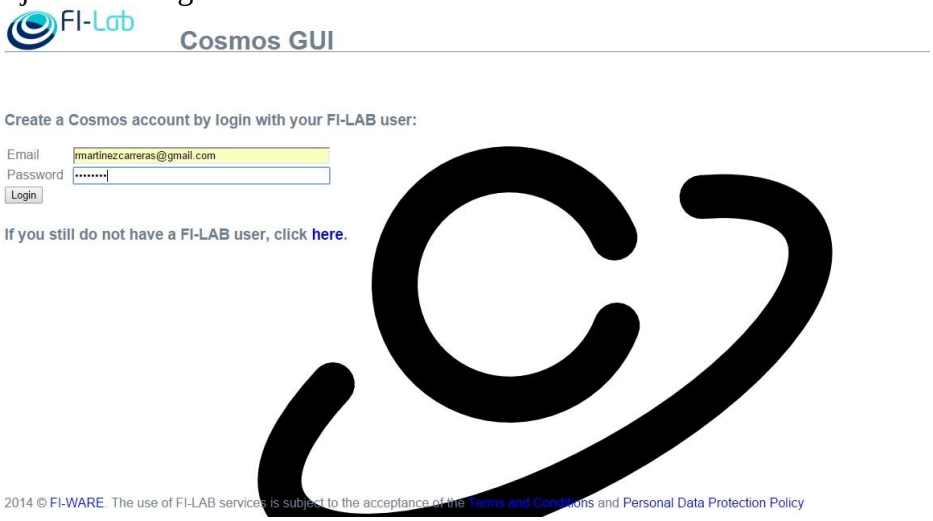


Figura 6.21. Login en Cosmos.

Tras el login, se notifica en pantalla acerca de la existencia del usuario en el sistema (Figura 6.22). Además, se muestran una serie de mecanismos de acceso a través de ssh que serán utilizados en la consola del *Orion Context Broker* para acceder a Cosmos.

Figura 6.22. Parámetros de Cosmos

6.2.3 Cygnus

6.2.3.1 Instalación

El inyector Cygnus es utilizado en los dos prototipos estudiados en este Trabajo Fin de Grado. Antes de proceder a la instalación de Cygnus, es necesario instalar Java SDK y Apache Maven (prerrequisitos). Para instalar Java SDK, simplemente se debe ejecutar el siguiente comando en una máquina CentOS.

```
sudo yum install java-1.6.0-openjdk-devel
```

Una vez instalado el JDK, hay que exportar la variable de entorno JAVA_HOME. Para ello, se debe ejecutar el siguiente comando:

```
export JAVA_HOME=/usr/lib/jvm/java-1.6.0-openjdk.x86_64
```

Para que esta acción sea permanente, editar */root/.bash_profile*, incluyendo la línea de arriba.

Apache Maven se descarga del enlace maven.apache.org y se instala en una carpeta elegida bajo criterio propio. El directorio elegido es */usr/local*
Los diferentes comandos a ejecutar para su instalación satisfactoria son:

```
wget http://www.eu.apache.org/dist/maven/maven-3/3.2.5/binaries/apache-maven-3.2.5-bin.tar.gz
tar xzvf apache-maven-3.2.5-bin.tar.gz
mv apache-maven-3.2.5 /usr/local
```

Llegados a este punto, se tienen las herramientas necesarias para la instalación de Cygnus. Si se quiere instalar Cygnus en una máquina CentOS proporcionada por FILAB, es necesario saber que ya existe una versión de Cygnus preinstalada. Por este motivo, en primer lugar hay que eliminar la versión existente e instalar una versión actual. En el caso de que se quiera instalar Cygnus en una máquina CentOS propia, este paso puede saltarse. Mencionada esta cuestión, para eliminar Cygnus se ejecuta:

```
sudo yum remove cygnus
```

Una vez eliminado Cygnus, se añade el repositorio FIWARE necesario:

```
sudo cat > /etc/yum.repos.d/fiware.repo <<EOL
[Fiware]
name=FIWARE repository
```

```
baseurl=http://repositories.testbed.fi-ware.eu/repo/rpm/x86_64/
gpgcheck=0
enabled=1
EOL
```

Finalmente, se procede a la nueva instalación mediante el comando:

```
sudo yum install cygnus
```

Se instalará la versión 0.7.1.

6.2.3.2 Configuración

La configuración de Cygnus se realiza editando el fichero **`agent_test.conf`**. Así, en este fichero se establece el número, tipo y disposición de los componentes de la arquitectura del inyector Cygnus (*sources*, *channels* y *sinks*).

El fichero se edita en la ruta **`/usr/cygnus/conf/agent_test.conf`**. A continuación, se describe el proceso de configuración de los componentes:

- **Configuración de source/s (fuente/s de datos):** especifica la configuración de la/s *source/s*. Los parámetros más importantes a indicar son el nombre del *channel* asociado, el tipo de *source* (HTTPSource), el puerto (5050), el manejador de los datos de contexto (*OrionRestHandler*) y el *TimeToLive*.
- **Configuración de channel/s (canale/s):** especifica el tipo, la capacidad y la capacidad de transacción que tendrá el/los *channel/s* (canal/es).
- **Configuración de sink/s (sumidero/s):** especifica la información referente al consumidor de los datos. En los protipos diseñados, los datos pueden ser consumidos por Cosmos o almacenarse de manera local. La variable que especifica el destino de los datos es *attr_persistence*. En el caso de que se desee almacenar en Cosmos, también hay que especificar el *host* y el puerto de destino así como los parámetros necesarios para la autenticación en Cosmos (*username* y *password*). Además, de manera independiente al destino de los datos, hay que especificar el canal desde el que se consumen eventos y el tipo de sumidero (OrionHDFSSink).

A continuación, se muestra la principal configuración del fichero que ha sido utilizada en la mayor parte de las simulaciones (una *source*, un *channel* y un *sink*).

```
cygnusagent.sources = http-source
cygnusagent.sinks = hdfs-sink
cygnusagent.channels = notifications

cygnusagent.sources.http-source.channels = notifications
cygnusagent.sources.http-source.type = org.apache.flume.source.http.HTTPSource
cygnusagent.sources.http-source.port = 5050
cygnusagent.sources.http-source.handler=
es.tid.fiware.fiwareconnectors.cygnus.handlers.OrionRestHandler
cygnusagent.sources.http-source.handler.orion_version = 0\.22\.*
cygnusagent.sources.http-source.handler.notification_target = /notify

cygnusagent.sinks.hdfs-sink.cosmos_host = 130.206.80.46
cygnusagent.sinks.hdfs-sink.cosmos_port = 14000
cygnusagent.sinks.hdfs-sink.cosmos_default_username = rmartinezcarreras
cygnusagent.sinks.hdfs-sink.cosmos_default_password = 01031983
cygnusagent.sinks.hdfs-sink.cosmos_dataset = /user/rmartinezcarreras/sensores
cygnusagent.sinks.hdfs-sink.hdfs_api = httpfs
cygnusagent.sinks.hdfs-sink1.attr_persistence = local

cygnusagent.sources.http-source.handler.default_service = def_serv
cygnusagent.sources.http-source.handler.default_service_path = def_servpath
cygnusagent.sources.http-source.handler.events_ttl = 2
```

```
cygnusagent.sources.http-source.interceptors = ts de
cygnusagent.sources.http-source.interceptors.ts.type = timestamp
cygnusagent.sources.http-source.interceptors.de.type =
es.tid.fiware.fiwareconnectors.cygnus.interceptors.DestinationExtractor$Builder
cygnusagent.sources.http-source.interceptors.de.matching_table =
/usr/cygnus/conf/matching_table.conf

cygnusagent.channels.notifications.type = memory
cygnusagent.channels.notifications.capacity = 1000
cygnusagent.channels.notifications.transactionCapacity = 100

cygnusagent.sinks.hdfs-sink.channel = notifications
cygnusagent.sinks.hdfs-sink.type= es.tid.fiware.fiwareconnectors.cygnus.sinks.OrionHDFSSink
```

Tal y como se describe en el apartado 5.1.5.1, existe la posibilidad de aumentar el número de *channels* y *sinks* con el fin de mejorar la eficiencia del sistema. Así, aparecen arquitecturas avanzadas del inyector Cygnus como la de múltiples canales y múltiples sumideros con selector de canal *Round Robin*. El siguiente fichero modela este tipo de arquitectura al configurar dos canales y dos sumideros, además del selector *Round Robin*.

```
cygnusagent.sources = http-source
cygnusagent.sinks = hdfs-sink1 hdfs-sink2
cygnusagent.channels = notifications1 notifications2

cygnusagent.sources.http-source.type = org.apache.flume.source.http.HTTPSource
cygnusagent.sources.http-source.channels = notifications1 notifications2
cygnusagent.sources.http-source.selector.type =
es.tid.fiware.fiwareconnectors.cygnus.channelselectors.RoundRobinChannelSelector
cygnusagent.sources.http-source.selector.storages = 1
cygnusagent.sources.http-source.selector.storages.storage1 = notifications1,notifications2
cygnusagent.sources.http-source.port = 5050
cygnusagent.sources.http-source.handler                                                    =
es.tid.fiware.fiwareconnectors.cygnus.handlers.OrionRestHandler
cygnusagent.sources.http-source.handler.orion_version = 0\.22\.*
cygnusagent.sources.http-source.handler.notification_target = /notify
cygnusagent.sources.http-source.handler.events_ttl = 1
cygnusagent.sources.http-source.interceptors = ts de
cygnusagent.sources.http-source.interceptors.ts.type = timestamp
cygnusagent.sources.http-source.interceptors.de.type =
es.tid.fiware.fiwareconnectors.cygnus.interceptors.DestinationExtractor$Builder
cygnusagent.sources.http-source.interceptors.de.matching_table =
/usr/cygnus/conf/matching_table.conf
cygnusagent.sinks.hdfs-sink1.cosmos_host = 130.206.80.46
cygnusagent.sinks.hdfs-sink1.cosmos_port = 14000
;cygnusagent.sinks.hdfs-sink1.attr_persistence = local
cygnusagent.sinks.hdfs-sink1.cosmos_default_username = rmartinezcarreras
cygnusagent.sinks.hdfs-sink1.cosmos_default_password = 01031983
cygnusagent.sinks.hdfs-sink1.cosmos_dataset = /user/rmartinezcarreras/sensores
cygnusagent.sinks.hdfs-sink1.hdfs_api = httpfs
cygnusagent.sinks.hdfs-sink1.channel = notifications1
cygnusagent.sinks.hdfs-sink1.type =
es.tid.fiware.fiwareconnectors.cygnus.sinks.OrionHDFSSink

cygnusagent.sinks.hdfs-sink2.cosmos_host = 130.206.80.46
cygnusagent.sinks.hdfs-sink2.cosmos_port = 14000
;cygnusagent.sinks.hdfs-sink2.attr_persistence = local
cygnusagent.sinks.hdfs-sink2.cosmos_default_username = rmartinezcarreras
cygnusagent.sinks.hdfs-sink2.cosmos_default_password = 01031983
cygnusagent.sinks.hdfs-sink2.cosmos_dataset = /user/rmartinezcarreras/sensores
cygnusagent.sinks.hdfs-sink2.hdfs_api = httpfs
cygnusagent.sinks.hdfs-sink2.channel = notifications2
cygnusagent.sinks.hdfs-sink2.type =
es.tid.fiware.fiwareconnectors.cygnus.sinks.OrionHDFSSink

cygnusagent.channels.notifications1.type = memory
```

```
cygnusagent.channels.notifications1.capacity = 100
cygnusagent.channels.notifications1.transactionCapacity = 100

cygnusagent.channels.notifications2.type = memory
cygnusagent.channels.notifications2.capacity = 100
cygnusagent.channels.notifications2.transactionCapacity = 100
```

La configuración anterior es una muestra de la escalabilidad del sistema pudiéndose aumentar el número de canales y sumideros.

6.2.3.3 Compilación

Los pasos para compilar el código de Cygnus son los siguientes:

1. Copiar carpeta **src** a **/usr/appRest/fiware-cygnus/flume**.
2. Eliminar carpeta **target** en **/usr/appRest/fiware-cygnus/flume/target**.
3. **cd /usr/appRest/fiware-cygnus/flume**.
4. **/usr/local/apache-maven-3.0.5/bin/mvn clean compile exec:exec assembly:single**.
5. **cp target/cygnus<>.jar /usr/cygnus/plugins.d/cygnus/lib**.

6.2.4 Instalación y configuración VPN

6.2.4.1 Habilitar repositorio EPEL

La instalación/configuración de la red VPN se utiliza en el Prototipo de Pruebas. Para crear la red virtual, se utiliza el *software* OpenVPN, donde el servidor se aloja en el equipo ubicado en el DSIE (Figura 6.3). En primer lugar, hay que preparar el repositorio EPEL para descargar el *software* OpenVPN. Para ello, se ejecuta la siguiente relación de comandos [42]:

```
wget https://fedoraproject.org/static/0608B895.txt
mv 0608B895.txt /etc/pki/rpm-gpg/RPM-GPG-KEY-EPEL-6
rpm --import /etc/pki/rpm-gpg/RPM-GPG-KEY-EPEL-6
```

Tras esta acción, verificar que el repositorio se ha instalado correctamente:

```
rpm - qa gpg*
gpg-pubkey-0608b895-4bd22942
```

Finalmente habilitar EPEL:

```
rpm -ivh epel-release-6-5.noarch.rpm
```

6.2.4.2 Generación de claves

El siguiente paso es la descarga de los componentes necesarios para la generación de claves, es decir, la descarga del *software* OpenVPN. La instalación genera una serie de ficheros en el directorio **/usr/share/easy-rsa/2.0**, ficheros que deberán ser copiados a una carpeta a elegir cuyo rol será el de directorio de trabajo. Una vez en el mismo, se crea un directorio denominado **keys**, donde se alojarán las claves creadas en el proceso [43] [44].

```
yum install openvpn easy-rsa
cp -ai /usr/share/easy-rsa/2.0 ~/easy-rsa
cd ~/easy-rsa
mkdir keys
```

Tras esta acción, han de generarse la clave y el certificado para la Autoridad Certificadora (CA). Para ello, editar el archivo **vars** mediante el siguiente comando:

```
gedit vars
```

y añadir la siguiente información:

```
export KEY_SIZE=2048
export KEY_DIR=”/root/easy-rsa/keys”
export KEY_COUNTRY=”ES”
export KEY_PROVINCE=”MU”
export KEY_CITY=”Cartagena”
export KEY_ORG=”UPCT”
export KEY_EMAIL=”jlrg.jose@gmail.com”
```

La variable **KEY_DIR** especifica la ruta donde están alojadas las claves que se van a generar. El tamaño de la clave RSA es controlado por la variable **KEY_SIZE** en el archivo **vars**. Por defecto, su valor es de 1024, aunque se especificará un tamaño de 2048 para obtener una mayor seguridad. Después de realizar esta acción, teclear lo siguiente para limpiar todos los archivos de la carpeta **keys**.

```
source ./vars
./clean-all
```

Llegados a este punto, se obtiene la configuración necesaria para comenzar a generar claves y certificados de autenticación de usuarios. Lo primero es generar los parámetros Diffie Hellman para la constitución de claves. Para ello, como usuario **root** ejecutar:

```
./build-dh
```

Una vez finalizado el proceso, se crea el certificado para la CA, el certificado y la clave para el servidor y el certificado y la clave para el cliente. Estas labores se realizan mediante la ejecución de los siguientes comandos respectivamente:

```
./pkitool --initca
./pkitool --server servidor
./pkitool cliente1
```

Para la generación de más clientes, han de ejecutarse las siguientes dos líneas tantas veces como clientes quieran crearse, donde X es el número de cliente a crear.

```
source ./vars
./pkitool clienteX
```

Si se accede a la ruta **/root/easy-rsa/keys**, se pueden ver todas estas claves generadas. El método de autenticación de servidor y clientes es TLS, por lo que debe generarse de igual manera una clave TLS en base a los parámetros Diffie Hellman negociados. Para tal fin, se ejecutan los siguientes comandos:

```
cd /root/easy-rsa/keys
openvpn --genkey --secret ta.key
```

Por último, copiar las claves almacenadas en el directorio de trabajo y pegarlas en la ruta de instalación del programa OpenVPN

```
cp -ai /root/easy-rsa/keys /etc/openvpn
```

6.2.4.3 Server.conf - Configuración del servidor VPN

Una vez generadas las claves necesarias y copiadas a la ruta de instalación del programa, solo resta la configuración de cada uno de los clientes y servidores que conformarán la red. La configuración del servidor es sencilla ya que solo se debe obtener la plantilla de configuración y editarla con las claves y parámetros especificados en pasos anteriores. Para obtener la plantilla ejecutar:

```
cp -ai /usr/share/doc/openvpn*/sample/sample-config-files/roadwarrior-server.conf
/etc/openvpn/server.conf
```

A continuación, situarse en **/etc/openvpn** y mediante el comando **nano** editar el archivo obtenido.

```
cd /etc/openvpn
nano server.conf
```

La edición del fichero se centrará en los siguientes puntos:

```
dev tap
proto udp
rutas ca.crt server.crt server.key (etc/openvpn/keys)
Establecer dh dh2048.pem
tls-auth /etc/openvpn/keys/ta.key 0
cipher AES-256-CBC
comp-lzo adaptive
mode server
server 10.8.0.0 255.255.255.0
ifconfig 10.8.0.1 10.8.0.2
push "route 10.8.0.1 255.255.255.0"
push "route 192.168.0.0 255.255.255.0"
route 10.8.0.0 255.255.255.0
persist-key
```

Para arrancar el servidor hay que situarse en la ruta del archivo **server.conf** y ejecutar

```
openvpn server.conf
```

6.2.4.4 Client.conf - Configuración del cliente VPN

Para configurar el cliente, copiar los archivos **clienteX.key**, **clienteX.crt** y **ca.crt** desde el servidor **(/etc/openvpn**) hasta **/etc/openvpn/clienteX**.

```
mkdir clienteX
cp keys/ca.crt etc/openvpn/clienteX
cp keys/clienteX.key etc/openvpn/clienteX
cp keys/clienteX.crt etc/openvpn/clienteX
```

Tras esto, descargar la plantilla de configuración y editarla:

```
cp -ai /usr/share/doc/openvpn*/sample/sample-config-files/client.conf /etc/openvpn/cliente.conf
nano cliente.conf
```

La edición del fichero se centrará en los siguientes puntos:

```
dev tap
proto udp
remote 212.128.44.177 1194
persist-key
rutas ca.crt clientX.crt clientX.key (en directorio clienteX)
tls-auth /etc/easy-rsa/cliente1/ta.key 1
cipher AES-256-CBC
comp-lzo
```

Para ejecutar el cliente (en máquinas Linux) situarse en la ruta **/etc/openvpn** y ejecutar

```
openvpn cliente.conf
```

6.2.5 Configuración de equipos

Estas acciones son necesarias para configurar cada cliente del simulador del Prototipo de Pruebas. Por este motivo, los equipos a configurar están ubicados en el laboratorio LSI1. Así, es necesario instalar un cliente VPN en cada equipo de la red y crear un directorio de trabajo en Eclipse que contenga la aplicación desarrollada. El cliente VPN permitirá la comunicación con el servidor VPN desde el que se inundará la red con los parámetros de la simulación.

Para realizar la instalación, es necesario disponer de los siguientes elementos generados en la instalación del servidor VPN:

- Claves y certificados de los clientes.
- Archivos **ca.crt** y **ta.key** del propio servidor.
- Fichero plantilla de configuración del cliente VPN.

Además, se necesita el proyecto en Java para crear el directorio de trabajo en el entorno de desarrollo Eclipse. La descarga del *software* restante se detalla a medida que se describe la instalación/configuración.

La instalación del servidor VPN se ha realizado mediante el *software* OpenVPN tal y como se describe en el apartado 6.2.4. Por este motivo, la instalación de los clientes debe realizarse con el mismo *software*. En Windows, su descarga está disponible en el siguiente enlace https://openvpn.net/index.php/open-source/downloads.html. La versión del sistema operativo de los equipos del LSI1 es Windows XP 32 bits, por lo que se descargará el instalador asociado a dichas características. La instalación del *software* es sencilla e intuitiva pero hay que tener en cuenta realizar la ejecución con permisos de administrador.

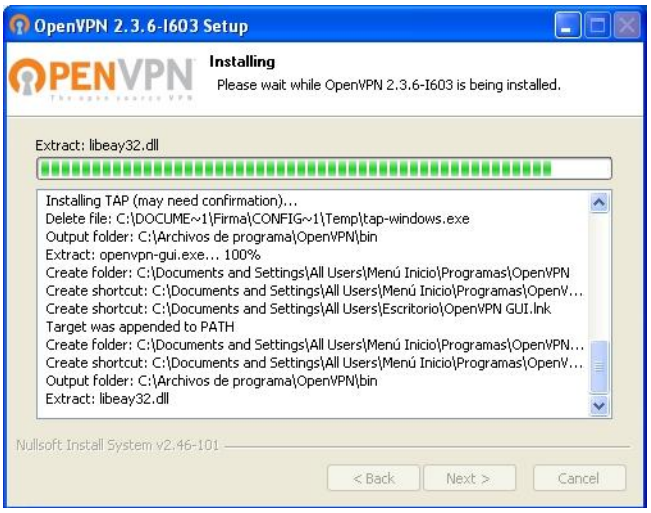


Figura 6.23. Instalación OpenVPN.

Los 2 pasos siguientes son necesarios solamente si se utiliza Windows XP como Sistema Operativo, como es el caso que nos ocupa. En la instalación de la red VPN realizada en el apartado 6.2.4, el tipo de red configurada ha sido TAP. Este sistema necesita de una interfaz virtual que permita realizar la comunicación entre las partes de manera satisfactoria. Por ello, es necesario realizar la instalación del *software* que se puede descargar en la sección *Tap-windows* del mismo enlace especificado anteriormente.

La instalación de este *software* se realiza de forma similar, sin embargo en la pestaña de elección de componentes hay que marcar todos ellos, es decir, **`TAP Virtual Ethernet Adapter`**, **`TAP Utilities`** y **`TAP SDK`**.

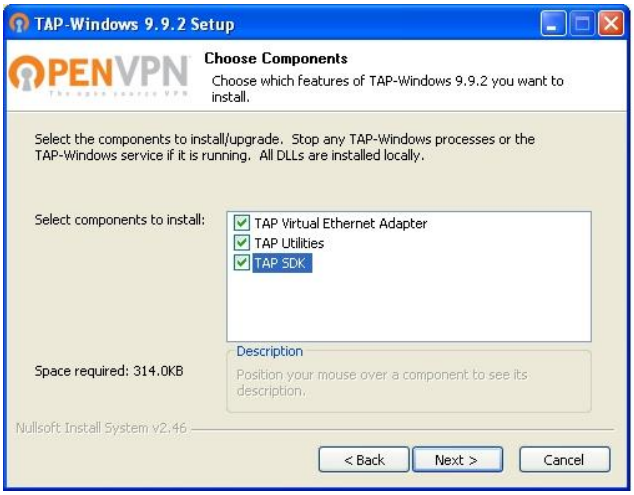


Figura 6.24. Instalación de TAP para Windows.

El siguiente punto importante para el correcto funcionamiento de la red VPN es la desactivación del Firewall de Windows, pues por defecto bloquea este tipo de conexiones tanto entrantes como salientes. Para ello, se debe de seguir la siguiente secuencia de operaciones:

Inicio → Panel de Control → Centro de seguridad → Firewall de Windows → Desactivado, y pulsar sobre el botón **Aceptar**.

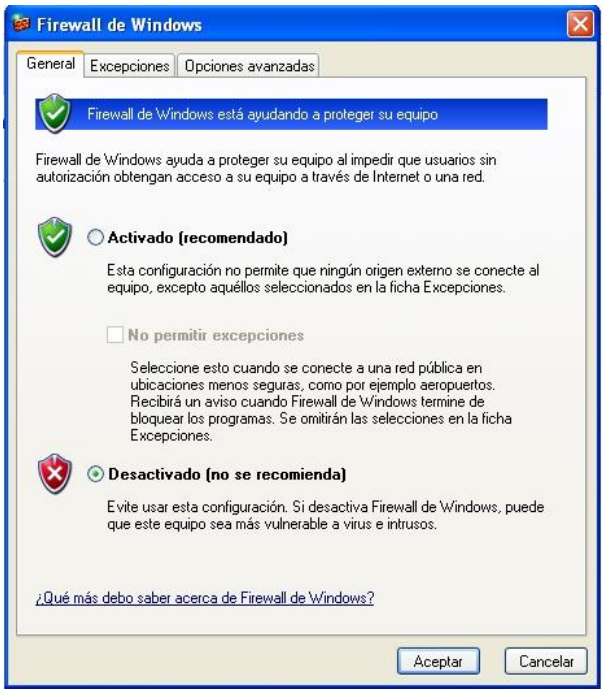


Figura 6.25. Desactivación Firewall de Windows.

En este punto, resta incluir la configuración de la red VPN. Para ello, se debe acceder al directorio donde se ha instalado el *software* OpenVPN. En la carpeta **`config`** introducir los archivos **`cliente.ovpn`**, **`ca.crt`**, **`ta.key`**, **`clienteX.key`** y **`clienteX.crt`**, donde X denota el número de cliente que se instala, siendo diferente en cada instalación.

Una vez que estos dos últimos ficheros han sido pegados en el directorio, se renombran a **`cliente.key`** y **`cliente.crt`** para que haya concordancia con el archivo **`cliente.ovpn`** y su configuración.

Tras ello, la carpeta config quedará de la siguiente manera:

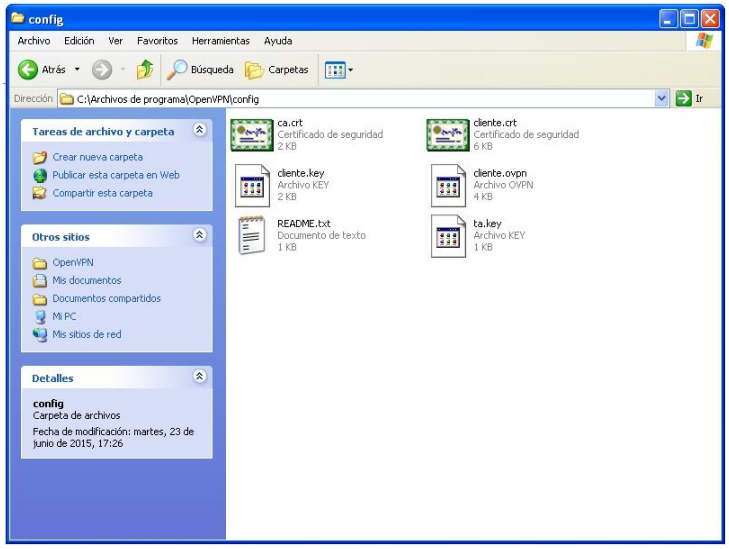


Figura 6.26. Carpeta config del cliente VPN.

En este punto la instalación y configuración del cliente OpenVPN ha finalizado, por lo que se puede proceder a ejecutar el programa. Para ello, hacer *click* en *Connect* y ver que existe conexión con el servidor VPN (IP 10.8.0.1) mediante el uso del comando **`ping`**.

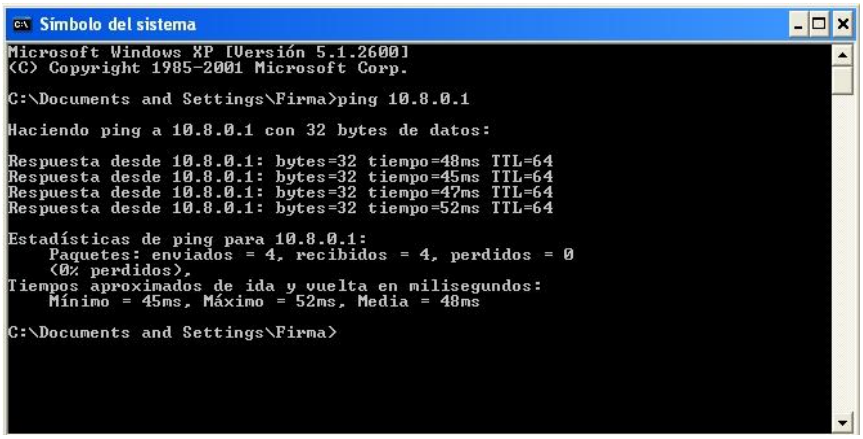


Figura 6.27. Conexión con servidor VPN mediante ping.

Una vez verificada la conexión con el servidor VPN, hay que configurar el entorno de desarrollo Eclipse con el proyecto que contiene la aplicación desarrollada en el Prototipo de Pruebas. Para importar el proyecto en Eclipse se selecciona un directorio de trabajo nuevo y se hace *click* en *File* → *Import*. Tras esta acción, aparece una ventana con diferentes pestañas. Desplegar la primera de ellas (General) y elegir la opción *Existing Projects into Workspace*. A continuación, *clickar* en *Next*. En la siguiente ventana, pulsar sobre *Browse* del punto *Select root directory* y dirigirse al directorio donde se encuentra el proyecto a importar, en este caso **`TestBenchServer`**. Finalmente, pulsar en **Aceptar**.

Mediante el procedimiento descrito, el proyecto **`TestBenchServer`** es importado al nuevo directorio de trabajo. Este proyecto ha sido implementado para los servidores de la red del Prototipo de Pruebas tal y como se describe en el apartado 6.3.1.1. Un aspecto a tener en cuenta es que Eclipse puede lanzar errores en algunas clases del proyecto debido a que no encuentra la librería org.json.jar. En este caso, hay que importar la librería modificando el *Build Path* del proyecto (*Click* derecho sobre el nombre del proyecto → *Build Path* → *Configure Build Path -> Libraries -> Add External JARs*).

Por último, en el cliente de la red (equipo situado en el DSIE), se importa el proyecto **`TestBenchClient`**. A modo de resumen, el software instalado y configurado en los diferentes equipos del Prototipo de Pruebas es el siguiente:

- Equipo situado en el DSIE: **`Servidor VPN`** + **`TestBenchClient`**.
- Equipos situados en el LSI1: **`Cliente VPN`** + **`TestBenchServer`**.

6.3 Implementación

6.3.1 Test Bench

6.3.1.1 Estructura de la Aplicación

A continuación, se muestra el diagrama de relaciones de uso de los diferentes paquetes que componen los proyectos desarrollados en Eclipse.

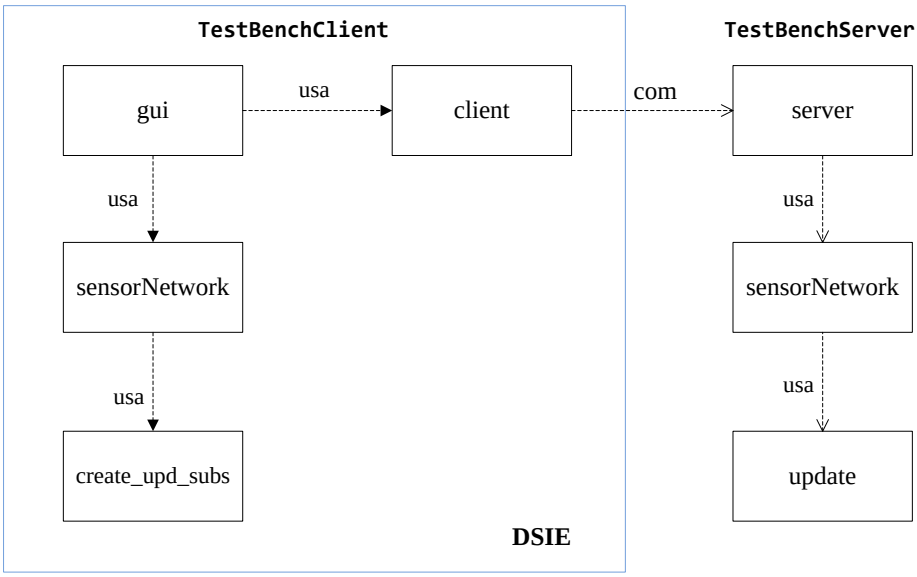


Figura 6.28. Diagrama de Relaciones de Uso.

Tal y como se puede observar en la Figura 6.288, se han desarrollado dos proyectos denominados **`TestBenchClient`** (Prototipos Inicial y de Pruebas) y **`TestBenchServer`** (Prototipo de Pruebas). En el nivel más bajo del proyecto **`TestBenchClient`**, se encuentra el paquete **`create_upd_subs`** que contiene las clases necesarias para construir operaciones de los tipos **`append`**, **`subscribe`** y **`update`** en formato **`Json`**. En el segundo nivel se halla el paquete **`sensorNetwork`** que contiene las clases **`SensorCB`**, **`SensorSB`** y **`SensorV`**, las cuales permiten enviar al **`Context Broker`** las operaciones de creación, suscripción y actualización respectivamente.

En el nivel más alto se encuentra el paquete **`gui`** que, tal y como su nombre indica, contiene la interfaz gráfica del simulador modelada por la clase **`Test`**. Esta interfaz permite realizar una simulación en modo local al utilizar las clases del paquete **`sensorNetwork`** (simulación del Prototipo Inicial) o realizar una simulación en modo remoto al utilizar las clases del paquete **`client`** (simulación del Prototipo de Pruebas)

Por otro lado, el paquete **`server`** del proyecto **`TestBenchServer`** permite adquirir los parámetros de la simulación lanzada a través de la interfaz gráfica del proyecto **`TestBenchClient`**. En el segundo nivel, el paquete **`sensorNetwork`** contiene la clase **`SensorV`** que implementa el envío al **`ContextBroker`** de las operaciones de actualización generadas por el paquete **`update`** que contiene las clases necesarias para construir operaciones de este tipo.

6.3.1.2 Operaciones create, update y subscription

Las operaciones asociadas a la creación, actualización y suscripción de entidades del *Context Broker* han sido implementadas por las clases del paquete **`create_upd_subs`** del proyecto **`TestBenchClient`** y del paquete **`update`** del proyecto **`TestBenchServer`**. Así, las operaciones de creación y actualización (**`append`**/**`update`**) siguen la siguiente estructura:

```
{"contextElements":
 [
   {
    "type":"Room",
    "isPattern":"false",
    "id":"Room1",
    "attributes":
      [
        {"name":"temperature",
         "type":"centigrade",
         "value":"0/1/0/2015-05-21X17:45:15.451"
        }
      ]
    }
 ],
 "updateAction":"APPEND"
}
```

La operación anterior permite crear una entidad denominada Room1 en la base de datos del *Context Broker*. Además, esta entidad tiene un atributo de nombre temperatura, tipo centígrado y valor determinado por un *String*. Si se quisiera realizar la operación de actualización, solo sería necesario cambiar el valor de **`updateAction`** por **`UPDATE`**.

Por otro lado, la estructura de las operaciones de suscripción (**`subscribe`**) es la siguiente:

```
{"entities":
 [
   {
    "type":"Room",
    "isPattern":"false",
    "id":"Room1"
   }
 ],
 "attributes":
 [
   "temperature"
 ],
 "reference":"http://130.206.85.99:5050/notify",
 "duration":"P1M",
 "notifyConditions":
 [
  {
    "type":"ONCHANGE",
    "condValues":
      [
       "temperature"
      ]
  }
 ]
}
```

La suscripción anterior se realiza sobre el atributo de temperatura de la entidad Room1. Además, se trata de una suscripción tipo **`ONCHANGE`** sobre el valor de temperatura. La duración de la suscripción es de un mes (**`P1M`**). Así, cuando un evento cumpla estos requisitos, se debe enviar una notificación a la dirección **`http://130.206.85.99:5050/notify`**.

Finalmente, mencionar que la implementación de estas operaciones ha sido realizada en formato **`Json`** mediante la librería org.json.jar.

6.3.1.3 Aspectos relativos a la conexión con el Broker

El envío de las operaciones mencionadas anteriormente al *Context Broker* se realiza mediante la clase **`HttpURLConnection`** de Java. El siguiente fragmento de código ilustra este hecho:

```
//Conexion con Orion Context Broker
String myURL="http://130.206.85.99:1026/ngsi10/updateContext";

try{

	url = new URL(myURL); //FILAB
	conn = (HttpURLConnection) url.openConnection();
	//Just want to do an HTTP POST here
	conn.setRequestMethod("POST");
	conn.setRequestProperty("Content-Type", "application/json");
	conn.setRequestProperty("Accept", "application/json");
	conn.setRequestProperty("Content-Length", "200");

	//To write output to this url
	conn.setDoOutput(true);
	conn.setConnectTimeout(100);
	conn.setReadTimeout(100);

	conn.connect();

	connected = true;

	os = conn.getOutputStream();

	System.out.println("connect ---> Conexion con FIWARE establecida");

}catch(Exception e){
	System.out.println("connect. Excepcion iniciar conexion (FAIL1): \t"+e.toString());
	e.printStackTrace();
	contfailIni++;
}
```

Es importante mencionar que, cada instancia de la clase **`HttpURLConnection`** es utilizada para realizar solo una conexión. Por este motivo, hay que establecer conexión con el *Broker* cada vez que se envía una solicitud. Aunque la característica mencionada es intrínseca a Java, resta eficiencia a la implementación del **`TestBench`**.

Por otro lado, se fija un tiempo de 100 ms para los *Timeouts* de conexión y de lectura de la respuesta del *Context Broker* para evitar que el *software* de simulación de la red de sensores se interrumpa y no cumpla con el periodo de envío de solicitudes establecido.

6.3.1.4 Clase Server.java

La clase **`Server.java`** es utilizada por el Prototipo de Pruebas y se encarga de permanecer a la espera de la recepción de parámetros para poder lanzar una nueva simulación. Para ello, utiliza un **`DatagramSocket`** que atiende peticiones del puerto 8080 y recibe un único *String* con los parámetros separados por comas. Los datos que se esperan recibir son, en orden:

- Número de sensores, representado por la variable **`nSensores`**.
- Periodo, representado por la variable **`periodo`**.
- Número de datos por sensor, representados por la variable **`it`**.
- Fecha de la máquina cliente (equipo del DSIE), representada por la variable **`fecha`**.
- Tamaño de *payload*, representado por la variable **`payload`**.

La variable **fecha** se utiliza para calcular el *offset* entre la máquina remota y la máquina local con el fin de sincronizar los relojes de ambas máquinas y obtener resultados coherentes. La separación de cada una de las variables dentro del *String* recibido se realiza por medio de la clase **StringTokenizer** que recibe como argumento el carácter de separación para poder iterar y obtener la totalidad de las variables.

Además, existe otra variable importante en esta clase. Se trata de la variable **idServer** que permite identificar al servidor que atiende y procesa la petición. Esta variable se utiliza para calcular los identificadores de los registros asociados a las entidades cuya información de contexto será actualizada por cada uno de los servidores de la red VPN, evitando sobrescribir datos, lo que provocaría pérdidas en la simulación.

A continuación se muestra la relación de variables que utiliza esta clase:

```
  long offset = 0;       //diferencia entre fechaDSIE y fecha local
  final int idServer=0;   //identificador del equipo
  String str;             //string de recepcion
  int nSensores = 0;      //numero de sensores virtuales
  int periodo = 0;       //periodo entre envio de datos
  int it=0;               //numero de datos a enviar
  long fecha;             //fecha del cliente (DSIE)
  int payload=0;          //payload
```

Las siguientes líneas de código muestran la creación del **DatagramSocket** y el tratamiento de la información entrante:

```
DatagramSocket serverSocket = new DatagramSocket(8080);

/* WHILE TRUE */

DatagramPacket receivePacket = new DatagramPacket(receiveData, receiveData.length);
serverSocket.receive(receivePacket);
str = new String( receivePacket.getData(), 0, receivePacket.getLength());

//Todo elemento separado por una coma lo aparta
StringTokenizer st=new StringTokenizer(str,",");

//iteracion de los elementos recibidos
String [] aux=new String [5];
      int j=0;
      while (st.hasMoreElements()){
            aux[j]=(String) st.nextElement();
            j++;
      }

//copia de variable auxiliar a su correspondiente
nSensores=Integer.parseInt(aux[0]);
periodo=Integer.parseInt(aux[1]);
it=Integer.parseInt(aux[2]);
fecha=Long.parseLong(aux[3]);
payload=Integer.parseInt(aux[4]);

//calculo de offset
Date date = new Date();
long time = date.getTime();
offset = fecha-time;
```

Una vez recibidos los parámetros de simulación, sólo resta conocer qué registros actualizará cada servidor de la red. Esta tarea es controlada por dos variables. La variable **ini** indica el primer registro del *Context Broker* a actualizar mientras que la variable **fin** indica el último de estos registros (en realidad el servidor llegará a actualizar hasta el registro **fin**-1).

El algoritmo de asignación utilizado es el siguiente:

```
int ini=idServer*nSensores;
int fin=nSensores*(idServer+1);
```

Así, si se supone que la variable **nSensores** es igual a 2 (cada servidor tiene que actualizar dos registros del *Context Broker*) y que hay dos servidores escuchando la petición con identificadores **idServer**=0 e **idServer**=1 respectivamente. Los valores de las variables **ini** y **fin** del primer servidor (**idServer**=0) serán **ini**=0 y **fin**=2, mientras que los del segundo servidor (**idServer**=1) serán **ini**=2 y **fin**=4. Así, el primer servidor tiene asignados los registros 0 y 1 del *Context Broker* mientras que el segundo tiene los registros 2 y 3 del mismo. De esta forma, gracias al identificador cada servidor actualizará registros diferentes.

Por último, la clase **Server** invoca a la clase que lanza la simulación. A continuación, se muestra la inicialización de los sensores (número de hilos atendiendo a la variable **nSensores**) y la llamada al constructor de la clase **SensorV.java**:

```
SensorV sensors[] = new SensorV[nSensores];

      for(int i = 0;ini<fin;ini++) {
          sensors[i] = new SensorV(periodo,ini,it,offset,payload);
          sensors[i].start();
          i++;
      }
```

Como se puede observar en la creación del **DatagramSocket**, toda esta información va incluída en un bucle **while(true)**, con objeto de reutilizar la conexión sin necesidad de arrancar el programa cada vez que se quiera ejecutar una simulación. En definitiva, una vez que se arranca la clase esta está continuamente escuchando el puerto 8080 y lista para atender peticiones.

6.3.1.5 Clase Client.java

La clase **Client.java** es utilizada por el Prototipo de Pruebas y se encargada de recibir los parámetros de la GUI de simulación y enviarlos a la red VPN mediante un **DatagramSocket**. Para ello, la clase **Client.java** dispone de un constructor que es llamado en la clase **Sim.java** (interfaz gráfica de la simulación) y que incluye todos los parámetros necesarios para arrancar la prueba. Una vez registrados estos parámetros en variables locales de la clase, se llama al método **start()** que crea un **DatagramSocket**, incluye la información recibida en un *String* y la envía a la dirección broadcast de la red VPN (10.8.0.255). Por último se cierra la conexión del *socket*.

Tal y como se ha explicado en el punto anterior, el orden de las variables tiene que ser el mismo en el envío y recepción, además de ir separadas por comas.

El código del método **start()** se muestra a continuación:

```
public void start() throws IOException {
    DatagramSocket clientSocket = new DatagramSocket();
    InetAddress IPAddress = InetAddress.getByName("10.8.0.255");
    byte[] sendData = new byte[100];

    try {

      Date date = new Date();
      long fecha = date.getTime();
      String param = Integer.toString(nSensores)+
                  ","+Integer.toString(periodo)+
                  ","+Integer.toString(it)+
                  ","+Long.toString(fecha)+
                  ","+Integer.toString(payload);
      sendData = param.getBytes();

      DatagramPacket sendPacket =
            new DatagramPacket(sendData,
                               sendData.length,
                               IPAddress,
                               8080);

      clientSocket.send(sendPacket);

    }

    finally {
      System.out.println("closing...");
      clientSocket.close();

    }
```

6.3.2 Cygnus

6.3.2.1 Recepción de notificaciones

El inyector Cygnus es utilizado por ambos prototipos. Con el objetivo de almacenar la información de contexto recibida del Context Broker, se procede a la modificación de ciertas partes del código de Cygnus. Los cambios se han realizado en las siguientes clases:

- **OrionHDFSSink.java** (con ruta **es.tid.fiware.fiwareconnectors.cygnus.sinks**).
- **OrionRestHandler.java** (con ruta **es.tid.fiware.fiwareconnectors.cygnus.handlers**).

Específicamente, los cambios han sido los siguientes:

- Clase **OrionHDFSSink.java**
  - **Añadir SinkTime**: esta traza permite conocer el instante de tiempo en el que el sumidero accede al dato del canal y lo transfiere a Cosmos. Es útil para conocer el retardo extremo a extremo, considerándose como inicio la generación en Eclipse y como final la extracción del dato por parte del sumidero.
  - **Persistencia de datos local**: aprovechando la variable **attr_persistence** se modifica el código del **método persist()** para que permita seleccionar el destino al que el sumidero enviará los datos. Si el valor de **attr_persistence** es igual a **local**, se utilizará un **StringBuffer** que irá acumulando los datos procesados, siendo almacenados de manera local en un fichero llamado **sim.txt** en la ruta **/root**. En el caso de que la variable **attr_persistence** sea diferente a ese valor, los datos se enviarán a Cosmos.

Las condiciones de generación y escritura del fichero son 5 segundos de simulación o la recepción del valor -4 en alguno de los datos, correspondiéndose con el último valor del sensor de la simulación. Para la generación y escritura de los ficheros se utiliza `FileWriter` (para referenciar al archivo) y `PrintWriter` (para escribir en el archivo).

- **Eliminación de ciertas trazas**: se comentan ciertas trazas de consola que no aportan información útil y provocan cierto retardo en las simulaciones.

- Clase **OrionRestHandler.java**
  - **Persistencia de datos local**: de la misma manera que en el lado del sumidero, se desarrolla un mecanismo en el método `getEvents()` para almacenar de manera local los datos obtenidos en la fuente. Los datos se almacenan de manera local en cada una de las simulaciones en la ruta `/root` bajo el nombre `source.txt`.
    Las condiciones de creación y escritura serán las mismas, es decir, 5 segundos de simulación o la existencia de un -4 como valor del dato.
  - **Eliminación de ciertas trazas**: se eliminan ciertas trazas `logger.info` que resultan innecesarias y solo retrasan la simulación.

6.3.3 Cosmos

6.3.3.1 Comandos

El Generic Enabler Cosmos es utilizado por el Prototipo Inicial. Dentro de Cosmos pueden diferenciarse dos sistemas fundamentales. Por un lado, se tiene un sistema Linux y por otro un sistema Hadoop de tipo HDFS que permite la comunicación con el *Context Broker* y almacena los datos de manera histórica.

Los comandos descritos en este apartado, se corresponden con el sistema Hadoop. Aunque existe una amplia variedad, sólo se muestran los más significativos [41].

Creación y eliminación de directorios

- **hadoop fs -mkdir <ruta>**
  Crea un directorio en la ruta que se le especifique.
- **hadoop fs -rmr <ruta>**
  Elimina el directorio de la ruta que se le especifique.

Creación, copia y descarga de ficheros

- **hadoop fs -put <nombre fichero> <ruta destino HDFS>**
  Crea un fichero en la ruta HDFS especificada. Este comando es análogo a *touch* de Linux.
- **hadoop fs -put <ruta local> <ruta destino HDFS>**
  Copia uno o múltiples ficheros desde un sistema local hasta un sistema Hadoop.
- **hadoop fs -get <ruta HDFS> <ruta destino local>**
  Descargar uno o múltiples ficheros hacia un sistema local.

Información sobre directorios y ficheros

- **hadoop fs -ls <ruta>**
  Proporciona información de los directorios y ficheros alojados en esa ruta.
- **hadoop fs -cat <ruta fichero>**
  Abrir, como si de un editor de texto se tratara, el fichero indicado.
- **hadoop fs -du <ruta>**
  Muestra el tamaño de ficheros y directorios contenidos en la ruta especificada.

6.3.3.2 Código Hive Basic Client

La clase **`HiveBasicClient.java`** permite establecer en la máquina del *Broker* un cliente Hive, brindando la posibilidad de lanzar consultas de manera remota a los datos históricos almacenados en Cosmos. Se puede modificar su código para lanzar de manera automática, las consultas necesarias para el acceso de datos, así como su tratamiento y descarga en un fichero de texto.

Para este propósito, se utiliza la plantilla facilitada por el equipo de FIWARE en su repositorio de github. El código en el que se basa el cliente implementado se aloja en el siguiente enlace. Las modificaciones más significativas que se han realizado residen en los métodos **`doQuery`** y **`main`**. En primer lugar, cabe destacar las librerías importadas en la clase, pues difieren ligeramente de las que se declaran en la plantilla utilizada, concretamente son las siguientes:

```
import java.io.*;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
```

El método **`doQuery`** se encarga de lanzar remotamente a Cosmos las instrucciones que se le pasen como argumento. Inicialmente, este método era de tipo void. Sin embargo, se ha modificado para que devuelva un *String*, que consistirá en los datos devueltos por la consulta que se lanzada a Cosmos.

Además, se añaden dos variables de tipo *String*, una que se usa como auxiliar y que recoge los datos devueltos por cada una de las iteraciones de la consulta y otra del mismo tipo que se usa como el String a devolver por el método. Esta última es fruto de las concatenaciones de la primera variable.

```
private static String doQuery(String query) {
       //final String - final String returned
        String fS = "";
        //String - used for each query result
        String s = "";
        try {
            // from here on, everything is SQL!
            Statement stmt = con.createStatement();
            ResultSet res = stmt.executeQuery(query);
            // iterate and concatenate the result on the final String
            while (res.next()) {
                s = "";
                for (int i = 1; i < res.getMetaData().getColumnCount(); i++) {
                    s += res.getString(i) + ",";
                } // for
                s += res.getString(res.getMetaData().getColumnCount());
                System.out.println(s);
                fS =fS+s+"\n";
            } // while

            // close everything
            res.close();
            stmt.close();
        } catch (SQLException ex) {
        System.out.println(ex);
        //System.exit(0);
        } // try catch
        return fS;
    } // doQuery
```

Como se observa, el método recoge como parámetro un *String* llamado *query*, establece una conexión SQL con Cosmos y lanza la *query* sobre ella, almacenando y concatenando la respuesta obtenida. En cuanto al método **`main`**, se pueden diferenciar tres partes o tareas fundamentales dentro del código:

- Conexión con Cosmos.
- Creación de tablas SQL y lanzamiento de consultas.
- Formateado de la cadena y descarga en fichero.

6.3.3.2.1 Conexión con Cosmos

Los parámetros necesarios para realizar la conexión con Cosmos son:

- *hiveServer*: dirección IP de Cosmos.
- *hivePort*: puerto de Cosmos, por defecto es el 10000.
- *cosmosUser*: usuario de Cosmos.
- *comosPassword*: contraseña configurada en Cosmos.

Gracias al método **`getConnection`**, se puede establecer una conexión remota con Cosmos. Adicionalmente se añaden dos variables que se utilizan para la identificación de la prueba realizada, como son el número de sensores (**`nOfSensors`**) y el periodo entre envío de información (**`sleepTime`**).

6.3.3.2.2 Creación de tablas SQL y lanzamiento de consultas

La creación de tablas SQL por parte del cliente Hive se realiza a través de una consulta. En dicha consulta, se puede delimitar la forma en la que estarán dispuestas las columnas, señalando cual es el carácter de separación que debe de haber entre ellas para poder discernirlas. Además de ello, se especifica la ruta donde se adquieren los datos y se almacena la tabla, siendo posible su creación de manera remota y no necesariamente estando ubicados en el mismo directorio.

El objetivo es el de automatizar esta tarea, por lo que se utiliza un bucle *for* que recorre cada uno de los directorios en los que se alojan los datos de la simulación, creando una tabla SQL para cada uno y lanzando su consulta asociada.

Al igual que en el método **`doQuery`**, se usan dos variables de tipo *String* con objeto de poder almacenar la respuesta e ir concatenando el histórico de las mismas. La variable auxiliar es la llamada **`preResult`**, mientras que el *String* final donde se irán concatenando las respuestas a las consultas se denomina **`result`**.

```
int n = Integer.parseInt(nOfSensors);
        System.out.println("-----\nCreando tablas, obteniendo datos\n-----");
        for(int i=1; i<=n;i++){
                //Deleting old tables
                doQuery("drop table rmartinezcarreras_sensorID"+i);

                //Creating tables
                String q = "create external table rmartinezcarreras_sensorID"+i+" (recvTs string,
recvT string, entityId string, entityType string, attrName string, attrType string, attrValue string,
attrMd1 string, attrMd2 string, attrMd3 string, sinkTime string) row format delimited fields
terminated by ',' location '/user/rmartinezcarreras/def_serv/def_serv_path/room"+i+"_room/'";

                doQuery(q);
                System.out.println("rmartinezcarreras_sensorID"+i);
                //Do Query! We select recvt and attrValue columns
                String     preResult     =     doQuery("select     recvT,attrValue,sinkTime     from
rmartinezcarreras_sensorID"+i);
                result = result + preResult;
        }//for
```

Por último, cabe destacar la utilización de *drop tables* para eliminar de manera previa cualquier tabla existente con el nombre que se pretende crear, con objeto de evitar errores de ejecución del código.

6.3.3.2.3 Formateado de la cadena y descarga en fichero

El objetivo del formateado no es más que el de dotar de cierta coherencia y organización de los datos para su posterior tratamiento en RStudio. Se ha elegido separar las columnas mediante comas, por lo que se tienen que ir eliminando todos los "residuos" que no sean objeto de estudio. El proceso de formateado ha sido el siguiente:

1. Eliminación de comillas dobles.
2. Eliminación del nombre de las columnas devueltas por la consulta.
3. Sustitución de / por comas, este símbolo aparecía en los datos enviados por Eclipse.
4. Eliminación de la inicial de los días de la semana albergados en el formato fecha-hora que devuelve Cosmos.

La porción de código que lleva a cabo esta labor es la siguiente:

```
//From the result, delete the innecesary strings
result = result.replace("\"" , "");
result = result.replace("recvTime:" , "");
result = result.replace(",attrValue:" , ",");
result = result.replace("sinkTime:" , "");
result = result.replace("/" , ",");
result = result.replace("X" , ",");
result = result.replace("M" , ",");
result = result.replace("T" , ",");
result = result.replace("W" , ",");
result = result.replace("F" , ",");
result = result.replace("S" , ",");
result = result.replace("}" , "");
```

Una vez formateada la salida, queda almacenar la información en un fichero. Para ello se utilizan **`FileWriter`** y **`PrintWriter`** de **`java.io`**. La ruta donde se almacena el fichero es:
`/usr/fiware-connectors/resources/hive-basic-client/Fiware_nOfSensors_sleepTime.txt`

El siguiente fragmento de código realiza la tarea descrita:

```
        //Save using FileWriter
            FileWriter fichero = null;
            PrintWriter pw = null;
            try
            {   File     archivo     =     new     File("/usr/fiware-connectors/resources/hive-basic-
client/Fiware_"+nOfSensors+"_"+sleepTime+".txt");
                fichero = new FileWriter(archivo);
                pw = new PrintWriter(fichero);

                pw.print(result);
                fichero.close();

            } catch (Exception e) {
                e.printStackTrace();
            }
```

6.3.3.2.4 Compilación

La clase se aloja en la máquina del *Context Broker*. Para ello, se abre sesión mediante WinSCP y se transfiere a la ruta siguiente:

`/usr/fiware-connectors/resources/hive-basic-client/src/main/java/es/tid/fiware/cosmos/hivebasicclient`

Una vez ahí, para compilarla se ejecutan los siguientes comandos:

```
cd /usr/fiware-connectors/resources/hive-basic-client
mvn package
```

Al finalizar el proceso la clase ya estará compilada y lista para utilizarse. Esta acción solo debe hacerse cuando haya cambios en el código de la clase.

Cuando se precise la necesidad de utilizar los servicios que proporciona la clase, hay que situarse en la ruta **`/usr/fiware-connectors/resources/hive-basic-client`** y ejecutar la siguiente línea de código modificando los argumentos de entrada según el tipo de simulación que se haya realizado.

```
mvn exec:java -Dexec.args="<nSensores> <periodo>"
```

Tras su ejecución, se obtienen los datos almacenados en Cosmos en un fichero de texto formateado y preparado para ser cargado en RStudio y poder realizar su análisis estadístico.

6.3.3.2.5 Delete.sh. Automatización del borrado de datos en Cosmos

Cosmos almacena de manera histórica los datos de las simulaciones, por lo que escribe en sus ficheros toda la información recibida del *Context Broker* de manera que en las simulaciones posteriores, los datos son agregados al histórico.

No obstante, el propósito que se persigue es la evaluación de manera individual de los resultados obtenidos entre simulaciones, siendo innecesarios los datos acumulados de simulaciones anteriores. Por ello, simulación tras simulación se deben ir borrando los ficheros generados en la prueba, pudiendo llegar a ser una tarea tediosa cuando se realizan simulaciones con un alto número de sensores (cada sensor tiene una carpeta asociada que contiene un fichero asociado).

Es importante desarrollar un *script* que elimine, de manera automática, los directorios asociados al finalizar cada una de las simulaciones pertinentes. La ruta en la que se alojan es **`user/rmartinezcarreras/def_serv/def_serv_path/roomX_room`**, donde X se corresponde con el número de sensor.

El *script* itera desde 1 hasta N, donde N será el número de sensores especificados por el usuario y que se toma como argumento de entrada. Para cada una de estas iteraciones se procede al borrado del directorio de ese sensor determinado.

Para crear y modificar el *script* se ejecutarán los siguientes comandos:

```
touch delete.sh
nano delete.sh
```

Así pues, la sintaxis en Cosmos para la ejecución del *script* es:

```
bash delete.sh <número de sensores>
```

El código del script que realiza lo explicado se muestra a continuación.

```
NUM=1
a="hadoop fs -rmr /user/rmartinezcarreras/def_serv/def_serv_path/room"
b="_room"

while [ $NUM -le $1 ]; do
       c=$a$NUM$b
       $c
       let NUM=$NUM+1
done
```

6.4 *Interfaz del Simulador*

6.4.1 Clase Sim.java

La clase **`Sim.java`** implementa la interfaz gráfica del simulador de la red de sensores.

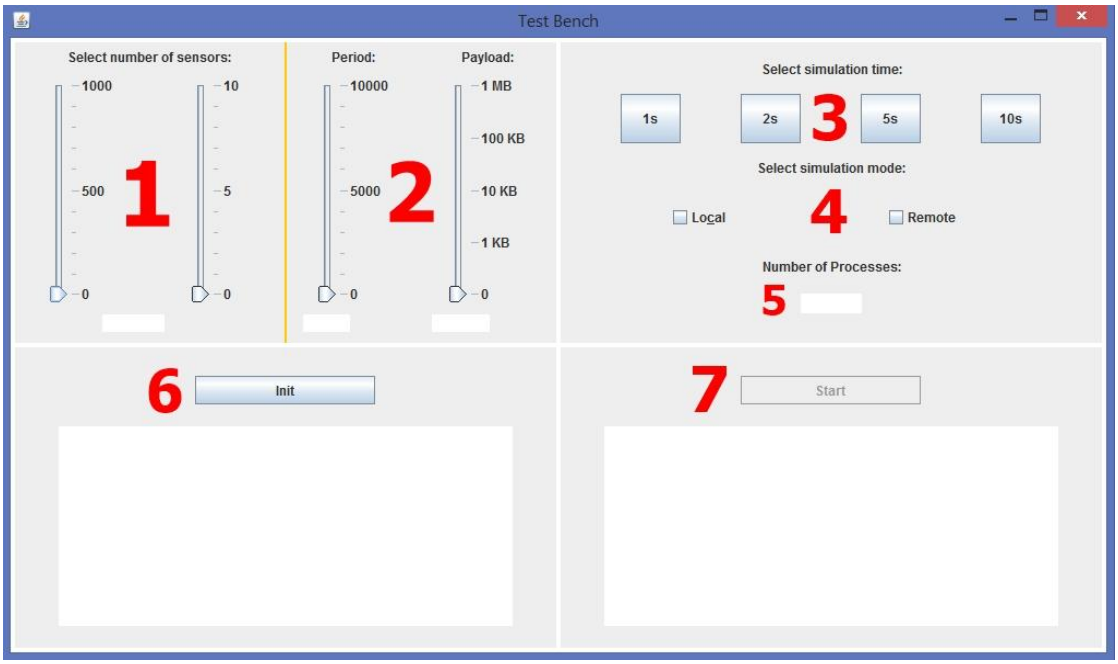


Figura 6.29. Interfaz gráfica del simulador.

La Figura 6.29 muestra la GUI del simulador. Tal y como se puede observar, la interfaz gráfica consta de 7 controles que permiten configurar la simulación de la red de sensores de los prototipos de laboratorio estudiados en el apartado 6.1. A continuación, se describe la funcionalidad de los controles implementados:

1. **Menú de Selección del número de sensores**: está formado por dos barras verticales de diferente resolución que tienen como fin establecer el número de sensores de la simulación. La barra de la izquierda permite una selección a mayor escala (0-1000 sensores) mientras que la barra de la derecha es más precisa (0-10 sensores). Así, la configuración final del número de sensores viene determinada por la selección realizada en ambas barras. Por ejemplo, si se selecciona 200 sensores en la barra de la izquierda y 5 sensores en la barra de la derecha, la configuración final es de 205 sensores.
2. **Menú de Selección de periodo y payload**: en este caso consta de dos barras verticales aunque para fines diferentes. La barra de la izquierda permite seleccionar el periodo de envío de datos de la simulación (en ms), mientras que la barra de la derecha permite seleccionar el *payload* que se incorpora en cada trama enviada al *Context Broker*.
3. **Selección del tiempo de simulación**: la configuración del tiempo de simulación se realiza mediante cuatro botones: 1s, 2s, 5s y 10s. El tiempo de simulación y el periodo determinan el número de solicitudes enviadas por cada sensor virtual al *Context Broker* durante la simulación.
4. **Modo de simulación**: el modo **Local** se corresponde a la ejecución del simulador asociado al Prototipo Inical (almacenamiento histórico en Cosmos), mientras que el modo **Remote** permite la ejecución del simulador correspondiente al Prototipo de Pruebas (almacenamiento local en equipo DSIE).
5. **Número de procesos**: este parámetro permite configurar el número de servidores de la red VPN. Para simulaciones del Prototipo Inicial este valor siempre es 1, siendo variable para simulaciones del Prototipo de Pruebas.

6. **Botón de inicialización**: permite crear tantas entidades y suscripciones como se hayan configurado mediante los controles 1 y 5. Así, el número total de entidades es el resultado de multiplicar el número de sensores por el número de procesos.
7. **Botón Start**: previo arranque de Cygnus, tras pulsar este botón se inicia la simulación con la configuración realizada mediante los controles 1-6.

El *TextField* que hay situado en la parte inferior del menú asociado al control 6 permite mostrar información relativa a la creación/suscripción de entidades en el *Context Broker* así como para recordar la inicialización del inyector Cygnus para el correcto funcionamiento de la prueba. Por otro lado, el *TextField* situado en la parte inferior del menú asociado al control 7 tiene la función de ofrecer un resumen de la configuración establecida en la simulación (número de paquetes enviados por sensor, número de paquetes totales en la simulación, periodo de la simulación, etc). La Figura 6.30 muestra un ejemplo de la configuración de parámetros de una simulación remota de 300 sensores por proceso, 500ms de periodo, 1KB de *payload*, 10 segundos de duración con 2 procesos.

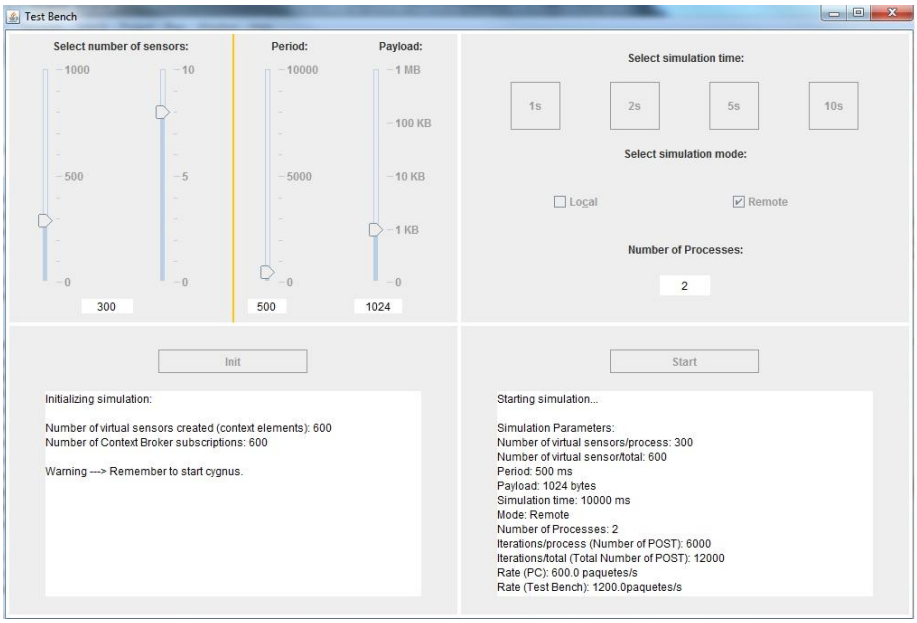


Figura 6.30. Ejemplo de configuración de parámetros de simulación.

Finalmente, Es importante mencionar que la clase **`SensorV.java`** imprime por consola trazas importantes en el transcurso de la simulación. Un ejemplo de esta característica es:

```
RUN ---> Iteracion:       48 Tiempo consumido:      110
updateRegister ---> ... calling updateDB
updateDB ----> El tamaño de la trama a enviar es de: 254 bytes
updateDB ----> try to write to os ... 254
updateDB ----> conn method is ... POST
updateDB ----> Inicio de la transmision
updateDB ----> Fin de la transmision
updateDB ----> Transmision OK!!!!!!!  ---> 200
RUN ---> Iteracion:       49 Tiempo consumido:      93
DATOS DE LA SIMULACION ---> Entro en el bucle a las 2015-06-29X11:44:33.208 y
termino a las 2015-06-29X11:44:43.177
Tiempo total de la simulacion: 9969
Numero de iteraciones:    50
Numero de paquetes enviados:    50
Numero de fallos en la Tx:      0
Numero de excepciones en metodo connect al iniciar la conexion:  0
Numero de excepciones al tx paquetes en updateDB:   0
Numero de excepciones al desconectar:  0
```

6.5 Procedimientos de Simulación

En el apartado 6.1 se describe el diseño de dos prototipos de laboratorio diferentes. Por lo tanto, para lanzar simulaciones es necesario realizar una serie de acciones en función del prototipo desplegado en el laboratorio.

Así, el Prototipo Inicial será empleado por aquellos escenarios cuya persistencia de datos se realice en Cosmos, mientras que el Prototipo de Pruebas será utilizado por aquellos escenarios cuya persistencia de datos se realice de manera local. En los siguientes apartados se explica paso a paso el procedimiento de simulación para cada prototipo de laboratorio.

6.5.1 Prototipo Inicial

En primer lugar, se necesita una ventana terminal en el lado del *Broker*, otra ventana terminal para la ejecución de Cygnus en la máquina del DSIE y una tercera ventana terminal para Cosmos (130.206.80.46 en PuTTY utilizando las credenciales de la cuenta en FILAB).

Los pasos para realizar una simulación son los siguientes:

1. Eliminar el contenido de la base de datos del *Broker* a través de su terminal. Para ello, ejecutar los siguientes comandos:

```
mongo orion
> db.dropDatabase()
{ "dropped" : "orion", "ok" : 1 }
> exit
```

2. Arrancar el *Broker* mediante la ejecución del siguiente comando:

```
contextBroker -fg
```

   Tras ello, el *Context Broker* se inicializará y quedará escuchando peticiones de creación/suscripción de entidades así como la recepción de nuevos datos. Todos los eventos que se produzcan serán imprimidos en pantalla.

3. Asegurarse de que el fichero **`agent_test.conf`** de Cygnus está configurado para el almacenamiento remoto. Para ello acceder al directorio donde se encuentra el archivo y editarlo.

```
cd /usr/cygnus/conf
nano agent_test.conf
```

   Y verificar que la siguiente línea no existe o en el caso de que existiera está comentada:

```
cygnusagent.sinks.hdfs-sink1.attr_persistence = local
```

4. Ejecutar **`Sim.java`** y establecer los parámetros de simulación (seleccionar **modo Local** y establecer **1 proceso**) y pulsar sobre el botón **`init`**. No cerrar la interfaz.
5. Arrancar Cygnus en un terminal, en este caso en el equipo del DSIE:

```
/usr/cygnus/bin/cygnus-flume-ng agent --conf /usr/cygnus/conf/ -f
/usr/cygnus/conf/agent_test.conf -n cygnusagent -Dflume.root.logger=INFO,console
```

   Tras esto, Cygnus estará inicializado y a la espera de recibir datos del *Context Broker* para procesarlos.

6. Comprobar la creación de entidades y suscripciones mediante los siguientes comandos en un terminal independiente del *Context Broker* (opcional):

```
mongo orion
>db.entities.count()
*aparecerá el número de entidades creadas*
>db.csubs.count()
*aparecerá el número de suscripciones creadas*
exit()
```

7. Lanzar la simulación en la interfaz gráfica pulsando sobre el botón **`Start`**.
8. Una vez finalizada, comprobar la existencia del fichero **`source.txt`** (fichero creado por la clase **`OrionRestHandler.java`** y perteneciente a la información procesada por el *Context Broker*) en la ruta /root de la máquina que ejecuta Cygnus, en este caso en FIWARE.
9. Parar el *Context Broker* mediante la combinación Ctrl+C en su terminal.
10. En el mismo terminal del *Context Broker*, acceder a la ruta donde se encuentra **`HiveBasicClient`**:

```
cd /usr/fiware-connectors/resources/hive-basic-client/
```

12. En el caso de que aún no se haya compilado el programa **`HiveBasicClient.java`**, realizar su compilación mediante el comando:

```
mvn package
```

Tras compilar el código de la clase, ejecutar la siguiente línea de comandos para obtener finalmente el fichero de texto con los datos de la simulación captados por Cosmos:

```
mvn exec:java -Dexec.args="<Número de sensores> <Periodo>"
```

Los parámetros de entrada <Número de sensores> y <Periodo> se deberán establecer de acuerdo a la configuración realizada en la interfaz de simulación.

6.5.2 Prototipo de Pruebas

En primer lugar, se necesita una ventana terminal en el lado del *Context Broker* y otra ventana terminal para la ejecución de Cygnus, situado en el equipo del DSIE.

Los pasos para realizar una simulación son los siguientes:

1. Arrancar los servidores necesarios para la realización de la prueba (ejecución de **`Server.java`** en los equipos del LSI-1). Se da por hecho de que estos equipos están conectados a la red VPN mediante el *software* OpenVPN.
2. Eliminar el contenido de la base de datos del *Broker* a través de su terminal. Para ello, ejecutar los siguientes comandos:

```
mongo orion
> db.dropDatabase()
{ "dropped" : "orion", "ok" : 1 }
> exit
```

3. Arrancar el *Context Broker* mediante la ejecución del siguiente comando:

```
contextBroker -fg
```

Tras ello, el Context Broker se inicializará y quedará escuchando peticiones de creación/suscripción de entidades así como la recepción de nuevos datos. Todos los eventos que se produzcan serán imprimidos en pantalla.

4. Asegurarse de que el fichero **`agent_test.conf`** de Cygnus está configurado para el almacenamiento local. Para ello acceder al directorio donde se encuentra el archivo y editarlo.

```
cd /usr/cygnus/conf
nano agent_test.conf
```

Y verificar que la siguiente línea existe y no está comentada:

```
cygnusagent.sinks.hdfs-sink1.attr_persistence = local
```

5. Ejecutar **`Sim.java`** y establecer los parámetros de simulación. Pulsar sobre el botón **`init`**. No cerrar la interfaz.

6. Arrancar Cygnus en un terminal del equipo del DSIE mediante el siguiente comando:

```
/usr/cygnus/bin/cygnus-flume-ng agent --conf /usr/cygnus/conf/ -f
/usr/cygnus/conf/agent_test.conf -n cygnusagent -Dflume.root.logger=INFO,console
```

   Tras esto, Cygnus estará inicializado y a la espera de recibir datos del *Context Broker* para procesarlos.
7. Comprobar la creación de entidades y suscripciones mediante los siguientes comandos en un terminal independiente del *Context Broker* (opcional):

```
mongo orion
>db.entities.count()
*aparecerá el número de entidades creadas*
>db.csubs.count()
*aparecerá el número de suscripciones creadas*
exit()
```

8. Lanzar la simulación en la interfaz gráfica pulsando sobre el botón **`Start`**.
9. Una vez finalizada, comprobar la existencia de los ficheros **`source.txt`** (fichero creado por la clase **`OrionRestHandler.java`** y perteneciente a la información procesada por el *Context Broker*) y **`sim.txt`** (fichero creado por la clase **`OrionHDFSSink.java`** y perteneciente a la información procesada tras Cygnus) en la ruta **`/root`** de la máquina que ejecuta Cygnus, en este caso el equipo del DSIE.
10. Parar el *Context Broker* mediante la combinación Ctrl+C en su terminal.

6.6 Procesado de datos con RStudio

Para el procesado de datos es necesario crear un proyecto en RStudio (*File→New Project...→Existing Directory*). En el mismo, hay que crear un fichero del tipo R Markdown donde se implementa el código R (*File→New File→R Markdown...*).

El código implementado en el fichero permite el análisis de los datos de simulación. Dicho código puede dividirse en 5 fragmentos atendiendo a su funcionalidad:

1. **Librerías**: instalación y carga de las librerías necesarias para el correcto funcionamiento del código.
2. **Creación de variables**: creación y configuración de variables necesarias para el análisis.
3. **Carga y adaptación de los datos de simulación**: implementación de la carga de los datos obtenidos de las simulaciones a través del fichero de texto generado.
4. **Elección del mejor y peor hilo**: representación gráfica del comportamiento del mejor y peor hilo asociado a los escenarios multihilo.
5. **Representación gráfica de los datos**: conjunto de gráficas resultantes del análisis.

A continuación, se realiza una descripción más detallada de estas funcionalidades.

6.6.1 Librerías

Las librerías necesarias para el tratamiento de los datos son **`lubridate`**, **`dplyr`**, **`ggplot2`** y **`tidyr`**. El procedimiento para la instalación de estas librerías en RStudio es el siguiente:

Hacer *click* sobre *Packages* en la parte inferior derecha de la pantalla y, posteriormente sobre *Install*. A continuación, se despliega una ventana de instalación que permite indicar qué librerías se desean instalar. Para instalar todas las librerías al mismo tiempo, hay que escribir sus nombres separados por espacios o por comas y pulsar sobre el botón *Install*.

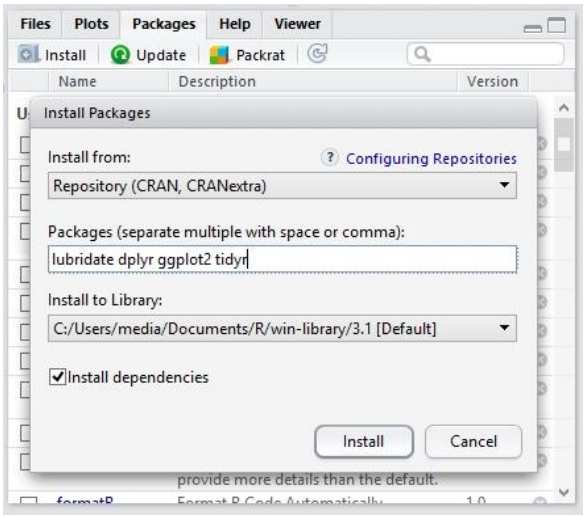


Figura 6.31. Instalación de librerías en RStudio

Una vez instaladas, solo resta realizar la carga en el fichero. La carga de librerías en RStudio se realiza a través de la instrucción **`library`**.

```
```{r}
library("lubridate")
library("dplyr")
library("ggplot2")
library("tidyr")
```
```

6.6.2 Creación de variables

Se trata del código encargado de la creación de variables para el cálculo y procesado de cierta información del fichero. Las variables que han sido creadas son las siguientes:

- **hilos**: representa el número de hilos por equipo de la simulación de la que se pretende realizar estudio.
- **procesos**: número de procesos que han participado en la simulación.
- **periodo**: periodo de tiempo en milisegundos establecido en la simulación.
- **max_datos_sensor**: número máximo de datos que un sensor puede llegar a enviar en la simulación. Como las simulaciones se truncan en 5 segundos, esta variable recoge el número de datos máximo que un sensor puede enviar en 5 segundos de acuerdo al periodo establecido.
- **sensores**: número total de sensores resultado del producto entre el número de hilos y el número de procesos.
- **max_datos_sim**: número máximo de datos de la simulación resultado del producto entre el número de sensores y el número máximo de datos por sensor.

Las variables **hilos**, **procesos** y **periodo** se configuran a mano cada vez que se realiza el estudio estadístico de una simulación. Como ejemplo, se muestra el código R asociado a la creación de variables para una simulación con 2 hilos, 8 procesos y un periodo de 200ms.

```
```{r}
hilos <- 2;
procesos <- 8;
periodo <- 200;
max_datos_sensor <- round( (1000/periodo)*5 );
sensores <- hilos*procesos;
max_datos_sim <- sensores * max_datos_sensor;
```
```

6.6.3 Carga y adaptación de los datos de simulación

Código que realiza la carga de los datos de la simulación a ser estudiados. Las gráficas resultantes consisten en una comparación entre las representaciones de los datos obtenidos en la simulación y los datos de una simulación ideal. Por ello, se necesitan cargar dos conjuntos de datos, uno que contiene los datos de la simulación y otro que contiene los datos de una hipotética simulación ideal.

Se considera como curva ideal la resultante de los datos generados por la aplicación, siendo simulado un solo sensor con la configuración de periodo acorde con la prueba con la que se pretende comparar. Así, al no haber ningún tipo de retardo de envío en la aplicación desarrollada en Eclipse (*Context Producer*), la generación de datos se hace de manera casi exacta con el periodo de tiempo especificado. Esta simulación ideal será desplazada en tiempo hasta coincidir con el instante en el que se produjo la simulación que se pretende estudiar.

Los datos de la simulación se almacenan en el conjunto de datos llamado **`Fiware`**, mientras que los datos de la simulación ideal se almacenaran en **`FiwareIdeal`**. El proceso de carga de datos es similar para ambos casos con la salvedad de que cada uno hace referencia a su correspondiente fichero de texto. A continuación, se detalla la carga de datos para el conjunto **`Fiware`**.

La instrucción de RStudio que permite la carga de datos desde un fichero de texto es **`read.table`**. La característica fundamental que necesita conocer esta instrucción es la separación de los datos dentro del fichero de texto, con el fin de poder separar por columnas cada dato. En el caso de los ficheros generados en las pruebas realizadas, los datos están separados por comas. Como ejemplo, se muestra la instrucción completa para la carga de datos de la simulación referente a 2 hilos en 8 equipos con un periodo de 200ms y sin *payload*:

```
#Cargar desde fichero
Fiware <- read.table(file = "data/2hilos_8equipos_200ms_payload0.txt",
                     header=FALSE, sep=",", stringsAsFactors = FALSE)
```

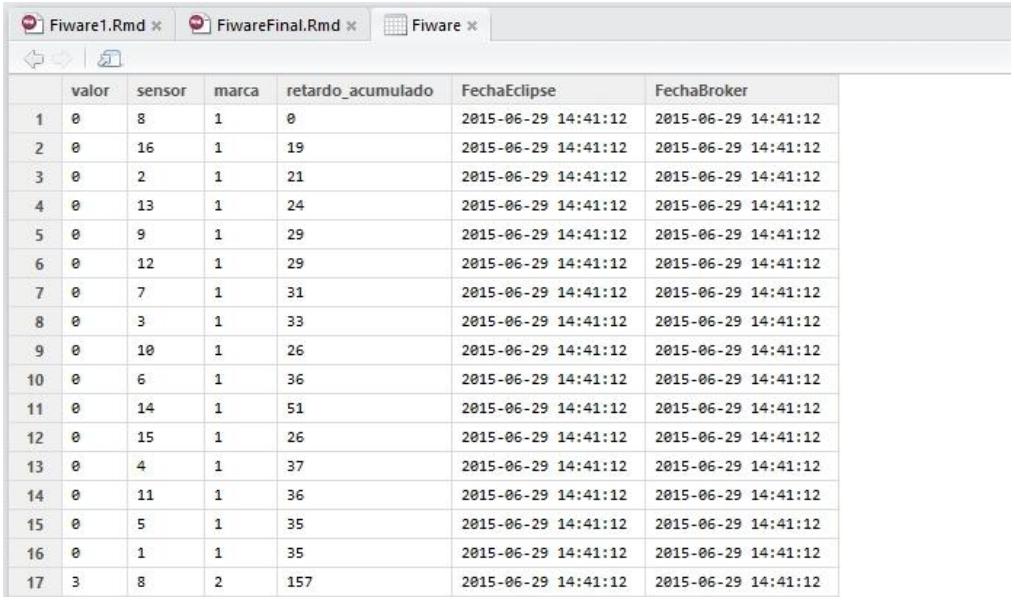
Por defecto, RStudio llama a cada una de las columnas con nombres como V1, V2, V3,... Por lo tanto, se necesita renombrar estas columnas para poder realizar el estudio y el tratamiento de la información de una manera más cómoda. Esta labor se realiza a través de la instrucción **`rename`**.

```
#Renombrar variables
Fiware <- rename(Fiware, valor = V1, sensor = V2, marca = V3,
                 fechaEclipse = V4, horaEclipse = V5, fechaBroker = V6,
                 horaBroker = V7,retardo_acumulado = V8)
```

Una vez cargados los datos, ciertas columnas como las referentes a las fechas y horas necesitan ser adaptadas a un formato de datos adecuado para RStudio. Para ello, se unifican cada par Fecha/Hora en una única columna y se le da el formato "*Año-Mes-Día Hora:Minuto:Segundos.Milisegundos*". Tras este proceso, las columnas que contenían fechas y horas pueden ser eliminadas. La unificación de columnas se realiza a través de la instrucción **`paste`**, mientras que la transformación a formato Fecha/Hora se realiza mediante **`strptime`**.

```
#Dar formato a fecha y hora. Para ello creamos una nueva columna que unifica
#fechaEclipse y horaEclipse
Fiware$FechaEclipse <- paste(Fiware$fechaEclipse, Fiware$horaEclipse)
Fiware$FechaEclipse <- as.POSIXct(strptime(Fiware$FechaEclipse, "%Y-%m-%d %H:%M:%OS"))
#Eliminamos las viejas columnas separadas y dejamos la nueva unificada y formateada
Fiware$fechaEclipse <- NULL
Fiware$horaEclipse <- NULL
#Misma operación con fechaBroker y horaBroker
Fiware$FechaBroker <- paste(Fiware$fechaBroker, Fiware$horaBroker)
Fiware$FechaBroker <- as.POSIXct(strptime(Fiware$FechaBroker, "%Y-%m-%d %H:%M:%OS"))
Fiware$fechaBroker <- NULL
Fiware$horaBroker <- NULL
```

Tras esta acción, se ha conseguido cargar los datos de manera satisfactoria estando preparados para ser tratados por el *software* RStudio. Los datos del conjunto **`Fiware`** tendrán una estructura similar a la mostrada en la Figura 6.32.


Fiware1.Rmd × FiwareFinal.Rmd × Fiware ×

	valor	sensor	marca	retardo_acumulado	FechaEclipse	FechaBroker
1	0	8	1	0	2015-06-29 14:41:12	2015-06-29 14:41:12
2	0	16	1	19	2015-06-29 14:41:12	2015-06-29 14:41:12
3	0	2	1	21	2015-06-29 14:41:12	2015-06-29 14:41:12
4	0	13	1	24	2015-06-29 14:41:12	2015-06-29 14:41:12
5	0	9	1	29	2015-06-29 14:41:12	2015-06-29 14:41:12
6	0	12	1	29	2015-06-29 14:41:12	2015-06-29 14:41:12
7	0	7	1	31	2015-06-29 14:41:12	2015-06-29 14:41:12
8	0	3	1	33	2015-06-29 14:41:12	2015-06-29 14:41:12
9	0	10	1	26	2015-06-29 14:41:12	2015-06-29 14:41:12
10	0	6	1	36	2015-06-29 14:41:12	2015-06-29 14:41:12
11	0	14	1	51	2015-06-29 14:41:12	2015-06-29 14:41:12
12	0	15	1	26	2015-06-29 14:41:12	2015-06-29 14:41:12
13	0	4	1	37	2015-06-29 14:41:12	2015-06-29 14:41:12
14	0	11	1	36	2015-06-29 14:41:12	2015-06-29 14:41:12
15	0	5	1	35	2015-06-29 14:41:12	2015-06-29 14:41:12
16	0	1	1	35	2015-06-29 14:41:12	2015-06-29 14:41:12
17	3	8	2	157	2015-06-29 14:41:12	2015-06-29 14:41:12

Figura 6.32. Estructura final de datos en RStudio.

La carga y adaptación de los datos referentes a la curva ideal se realiza de la misma manera con la diferencia de que se almacenan en el conjunto de datos llamado **`FiwareIdeal`**. No obstante, tal y como se ha comentado, estos datos necesitan ser desplazados en el tiempo para que puedan coincidir con los datos de la simulación que se pretende evaluar. Para ello, se necesita conocer la diferencia entre tiempos de generación de datos (FechaEclipse) entre ambos conjuntos y sumar dicha diferencia a los tiempos de los datos de **`FiwareIdeal`**. Las líneas de código que implementan esta acción son las siguientes:

```
diferencia <- Fiware$FechaEclipse[1] - FiwareIdeal$FechaEclipse[1]
FiwareIdeal <- FiwareIdeal %>%
               arrange(sensor) %>%
               group_by(sensor) %>%
               mutate(FechaEclipse = FechaEclipse + diferencia[1])
```

6.6.4 Elección del mejor y peor hilo

La elección del mejor y peor hilo viene dada por las siguientes características ordenadas de mayor a menor importancia:

- Número de paquetes perdidos.
- Número de paquetes desordenados.
- Retardo de procesado en el *Broker*.

En primer lugar, se crea una variable **`estudio`** que adquiere parámetros como el número de paquetes por sensor, pérdidas, paquetes desordenados, retardo de procesado del *Broker* entre paquetes y retardo producido en el envío Eclipse-Broker. De esta forma, se consigue agrupar en una sola entidad toda la información relevante para la elección de los casos extremos. El siguiente fragmento de código ilustra este hecho:

```
estudio <- Fiware %>%
  group_by(sensor) %>%
  summarise(paquetes = n(),
            desorden = sum(diff(marca)<0),
            RBroker = mean(diff(FechaBroker)),
           REclipseBroker = mean(FechaBroker-FechaEclipse))

estudio <- estudio %>%
  mutate(perdidas = max_datos_sensor - paquetes)
```

Una vez obtenida esta variable, el mejor sensor será aquel que tenga el menor número de pérdidas. En el caso de que varios sensores tengan el mínimo de pérdidas, será mejor el que menos datos desordenados tenga. Si por alguna razón siguiera habiendo sensores con el mismo número de datos desordenados será mejor el que menos retardo obtenga en el Broker.

De la misma forma, el peor sensor será aquel que mayor número de pérdidas tenga, así como mayor número de datos desordenados y mayor retardo en el Broker en el caso de que hubiera varios sensores con las mismas pérdidas.

Para encontrar el mejor/peor sensor se utiliza la instrucción **`filter`**, capaz de filtrar datos que satisfagan una determinada condición impuesta dentro de un conjunto de datos. Concretamente, se ha implementado el siguiente código:

```
mejor_sensor <- estudio %>% filter(perdidas == min(perdidas, na.rm = TRUE))
mejor_sensor <- mejor_sensor %>% filter(desorden == min(desorden, na.rm = TRUE))
mejor_sensor <- mejor_sensor %>% filter(RBroker == min(RBroker, na.rm = TRUE))

peor_sensor <- estudio %>% filter(perdidas == max(perdidas, na.rm = TRUE))
peor_sensor <- peor_sensor %>% filter(desorden == max(desorden, na.rm = TRUE))
peor_sensor <- peor_sensor %>% filter(RBroker == max(RBroker, na.rm = TRUE))
```

6.6.5 Representación gráfica

La representación gráfica de los datos viene dada por dos tipos de gráfica. Por un lado, se realizan gráficas de líneas para ver la evolución temporal de producción y procesado de los datos frente a la curva ideal, y por otro lado se realizan gráficas de barras que muestran la relación de paquetes perdidos por sensor.

El eje X de las gráficas de líneas se corresponde con el tiempo de simulación expresado en segundos, mientras que el eje Y se corresponde con los valores mencionados.

En las simulaciones con un solo hilo se representa el primer tipo de gráfica mostrando dos curvas, una referente a la rampa ideal de la simulación en la que se reflejan todos los valores con su respectivo periodo de tiempo entre datos y la curva del *Broker* que evalúa su comportamiento obtenida como resultado del procesamiento de los datos por parte de este, pudiendo haber pérdidas y retardos variables. En estas simulaciones con un único sensor, el mejor y peor hilo lógicamente es el mismo, por lo que en realidad se imprimen dos curvas superpuestas aunque el resultado visual sea una única curva.

En las gráficas de simulaciones multihilo se representan tres curvas: la correspondiente a la rampa ideal, similar a la explicada anteriormente, y las curvas producidas por el mejor y peor sensor de la simulación. Además, en este tipo de simulaciones también se representa un diagrama de barras con la relación de paquetes perdidos por sensor.

La representación gráfica en RStudio se realiza mediante la instrucción **`ggplot`** de la librería **`ggplot2`**. Dentro de un objeto **`ggplot`**, la representación de los puntos de datos se realiza con **`geom_point`**, mientras que la unión de esos puntos con una línea se realiza con **`geom_line`**. Las gráficas de barras se realizan mediante **`geom_bar`**, y la adición de texto en cada uno de los datos mediante **`geom_text`**.

El código asociado a la representación del primer tipo de gráfica (línea ideal + mejor sensor + peor sensor) es el siguiente:

```
  ggplot(data = Fiware, aes(colour = factor(sensor))) +
  geom_line(data = FiwareIdeal, aes(x = FechaEclipse, y = valor)) +
  geom_point(data= FiwareIdeal,aes(x = FechaEclipse, y = valor)) +

  geom_line(data = mejor_sensor,aes(x = FechaBroker, y = valor)) +
  geom_point(data = mejor_sensor,aes(x = FechaBroker, y = valor)) +

  geom_line(data = peor_sensor,aes(x = FechaBroker, y = valor)) +
  geom_point(data = peor_sensor,aes(x = FechaBroker, y = valor))+ xlab("Segundo")
+ ylab("Valor")
```

Por otro lado, el código asociado a la representación gráfica madiante barras es el siguiente:

```
plot <- ggplot(data = estudio,aes(x = sensor, y = perdidas))
plot +geom_bar(stat='identity')
     +geom_text(data=estudio,aes(label=perdidas),vjust=-0.5,size = 3)
```

7 Protocolo de Pruebas y Resultados.

Este capítulo contiene tanto la descripción del protocolo de pruebas que ha condicionado las simulaciones de los prototipos estudiados en el capítulo 6, como la valoración de los resultados obtenidos.

7.1 Protocolo de pruebas

En primer lugar, se han realizado pruebas acerca del funcionamiento del Prototipo Inicial de Laboratorio que han permitido estudiar el comportamiento de los GE estudiados en el capítulo 5. Además, se han evaluado nuevas vías para mejorar su eficiencia como la incorporación de varios canales junto con el algoritmo de selección *Round Robin* en Cygnus.

Posteriormente, se ha testeado el Prototipo de Pruebas para evaluar el rendimiento del *Context Broker*. Para conseguir tal fin, se define una serie de escenarios que permiten conocer de manera eficiente cuáles son las limitaciones que posee la plataforma FI-WARE relativas a la recepción, procesado y envío de información de contexto. Dichos escenarios son:

- **Escenario 1:** deducción del periodo mínimo de envío entre datos.
- **Escenario 2:** influencia del *payload* en el procesado de la información.
- **Escenario 3:** escalabilidad de sensores utilizando varios ordenadores con un único proceso e hilo.
- **Escenario 4:** escalabilidad de sensores en un ordenador con un único proceso y diferentes hilos.
- **Escenario 5:** escalabilidad de sensores variando el número de ordenadores utilizando un único proceso y 12 hilos.
- **Escenario 6:** escalabilidad de sensores variando el número de ordenadores con un único proceso y 2 hilos.

Las principales medidas a obtener de estas pruebas son: el **periodo mínimo (límite)** de envío entre datos del Prototipo de Pruebas, influencia del **payload** en el tratamiento de la información y el **número máximo de sensores** a simular siendo el **periodo mínimo**.

El tiempo de simulación es siempre de 5 segundos con el fin de conseguir una alta resolución en el tratamiento de la información. Además, las simulaciones se considerarán válidas cuando las pérdidas de datos sean inferiores al 5% del total de datos que se deberían recibir. Por otro lado, los valores de los datos consistirán en incrementos de 3, con el fin de poder discernir pérdidas y paquetes desordenados en las gráficas. El eje X de las gráficas se corresponderá con el tiempo de simulación expresado en segundos, mientras que el eje Y serán los valores de los datos obtenidos de la información de contexto enviada al Broker por los sensores virtuales.

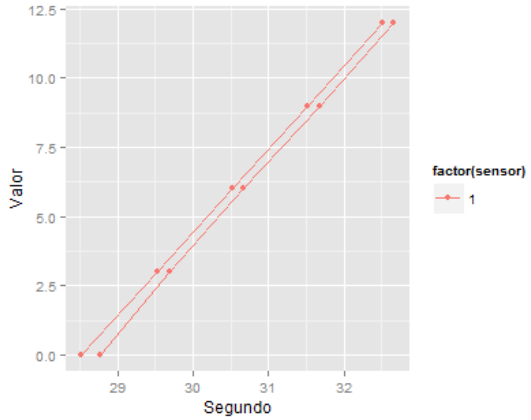
En las gráficas asociadas a un sensor se muestran dos curvas, una referente a la rampa ideal de la simulación, en la que se reflejan todos los valores con su respectivo periodo de tiempo entre datos, y la curva del datos recibidos en la *source* de Cygnus, que permite evaluar el rendimiento del *Broker*, pudiendo existir pérdidas y retardos variables. En las gráficas de asociadas a las simulaciones de varios sensores se representan tres curvas. La correspondiente a la rampa ideal, similar a la explicada anteriormente, y las curvas producidas por el mejor y el peor sensor de la simulación. La elección del mejor y peor sensor viene dada por las siguientes criterios ordenados de mayor a menor importancia: (1) número de paquetes perdidos, (2) número de paquetes desordenados y (3) retardo de procesado en el *Broker*. Además, también se representa un diagrama de barras que ilustra la relación de paquetes perdidos por sensor.

7.2 Resultados

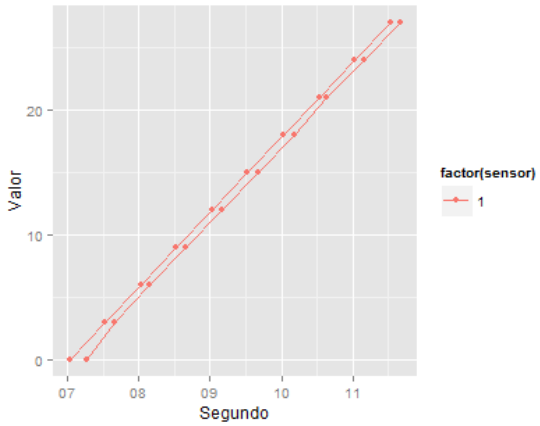
7.2.1 Escenario 1

En este escenario se pretende deducir cual es el periodo mínimo de tiempo que puede transcurrir entre envíos realizados al *Broker*. Por ello, se realiza una serie de simulaciones partiendo de una configuración inicial del periodo de 1 segundo y disminuyéndolo progresivamente hasta llegar a 150ms.

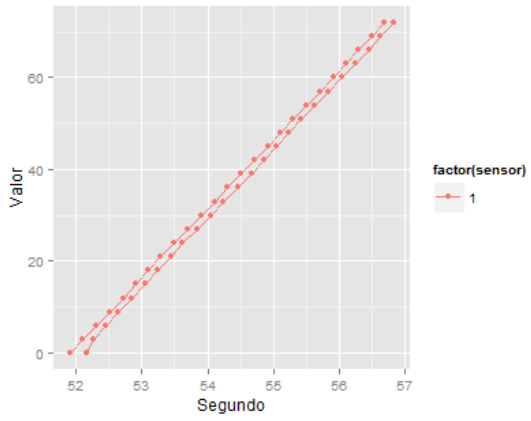
La gráfica obtenida para un segundo de periodo (1000ms) es la siguiente:



Como esta simulación es satisfactoria, se procede a disminuir el periodo de tiempo. Así, la gráfica obtenida para 500ms de periodo es la siguiente:

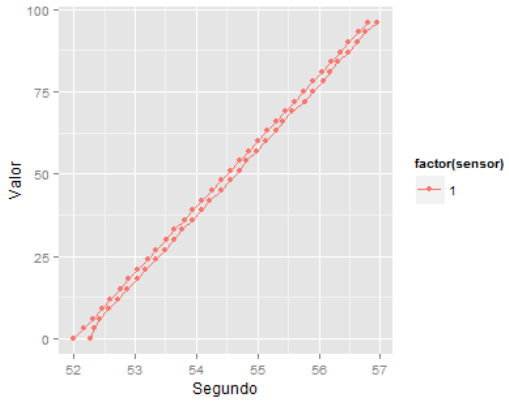


Es evidente que la simulación ha resultado satisfactoria, puesto que las gráficas son muy similares y no se aprecia pérdida de datos. La gráfica obtenida para 200ms es la siguiente:



En la gráfica anterior, se puede apreciar una mayor cantidad de datos, concretamente 25, como resultado de enviar 5 datos por segundo durante 5 segundos.

Finalmente, la gráfica obtenida para 150ms de periodo es la siguiente:

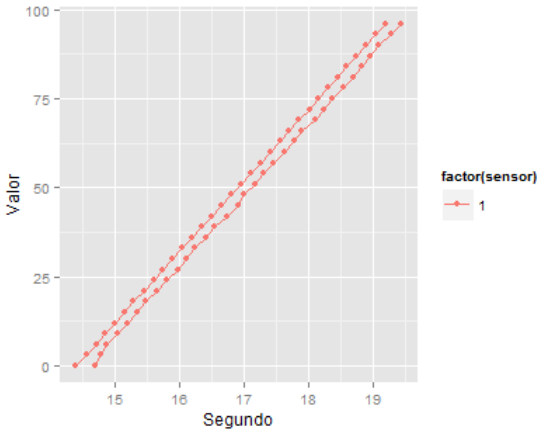


Con este periodo de tiempo configurado el comportamiento del *Broker* es satisfactorio. Además, la aplicación desarrollada en Eclipse necesita un periodo aproximado de 120 ms para enviar los datos (debido al funcionamiento de la clase **`HTTPURLConnection`** descrito en el apartado 6.3.1.3), por lo que todo periodo igual o inferior al mismo dará como resultado simulaciones fallidas. Por este motivo, y para dejar un cierto margen, a priori se considera 150ms el periodo límite de las simulaciones.

7.2.2 Escenario 2

En este escenario de simulación se evalua como afecta el *payload* en el procesado de la información por parte del *Broker*. Para ello, se realizan 2 simulaciones que tienen como parámetros los últimos 2 periodos configurados en el escenario anterior (150ms y 200ms) y un *payload* de 1KB.

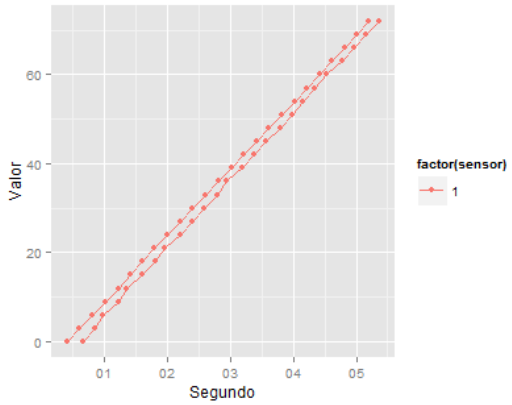
La simulación de 150ms y 1KB de *payload* da como resultado la siguiente gráfica:



Aunque sigue habiendo ausencia de pérdidas, se empiezan a apreciar diferencias con respecto a la gráfica anterior sin *payload*, observándose un mayor *gap* entre la gráfica ideal y la del *Broker* así como un retardo con mayor variabilidad entre los datos.
Además, el tiempo entre iteraciones de la aplicación desarrollada en Eclipse ha aumentado debido a la presencia del payload de los 120ms hasta unos 150-170ms aproximadamente, por lo que se descarta utilizar un periodo de tiempo de 150ms para el resto de las simulaciones. Esto provocaría la acumulación de cierto retardo que trastocaría los resultados convirtiéndolos en no válidos.

Así, se restablece el periodo mínimo a 200ms para la ejecución de las simulaciones y, como el aumento del *payload* influye en el tiempo de iteración de la aplicación del Prototipo de Pruebas, se establece un payload máximo de 1KB.

La gráfica resultante al simular con los parámetros seleccionados (200ms de periodo y 1KB de *payload*) es la siguiente:

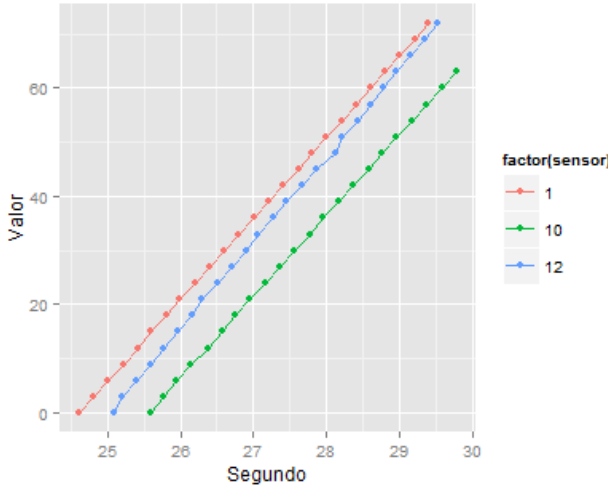


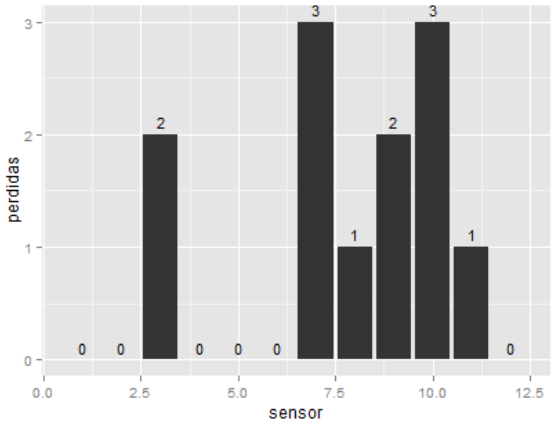
Como era de esperar, no existen pérdidas del mismo modo que en la simulación anterior (150ms, 1KB). Sin embargo, ha disminuido el retardo entre la rampa ideal y la rampa de simulación que era el propósito de esta simulación.

7.2.3 Escenario 3

Este escenario modela el escalado de sensores utilizando varios ordenadores con un único proceso e hilo cuyo propósito es la evaluación del número máximo de sensores virtuales dedicados (SVD) capaces de ser simulados, siendo un sensor virtual dedicado un ordenador en el que se ejecuta uno/varios proceso/s Eclipse que simula/n el comportamiento de un sensor (un hilo).

Como en el laboratorio LSI1 hay 12 equipos, se empieza simulando 12 sensores virtuales dedicados, obteniendo las siguientes gráficas:





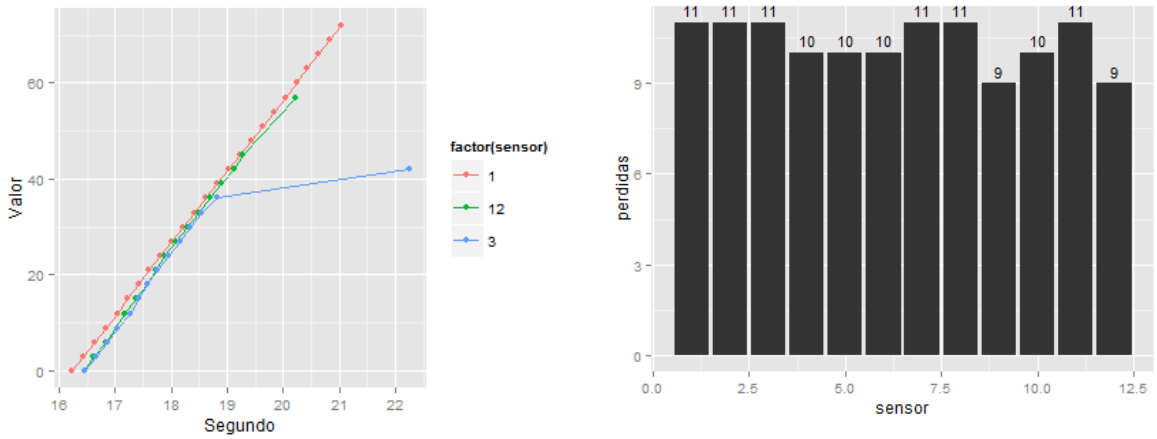
La simulación es válida porque las pérdidas no superan el 5% del total de paquetes. Los peores sensores han sido el 7 y el 10 con un total de 3 pérdidas cada uno, aunque el sensor número 10 ha obtenido un mayor retardo de procesado, siendo este peor de acuerdo a los criterios establecidos en el protocolo de pruebas.

En cuanto al mejor sensor, se observa que no se han producido pérdidas asociadas a los sensores 1, 2, 4, 5, 6 y 12, siendo el menor retardo de procesado el correspondiente al sensor número 12.

Debido a que se han obtenido buenos resultados y no hay más ordenadores en el laboratorio, se empieza a lanzar procesos por equipo con el fin de poder conocer la cantidad de sensores virtuales dedicados que se pueden llegar a simular. Por ello, la siguiente prueba a realizar consta de 11 equipos que ejecutan un solo proceso y de 1 equipo que ejecuta dos procesos. De esta forma se obtienen un total de 13 sensores virtuales dedicados.

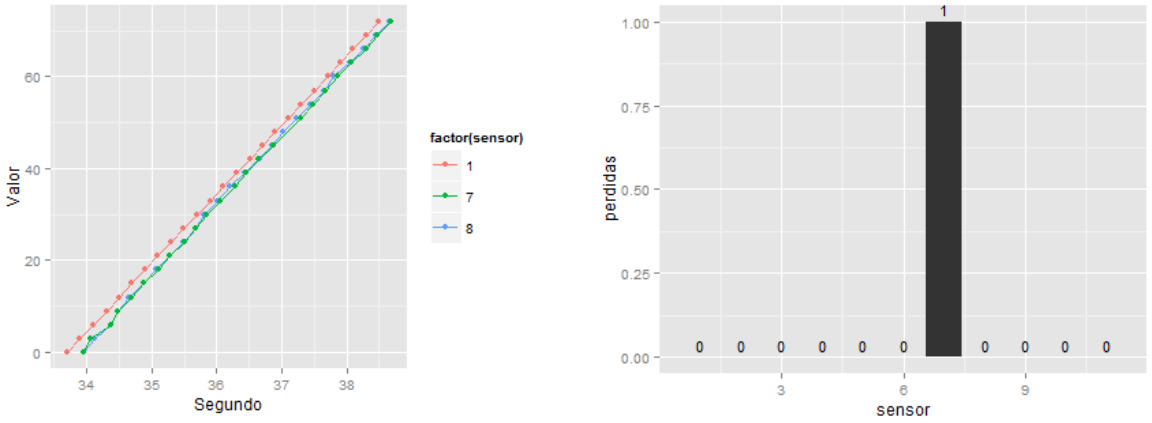
Tras realizarse un total de tres intentos de la prueba citada, no se han obtenido la cantidad mínima de datos necesarios para poder representar de manera objetiva el resultado de la simulación, debido a que no se ha recibido el dato con valor -4 o bien porque ha habido un fallo en la simulación y ha terminado antes de los 5 segundos. Estas son las condiciones de la clase **`OrionRestHandler.java`** que permitía volcar el contenido del **`BufferString`** en el fichero de texto **`source.txt`**. Este fichero no se ha obtenido en ninguno de los tres intentos. Por esta razón, esta simulación se considera no válida, siendo asumible que bajo estas condiciones el número máximo de sensores virtuales dedicados que FIWARE es capaz de soportar es 12.

Con objeto de ver si realmente afecta el *payload* y poder conocer el límite de sensores capaces de ser simulados con carga de datos añadida, se vuelve a lanzar la simulación de 12 sensores virtuales dedicados añadiéndole 1KB de *payload*, resultando las siguientes gráficas:



Claramente se observa el incremento significativo de paquetes perdidos. Tanto es así que la simulación no se considera como válida. El mejor sensor ha sido el 12, mientras que el peor ha sido el 3, apreciándose el significativo retardo existente entre los últimos datos procesados.

De un total de 300 datos que se esperaban recibir, solo se han recibido 176, por lo que claramente el *payload* influye en las simulaciones. No obstante, si en vez de 12 sensores se simulan 11 con los mismos parámetros, se obtienen resultados favorables tal y como muestran las siguientes gráficas, por lo que el límite con *payload* se sitúa en 11 sensores virtuales dedicados.

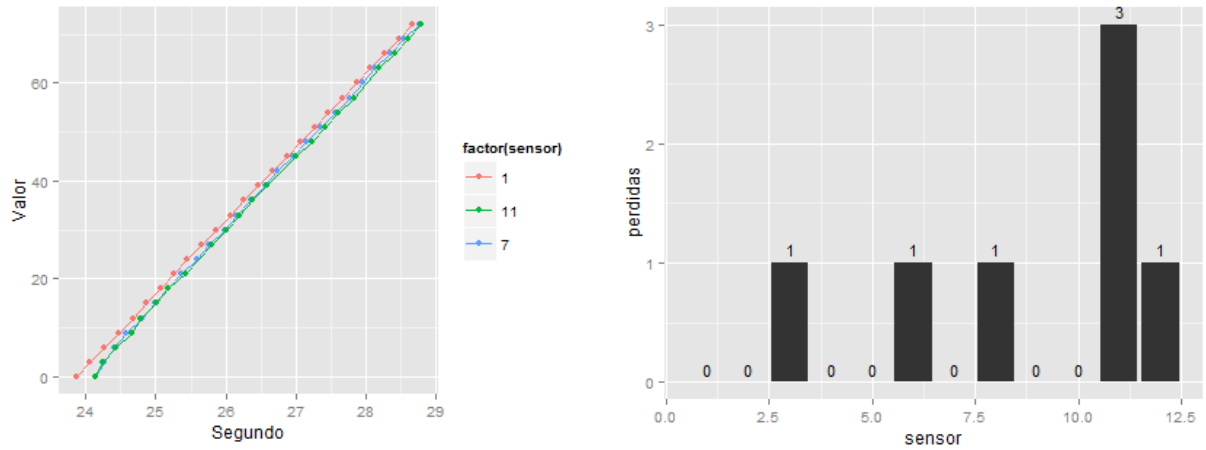


Además de acercarse al ideal, se observa una mínima diferencia entre el mejor y peor sensor. Cabe mencionar que solamente ha habido una pérdida, pues de 275 datos que se esperaban se han obtenido 274.

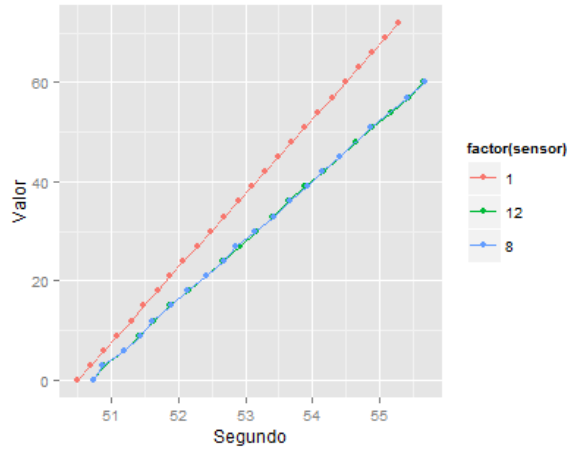
7.2.4 Escenario 4

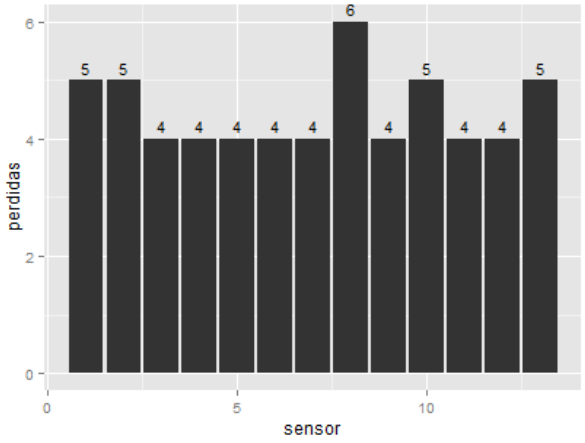
En este escenario se pretende conocer el número máximo de hilos que pueden simularse en un único ordenador con un único proceso. Las simulaciones son satisfactorias hasta 12 hilos, empezándose a obtener resultados no deseados a partir de 13 sensores virtuales. Estos límites serán utilizados en este escenario.

La simulación en un equipo con 12 hilos, un periodo de 200ms y sin *payload* produce las siguientes gráficas:



Siendo el mejor sensor el 7 y el peor claramente el 11 ya que conlleva el mayor número de pérdidas. Con la simulación de mismas características pero con un total de 13 hilos se obtiene los siguientes resultados en los que se observa un aumento significativo de la pérdida de paquetes por sensor además de un elevado retardo con respecto a la rampa ideal. Por ello, se deduce que el máximo número de hilos que se pueden simular por ordenador es 12.

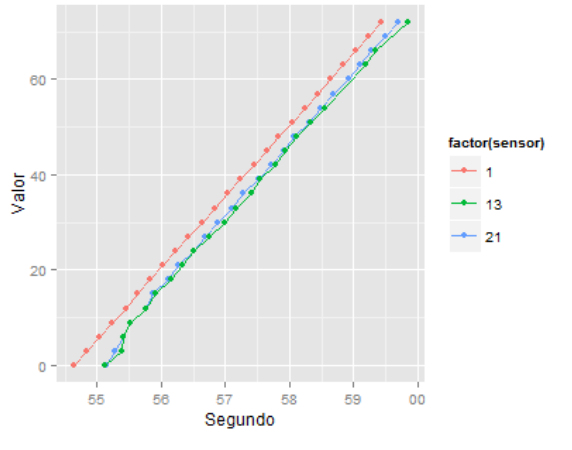


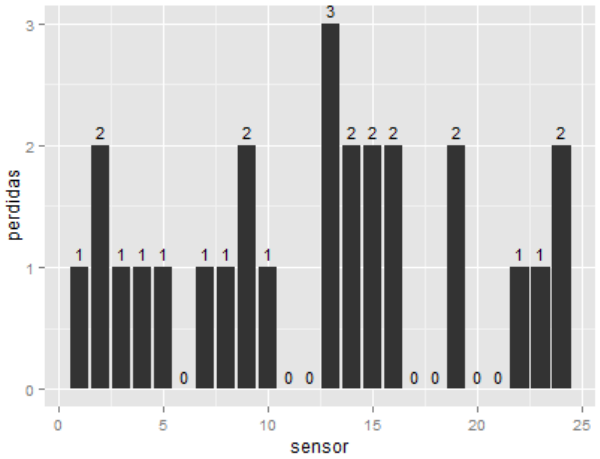


7.2.5 Escenario 5

Este escenario modela el escalado de sensores al variar el número de ordenadores utilizando un único proceso y 12 hilos. Así, se pretende evaluar cuantos equipos soporta el *Broker* enviando a la máxima tasa (200ms) y con el máximo número de hilos por ordenador (12 hilos).

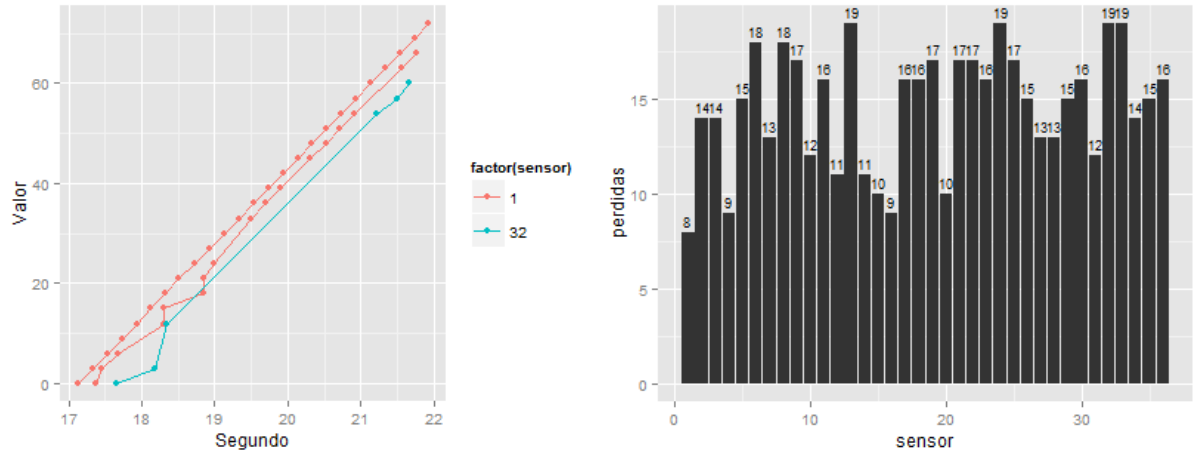
Primero, se replica la configuración realizada en el escenario 7.2.4 en otro ordenador más, obteniendo así un total de 24 hilos repartidos en 2 equipos con 12 hilos cada uno. Los resultados son los siguientes:





La simulación resulta satisfactoria. Se ha conseguido simular de 24 sensores virtuales ejecutándose de manera concurrente y enviando información de contexto al *Broker*.

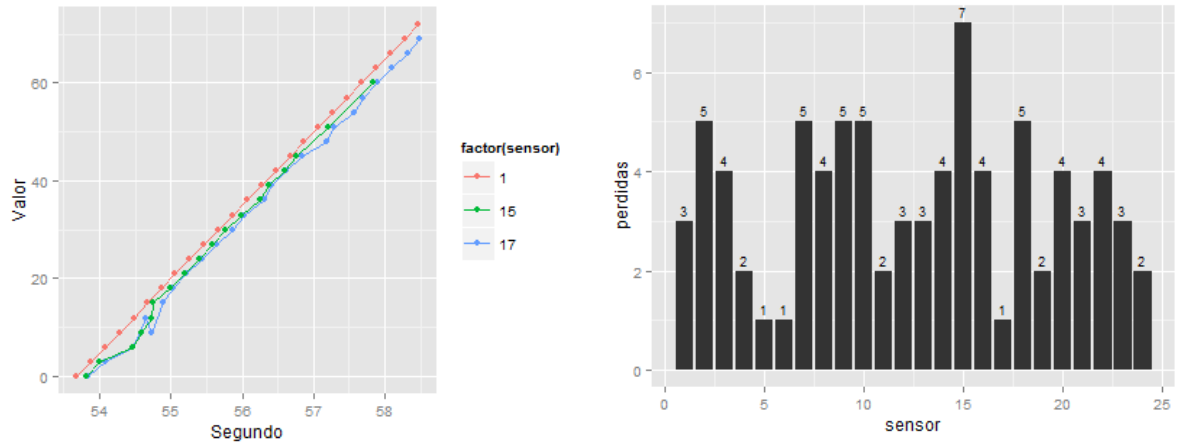
En la siguiente simulación se intenta llegar hasta 36 sensores al incorporar un nuevo equipo a la simulación. Por lo tanto, el escenario consiste en de 3 equipos que ejecutan 12 hilos cada uno. El resultado de la simulación no es satisfactorio ya que se puede observar la aparición de paquetes desordenados tanto en el mejor como en el peor hilo, además de existir pérdidas de hasta 19 paquetes. En esta simulación se esperaran obtener 900 datos de contexto, donde el máximo número de pérdidas admisible es de 45. Solamente con las pérdidas de los sensores 13, 24, 32 y 33 (con 19 pérdidas cada uno) ya se supera este número. Las gráficas resultantes son las siguientes:



7.2.6 Escenario 6

Este escenario modela el escalado de sensores al variar el número de ordenadores con un único proceso y 2 hilos. Con objeto de verificar el diferente comportamiento entre la concurrencia de procesos y la de hilos, además de evaluar cual de las dos ofrece una mayor flexibilidad, se realizan las mismas pruebas que en el escenario anterior con la diferencia de que ahora se simulan 2 hilos en 12 equipos frente a los 12 hilos en 2 equipos simulados anteriormente.

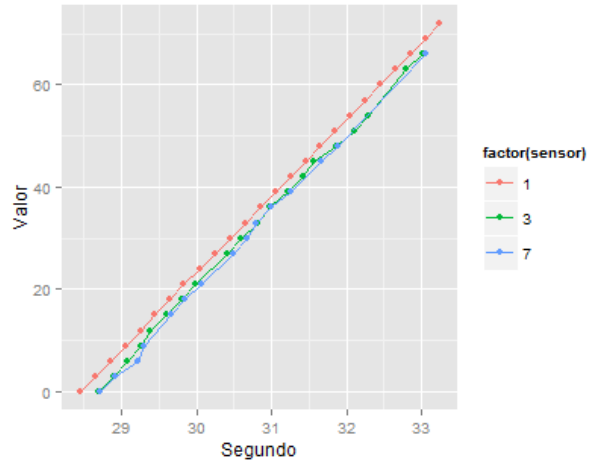
Gracias al resultado de estas pruebas podremos comparar el grado de flexibilidad de los hilos frente a los procesos en los equipos. Así, se comienza simulando los 24 hilos obtenidos anteriormente, siendo ahora la configuración de 2 hilos en 12 equipos diferentes. Los resultados son:

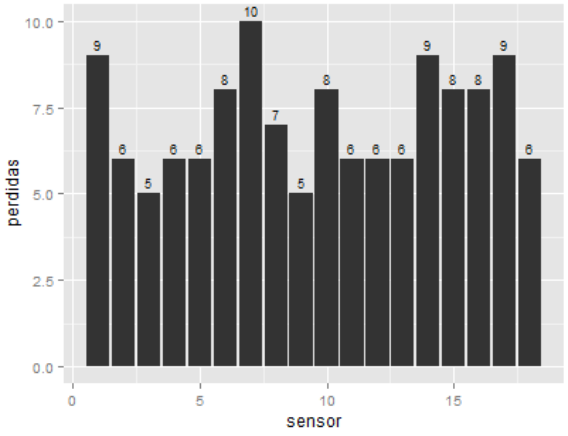


Como se puede observar la simulación no ha resultado satisfactoria, puesto que se pierde una gran cantidad de paquetes, además de llegar paquetes desordenados. Para conocer el límite, se procede a realizar simulaciones disminuyendo el número de equipos.

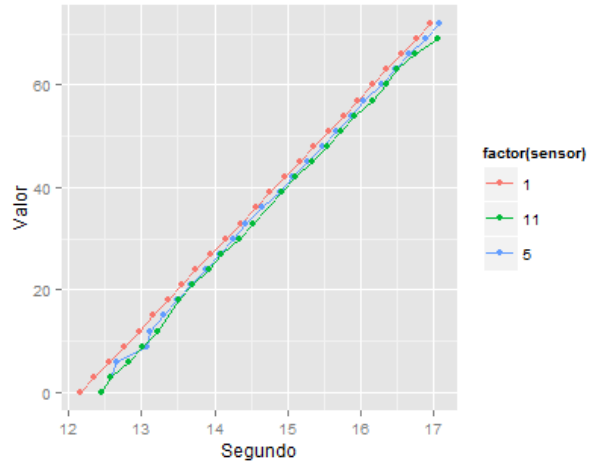
Siguiendo este procedimiento, se deduce que entre 8 y 9 equipos se encuentra el límite entre resultados satisfactorios y no satisfactorios (lo que equivale a 16 y 18 sensores virtuales respectivamente). Así, la simulación es satisfactoria con 8 equipos y no satisfactoria con 9 equipos. Como era de esperar, se obtienen mejores resultados simulando muchos hilos en pocos equipos que pocos hilos en muchos equipos (esto es debido a la concurrencia de los hilos).

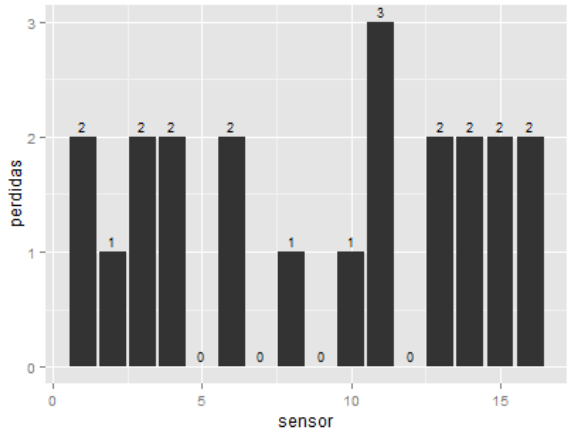
Las gráficas de 2 hilos ejecutándose en 9 equipos diferentes son:





Mientras que las gráficas asociadas a la simulación de 2 hilos en 8 equipos son:





Pudiéndose ver reflejado el buen funcionamiento en las mínimas pérdidas obtenidas en el diagrama de barras.

7.2.7 Resumen

Escenarios	Resultados
Escenario 1	Se considera 150ms el periodo límite de las simulaciones, debido a que la aplicación desarrollada en Eclipse necesita un periodo aproximado de 120 ms para enviar los datos.
Escenario 2	Se establece un payload máximo de 1KB. Se observa que el payload tiene influencia sobre el retardo en el envío de datos al Broker, llegando a ser de hasta 170ms en algunos casos, por lo que se reestablece el periodo límite de simulación a 200ms.
Escenario 3	El número máximo de sensores virtuales dedicados es de 12 tras no obtenerse la cantidad mínima de datos necesarios para poder representar de manera objetiva el resultado de la simulación en la prueba con 13 sensores virtuales dedicados.
Escenario 4	El número máximo de hilos que pueden ser simulados por equipo es de 12. Las pruebas realizadas con 13 hilos arrojan elevados retardos además de una gran cantidad de pérdidas de datos.
Escenario 5	24 sensores funcionando de manera concurrente en 2 equipos y 12 hilos cada uno. Los resultados de las pruebas realizadas con 3 equipos y 12 hilos (36 sensores) muestran una tasa elevada de paquetes desordenados en el Broker además de pérdidas superiores al límite del 5% especificado en los criterios de selección.
Escenario 6	El límite se sitúa entre 8 y 9 equipos (lo que equivale a 16 y 18 sensores respectivamente). Los resultados de la simulación dependen del estado de la red por lo que, aunque se han llegado a obtener buenos resultados en las simulaciones referentes a 9 equipos (18 sensores en total), las simulaciones con 8 equipos (16 sensores) han resultado más fiables tanto en términos de pérdidas, retardos y paquetes desordenados, siendo este el límite seleccionado.

7.3 Conclusiones

En primer lugar, se procede a la justificación del alcance de los hitos asociados al presente Proyecto Fin de Carrera que han sido planteados en el aparatado 1.2. Así, los aspectos fundamentales de tres de los componentes que conforman el capítulo *Data/Context Management* de la plataforma FI-WARE han sido abordados en el capítulo 5. Por otro lado, en el apartado 6.1 se realiza el diseño de los dos Prototipos de Laboratorio, describiéndose su funcionamientos en el apartado 6.5. Así, se ha analizado un Prototipo Inicial cuyo fin es el aprendizaje de la metodología a seguir en FIWARE para desarrollar aplicaciones en el ámbito de la IoT. Según estaba previsto, dicho prototipo incorpora una aplicación software genérica.

Además, se ha desarrollado y desplegado un Prototipo de Pruebas cuya arquitectura extiende a la del Prototipo Inicial y que ha permitido la evaluación del rendimiento del *Generic Enabler Context Broker* que constituye el *core* del capítulo FIWARE *Data/Context Management*. Este Prototipo de Pruebas es capaz de generar un fichero que contiene los datos asociados a la información de contexto recibida a la salida del *Broker*. De esta manera, el fichero generado es tratado y analizado mediante la herramienta RStudio para el cálculo de las siguientes medidas (apartado 6.6): retardo entre el envío de datos por el simulador de sensores y su recepción a la salida del *Broker* (sensor virtual-*source* de Cygnus) y relación de paquetes enviados/recibidos por este componente.

Finalmente, en el apartado 7 se describe el protocolo de pruebas utilizado para la evaluación de la plataforma FI-WARE.

Por otro lado, se considera oportuno razonar el empleo de la plataforma FIWARE en la Agricultura de Precisión a partir del estudio realizado acerca de su rendimiento. La Agricultura de Precisión es un método de estimar, evaluar y entender los cambios que se producen en los cultivos, cuyo objetivo es poder determinar con exactitud las necesidades de riego y de fertilizantes, las fases de desarrollo y de maduración de los productos, los puntos óptimos de siembra y de recolección, etc. Utilizando para ello las tecnologías de la información [45].

La aplicación y el despliegue de sensores dentro de la agricultura de precisión varía en función de las dimensiones del terreno a estudiar. Así pues, según estudios realizados por el Proyecto SICORI de la fundación Séneca y desarrollado en el DSIE de la Universidad Politécnica de Cartagena se deduce que en cultivos de mediano tamaño se tomarán como máximo 10 sensores para su estudio [46], mientras que para cultivos grandes pueden llegar a utilizarse más de 20 sensores para cubrir de manera satisfactoria el área de estudio [47]. En un estudio realizado en el Mar Menor, se utiliza un total de 3 subredes con 4 sensores cada una, habiendo un total de 12 sensores para el estudio [48].

Además del despliegue de sensores, es interesante conocer el tiempo de muestreo por parte de los dispositivos instalados. En la agricultura de precisión no es frecuente realizar un muestreo constante de los datos, es decir, el desarrollo de aplicaciones en tiempo real. Siguiendo con el mismo estudio realizado en el Mar Menor por parte del Proyecto SICORI, el muestreo de datos se realiza cada 20 minutos. Otros documentos del mismo grupo de investigación apuntan a periodos menores, como por ejemplo de 15 minutos, 10 minutos o incluso 5 minutos [49].

Los resultados de las simulaciones de los escenarios planteados en este trabajo han sido descritos en el apartado 7.2. Tal y como se puede observar, se han podido simular hasta 12 sensores virtuales dedicados con un intervalo de muestreo de 200ms en la plataforma FIWARE. Este resultado permite implementar en esta plataforma los escenarios mencionados anteriormente de manera satisfactoria. Presumiblemente, con periodos de muestreo superiores se puede llegar a desplegar un mayor número de sensores. Sin embargo, no se han realizado pruebas de escalabilidad de la plataforma. Dichos trabajos han sido propuestos en el apartado 8.

Como conclusión, las aplicaciones utilizadas en la agricultura de precisión no son de tiempo real y los periodos de muestreo varían en el orden de minutos debido a condicionantes como la duración de las baterías de los sensores. Los resultados obtenidos (12 sensores, ΔT=200ms) permiten afirmar que es posible el desarrollo e implementación de aplicaciones IoT en FIWARE para el grupo de escenarios estudiados.

8 Líneas futuras de investigación

A continuación, se proponen tres líneas futuras de investigación estrechamente relacionadas con el Trabajo Fin de Grado realizado. La primera consiste en implementar nuevas funcionalidades en el Prototipo de Pruebas, lo que permitiría la evaluación de parámetros diferentes del rendimiento, como es el caso de la escalabilidad. Esta nueva línea de investigación aportaría un mayor conocimiento acerca de la plataforma FIWARE.

Como segunda línea de investigación, se propone el estudio de los *Generic Enabler Complex Event Processing* (CEP) y Kurento que permiten respectivamente análisis de eventos y procesamiento de información multimedia en tiempo real. En este caso, el propósito perseguido sería el dominio de todos los componentes del capítulo *Data/Context Management* de FIWARE que permiten desarrollar una amplia variedad de aplicaciones en el ámbito de Internet de las Cosas.

Finalmente, la tercera línea de investigación conistiría en el estudio y la evaluación de plataformas de Internet de las Cosas alternativas a las estudiadas en el presente trabajo de investigación mediante una versión específica del Prototipo de Pruebas estudiado en FIWARE con el fin de realizar una comparación y evaluación objetiva de los resultados obtenidos entre ellas para su aplicación en la agricultura de precisión.

Anexos

Anexo I. Nimbits

Anexo I.I. Detalles de instalación

Nimbits ofrece la posibilidad de registrar los datos tanto en un servidor local como en un servidor remoto interno de la plataforma. Se opta por la segunda opción para poder exprimir al máximo las funcionalidades y posibilidades que brinda la plataforma en cuestión. Para realizar el registro en la plataforma a través de su *Cloud* interna es tan sencillo como ingresar en su página principal y hacer click en el apartado *Login* del *banner*. Una vez dentro solamente hay que introducir usuario y contraseña de una cuenta de Google para que automáticamente se cree y vincule a ella una cuenta de Nimbits.

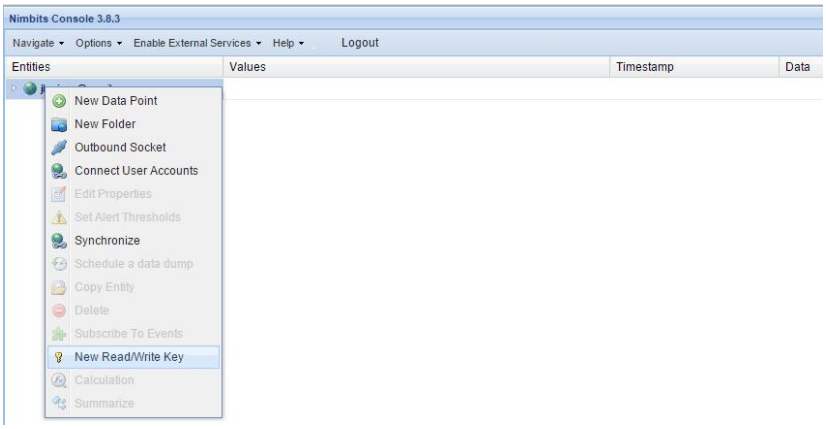
Parte del *software* necesario para la configuración de periféricos y sensores que utiliza la plataforma se puede descargar a través de su repositorio de github.com. Concretamente a través del siguiente enlace https://github.com/bsautner/com.nimbits

Anexo I.II. Detalles de implementación

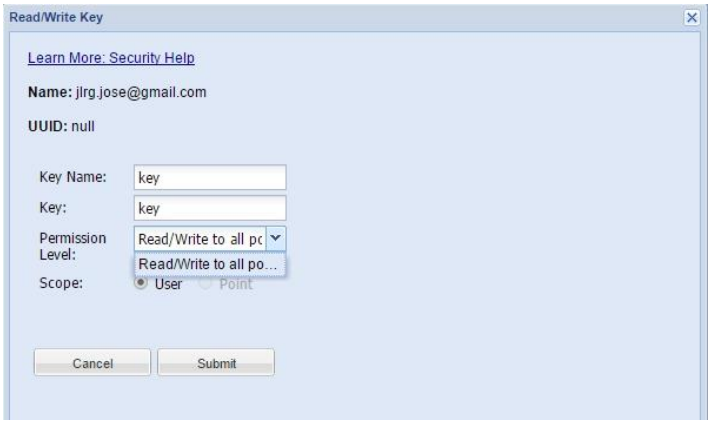
Anexo I.II.I. Preparación de la aplicación web

Tal y como se puede apreciar, no es necesaria la descarga de ningún *software* específico para el funcionamiento de la plataforma. Esto es interesante en términos de escalabilidad y usabilidad en cuanto a la máquina se refiere, pues el rendimiento de la plataforma no dependerá de la máquina en la que se ejecute al no consumir recursos de la misma, sino que dependerá directamente del servidor propio que utilice.

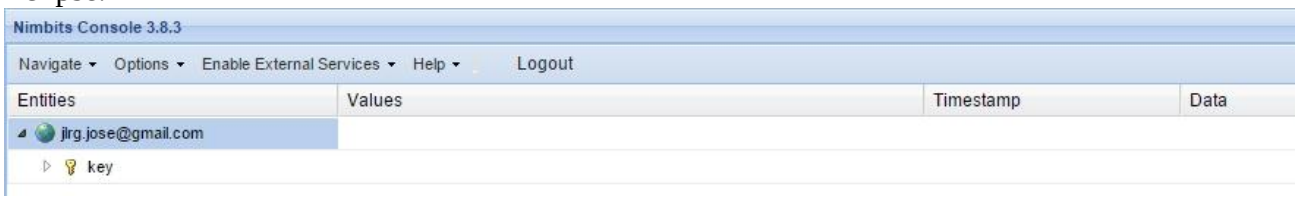
En términos de seguridad, es necesaria la creación de una clave (*Read/Write key*) para el acceso y modificación de los datos en la cuenta Nimbits creada. Esta clave permitirá controlar el acceso a la información restringiéndola a usuarios autorizados. Para llevar a cabo esta tarea se pulsará botón derecho sobre la cuenta de Nimbits y se seleccionará la opción "New Read/Write Key".



Se despliega una ventana con las siguientes características:
1) **Key Name**: nombre de la clave
2) **Key**: contraseña de la clave
3) **Permission Level**: si se desea que la clave permita solamente la lectura, escritura o ambas.
La configuración que se va a tomar para la puesta en marcha del caso de estudio será la siguiente:



Una vez creada la clave de acceso a nuestra cuenta de Nimbits habilitada para la escritura y lectura de datos se obtendrá la estructura básica que se utilizará como punto de partida para el desarrollo del caso de estudio propuesto mediante código Java a través de Eclipse.

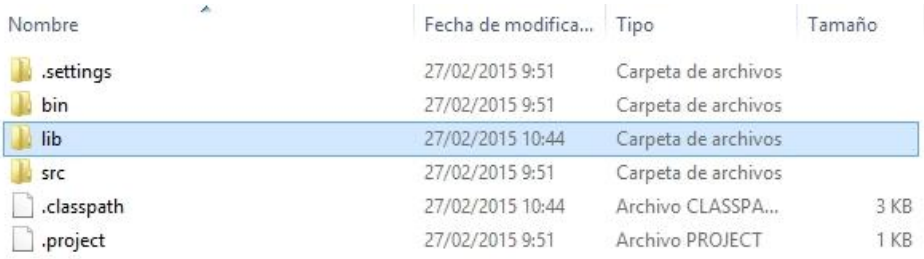


Anexo I.II.II. Implementación en Eclipse

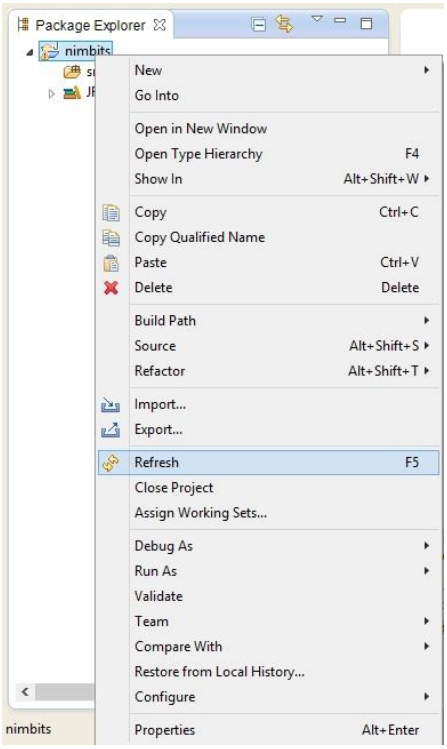
Antes de comenzar con la programación del caso de estudio, se debe preparar Eclipse para su correcto funcionamiento con la plataforma. Para ello se deben añadir las librerías y paquetes necesarios. Las librerías utilizadas, junto con sus enlaces de descarga, son las siguientes:

- apache-commons.jar
- common-lang3.jar
- gson-1.7.2.jar
- guava-10.0.jar
- httpclient-4.2.3.jar
- httpcore-4.2.3.jar
- jetty-util-8.1.7.v20120910.jar
- jetty-websocket-8.1.16.v20140903.jar
- junit-4.12-beta-3.jar
- logback-classic-0.9.jar
- logback-core-0.9.6.jar
- retrofit-1.5.0.jar
- slf4j-api-1.7.7.jar

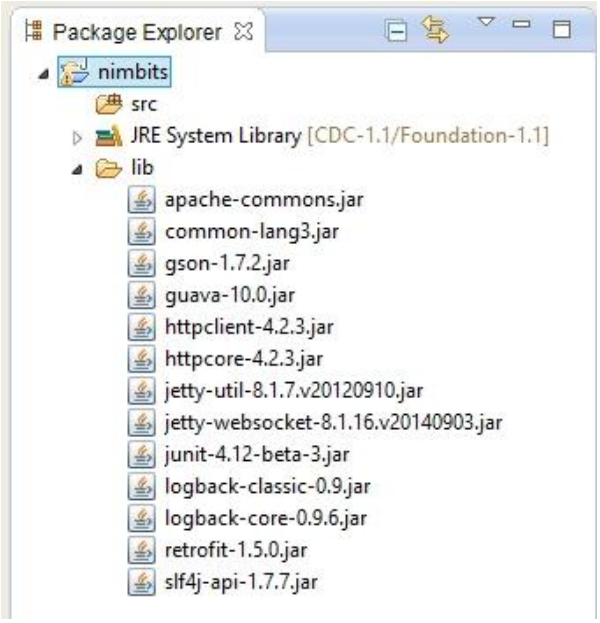
Una vez descargados todos los archivos, se descomprimen y se almacenan en una carpeta llamada "*lib*", que alojaremos en la ruta del proyecto Eclipse afectado en la importación.



Dentro de Eclipse, botón derecho encima del proyecto y *Refresh*.

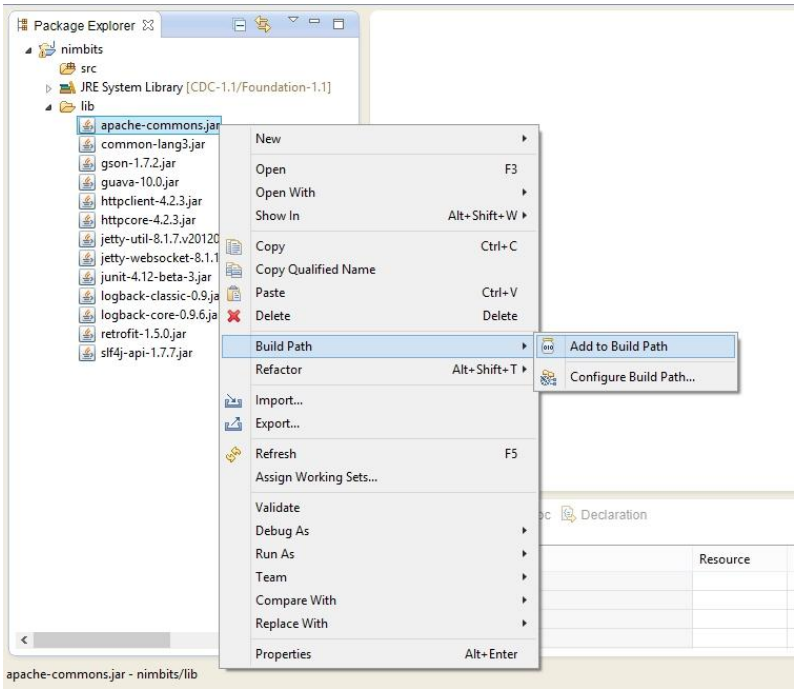


Una vez realizado el refresco, aparecerá la carpeta "*lib*" con los archivos .jar previamente descargados y descomprimidos.

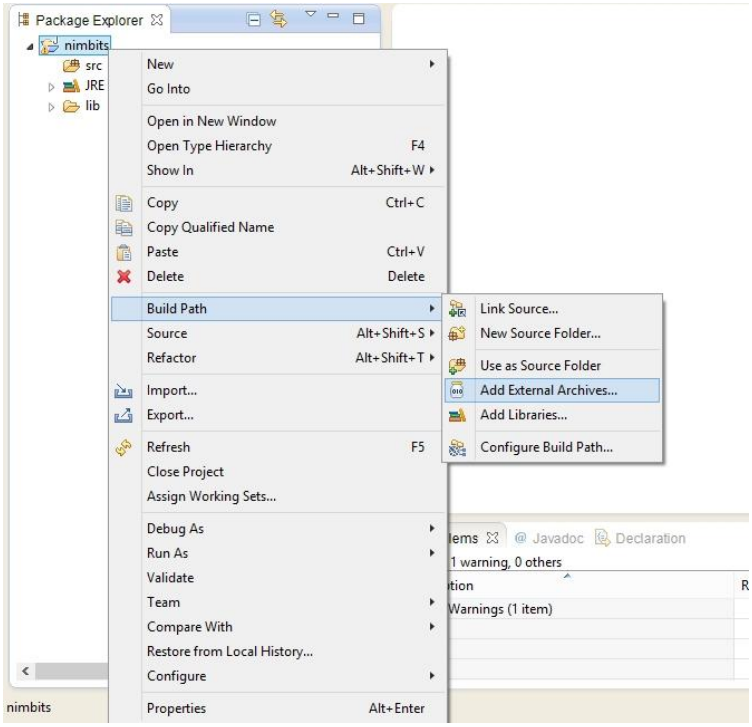


Llegados a este punto tenemos dos opciones para la importación de las librerías.

1. Botón derecho sobre cada archivo .jar > Build Path > Add to Build Path.

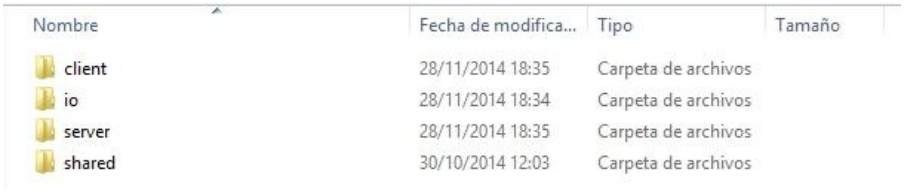


2. Añadir todas las librerías alojadas en la carpeta "lib" a la vez pulsando botón derecho sobre el proyecto > Build Path > Add External Archives... En este punto aparece un explorador de archivos, donde hay que especificar la ruta de la carpeta lib y seleccionar cada una de ellas. Por último se pulsa en 'Aceptar'.



De cualquiera de las maneras indicadas, las librerías serán incorporadas satisfactoriamente apareciendo una nueva pestaña asociada al proyecto denominada "*Referenced Libraries*", donde se encuentra cada librería importada.

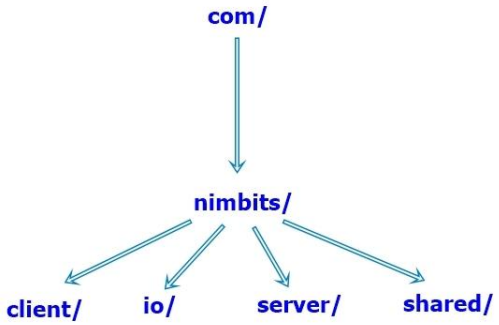
Por otro lado, los paquetes que se necesitan para finalizar la configuración y puesta en marcha de Eclipse para Nimbits son:



Estas cuatro carpetas deben de ir alojadas en la ruta *com/nimbits*. Para conseguirlo, hay que crear manualmente la carpeta com y, en su interior, copiar el contenido de las siguientes rutas, sobrescribiendo, si es necesario:

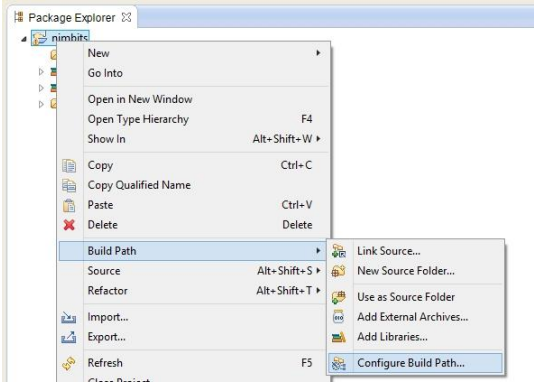
- *nimbits_model\src\main\java\com*
- *nimbits_io\src\main\java\com*

Tras ello, se obtienen las carpetas de la figura anterior, consiguiendo una estructura de directorios fiel al siguiente esquema:

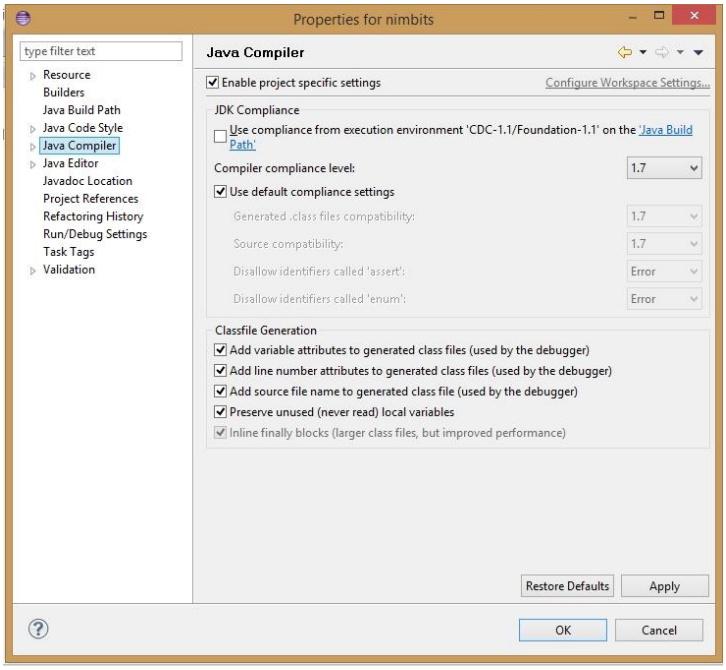


Una vez realizada esta acción se procederá a la importación en el proyecto Eclipse. La importación de paquetes a Eclipse es bastante sencilla. Sin embargo, hay que tener en cuenta algunos aspectos para poder realizar la importación de manera satisfactoria y evitar que Eclipse lance errores.

1. Se debe utilizar el JDK 1.7 para su correcto funcionamiento, para ello se pulsa el botón derecho sobre el proyecto > Build Path > Configure Build Path.

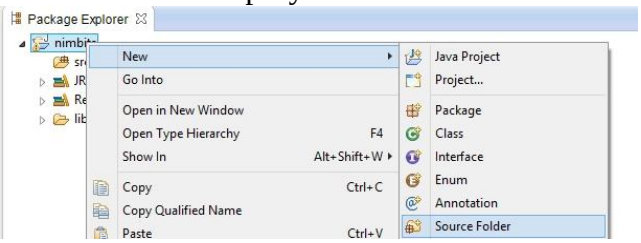


   En la ventana emergente que aparece se debe pulsar en *Java Compiler*, desmarcar la primera opción y seleccionar como nivel de compilador el JDK 1.7.



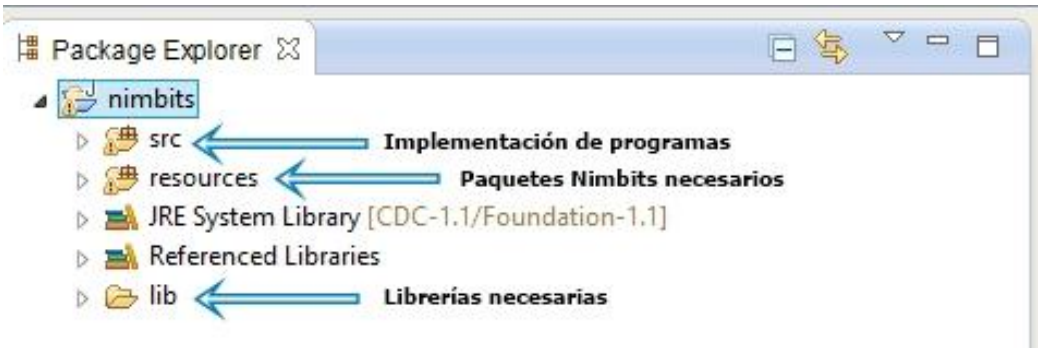
2. Se debe de respetar el encapsulado de las clases en los paquetes especificados en su definición. Comúnmente los diferentes servicios basados en Java que se publican en Internet siguen una estructura definida en cuanto a la encapsulación de clases dentro de diferentes paquetes. Esto ayuda a la depuración de código y al acceso eficiente de las diferentes temáticas que componen la plataforma, entre otras cosas. Concretamente, se ha indicado la jerarquía de clases que Nimbits sigue, prestando especial atención a los paquetes com/nimbits.

   Suponiendo que se respetan las dos premisas anteriores y que se tienen los archivos necesarios para su implementación, la forma de importarlos a java es muy simple. Lo primero se deberá crear una carpeta fuente "*source folder*" que albergará dichos archivos. Para ello se pulsa botón derecho sobre el proyecto > New > Source Folder



   La nueva carpeta recibe el nombre de *"resources"*. Una vez creada, tan solo resta acceder al directorio donde se encuentra nuestro proyecto, entrar en la nueva carpeta *"resources"* y pegar los archivos necesarios y preparados con anterioridad para su implementación. Por último, volver a Eclipse y refrescar el proyecto pulsando F5 o mediante botón derecho > *Refresh*.

Tras esta acción, se puede ver un árbol en Eclipse con el siguiente aspecto, preparado para la interacción con la plataforma Nimbits:



A continuación, se va explicar la implementación en Eclipse de una clase capaz de realizar el envío de datos a la plataforma Nimbits.
El programa asume que se ha realizado *login* en una instancia de un **`SERVER`** Nimbits con dirección **`INSTANCE_URL`**, además de la creación de una clave de carácter global creada con el valor **`key`.**

Cuando se ejecute el programa, se descargará la sesión de usuario y se crearán tantos puntos de datos en el *Cloud* de Nimbits como se especifique en la variable **`nClientes`**. Cada uno de los puntos de datos creado recibe valores aleatorios generados por la clase **`Client`** mediante un periodo determinado.

En primer lugar, se deben especificar los paquetes necesarios para la comunicación con la nube de la plataforma Nimbits. Concretamente se importan los siguientes, añadidos previamente al proyecto:

```
import com.nimbits.client.model.UrlContainer;
import com.nimbits.client.model.common.impl.CommonFactory;
import com.nimbits.client.model.email.EmailAddress;
import com.nimbits.client.model.server.Server;
import com.nimbits.client.model.server.ServerFactory;
import com.nimbits.client.model.user.User;
import com.nimbits.client.model.value.Value;
import com.nimbits.io.helper.HelperFactory;
import com.nimbits.io.helper.PointHelper;
import com.nimbits.io.helper.UserHelper;
import com.nimbits.io.helper.ValueHelper;
import java.util.Random;
```

La clase se denomina **`Client`** e implementa la interfaz **`Runnable`** para habilitar el uso de *threads* en el código con el fin de poder simular el envío de datos a través de diferentes sensores hacia la nube. Así, cada uno de los threads creados simula un sensor diferente. El constructor de la clase tiene la siguiente estructura:

```
public Client(String threadName, String pointName, Server SERVER, EmailAddress
EMAIL_ADDRESS, String ACCESS_KEY){}
```

- **`pointName:`** nombre del punto de datos que se va a alimentar.
- **`threadName:`** *thread* encargado de alimentar el punto de datos anterior.
- **`SERVER:`** recoge la instancia del servidor, entendiéndose por ésta la URL.
- **`EMAIL_ADDRESS`**: dentro del **`SERVER`**, qué cliente solicita el acceso. Se especifica el correo electrónico del cliente.
- **`ACCESS_KEY:`** clave de carácter global creada con valor **`key`** por defecto. Se utiliza para autenticar al **`EMAIL_ADDRESS`** y habilita la lectura/escritura de datos.

Con estos parámetros correctamente configurados, se obtiene acceso a la nube, al crear un nuevo punto de datos en la instancia y un nuevo *thread* asociado a ese punto de datos.

```
//Administracion de puntos de datos
this.pointHelper     =     HelperFactory.getPointHelper(SERVER,     EMAIL_ADDRESS,
ACCESS_KEY);

//Crear puntos de datos
this.pointHelper.createPoint(pointName, "Some Random Description");
th = new Thread(this);
th.setName(threadName);
th.start();
```

Una vez creado y lanzado el *thread* mediante el método *start* asociado al mismo se ejecutará la porción de código asociada al método *run()*. Este método será el encargado de insertar datos aleatorios dentro del punto de datos asociado al *thread* que ejecuta el método. Por ello, las características destacables a tener en cuenta para la correcta configuración del método son tres:

- *Thread* que ejecuta el método
- Punto de datos afectado
- Valor a insertar

Tal y como se ha indicado anteriormente, se crean tantos *threads* como **`nClientes`** se esperan simular. El nombre que se le asigna a cada uno de los *threads* creados es un entero comprendido entre 0 y **`nClientes`**, no permitiéndose dos *threads* con el mismo nombre.

Del mismo modo, con la creación de cada uno de los *threads* se crea un punto de datos asociado al mismo, cuyo nombre se recoge en un *String* con la forma **`"DataPoint"+th.getName()`**. Así pues, el punto de datos *DataPoint0* recibirá datos del *thread 0*, el punto de datos *DataPoint1* recibirá datos del *thread 1*, y así sucesivamente.

Por último, el valor a insertar será fruto de un valor aleatorio creado por la función **`Random()`**. El código descrito es el siguiente:

```
String pName = th.getName();
Random r = new Random();

// Permite la inserción de datos en los puntos de datos.
ValueHelper valueHelper = HelperFactory.getValueHelper(SERVER, EMAIL_ADDRESS,
ACCESS_KEY);

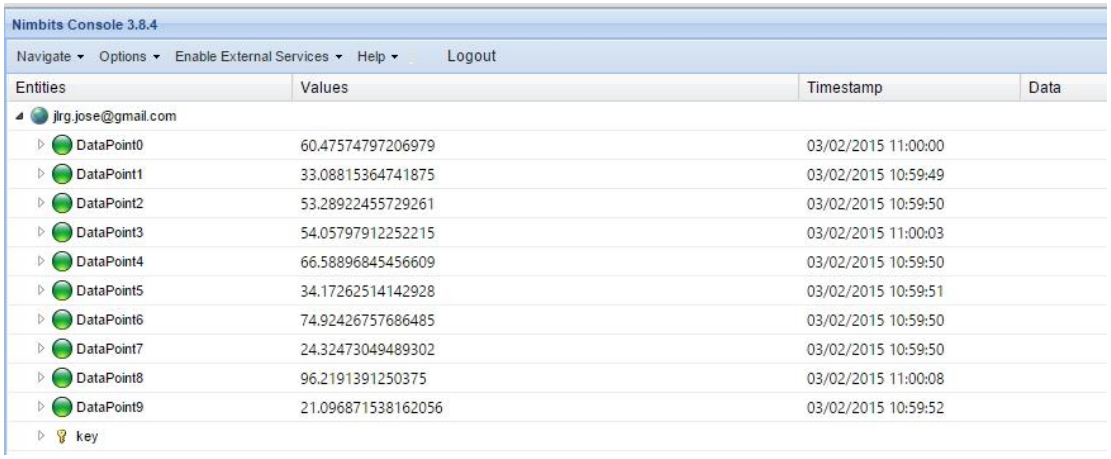
      for (int i = 0; i < 250; i++) {  //250 datos
         try{
             double valor = r.nextDouble() * 100;
             Value value = valueHelper.recordValue("DataPoint"+pName, valor);
             th.sleep(500);
            }
          catch (Exception e){
            System.out.println(e);
            }
      }
```

Por defecto se establece el envío de 250 datos a una frecuencia de 0.5 segundos y con valores comprendidos entre 0 y 100.

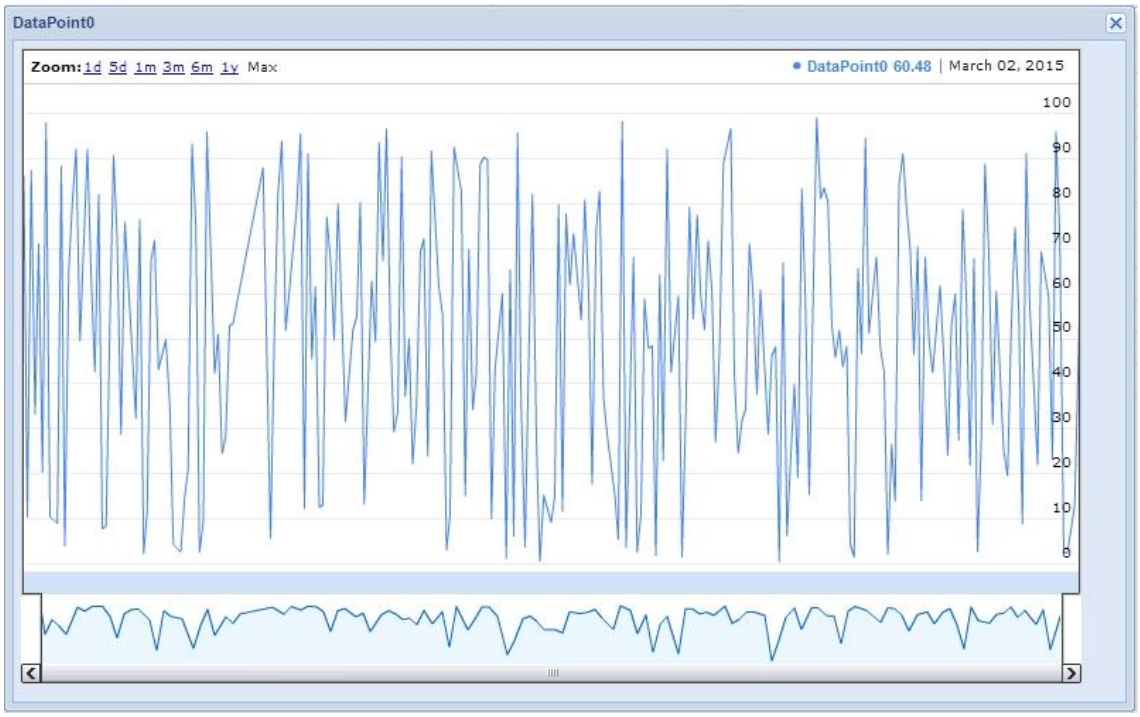
Por último, el método *main* será el lugar donde se establezca la sesión de usuario y se irán creando *threads* a través del constructor de la clase especificado anteriormente. El proceso de establecimiento de la sesión es el siguiente:

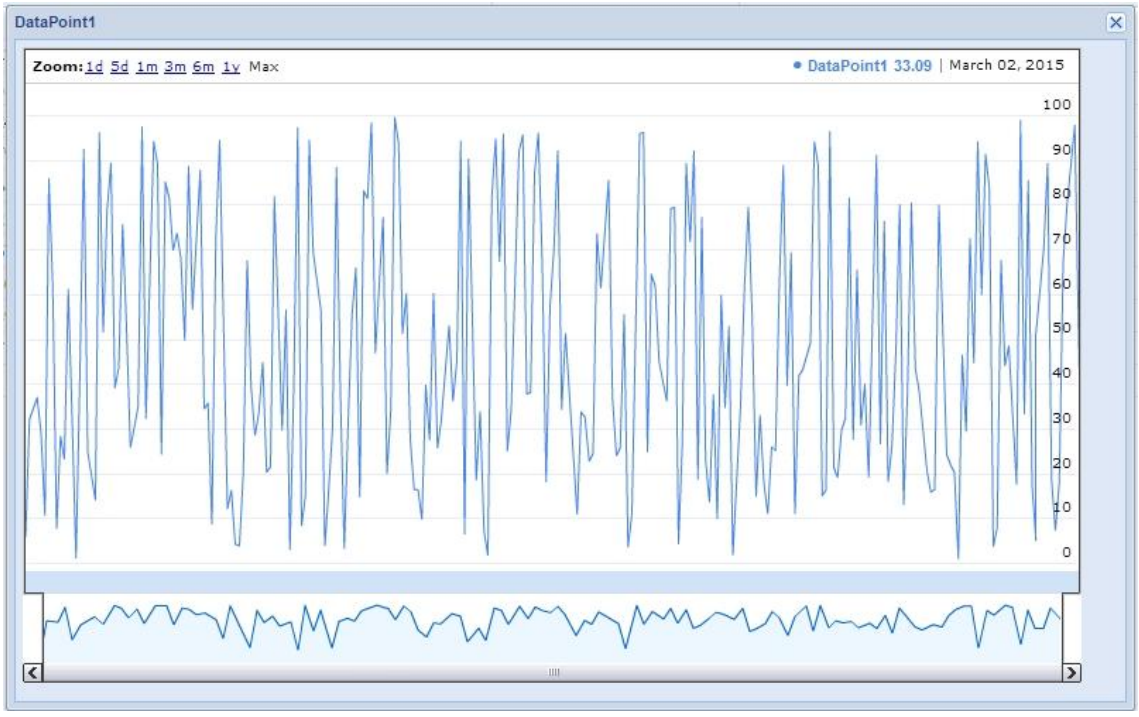
```
//Especifica el email del usuario.
EmailAddress EMAIL_ADDRESS = CommonFactory.createEmailAddress("jlrg.jose@gmail.com");
//Clave de acceso global
String ACCESS_KEY = "key";
// appid.appspot.com para google app engine;
// cloud.nimbits.com para cloud publica;
// localhost:8080/nimbits instancia local;
String URL = "cloud.nimbits.com";
UrlContainer INSTANCE_URL = UrlContainer.getInstance(URL);
Server SERVER = ServerFactory.getInstance(INSTANCE_URL);
UserHelper sessionHelper = HelperFactory.getUserHelper(SERVER, EMAIL_ADDRESS, ACCESS_KEY);
User user = sessionHelper.getSession();
```

El resultado final tras la ejecución del código explicado con *nClientes*=10 es el siguiente:



A modo de ejemplo, se muestran las gráficas resultantes de *DataPoint0* y *DataPoint1:*





Anexo II. Amazon Web Services

Anexo II.I. Detalles de instalación

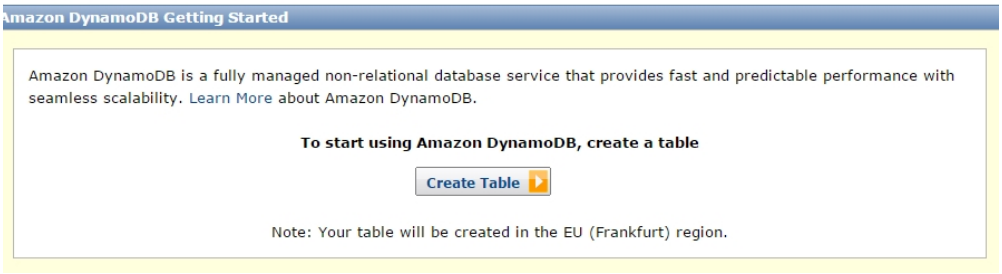
Anexo II.I.I. Creación de una Base de datos en AWS

En este apartado, se realiza una explicación del proceso llevado a cabo para la instalación y configuración de la plataforma así como la comunicación con el entorno de desarrollo Eclipse.

En primer lugar se debe crear una cuenta en Amazon, para ello es necesario acceder a la web http://aws.amazon.com/, hacer click sobre "Iniciar sesión" y crear una nueva cuenta siguiendo cada uno de los pasos que se requieren. Una vez realizado el registro, se habilita el acceso a la consola de la plataforma.

Amazon dispone de una gran cantidad y variedad de servicios tales como servicios de Base de Datos, servicios de networking, administración y seguridad o servicios móviles. Concretamente, el servicio que resulta de interés para el propósito de este trabajo de investigación es el de Base de Datos. Dentro del mismo, Amazon ofrece diferentes posibilidades para la creación y administración de Bases de Datos: RDS, DynamoDB, ElastiCache y Redshift.

Sucesivamente, se describen las pruebas realizadas utilizando el servicio DynamoDB. En primer lugar, hay que hacer click sobre DynamoDB para acceder a la consola de dicha base de datos. En dicha consola, se permite crear, modificar y explorar diferentes tablas creadas en este servicio. Para poder comenzar a almacenar datos es necesario crear una tabla al hacer click sobre *Create Table*.

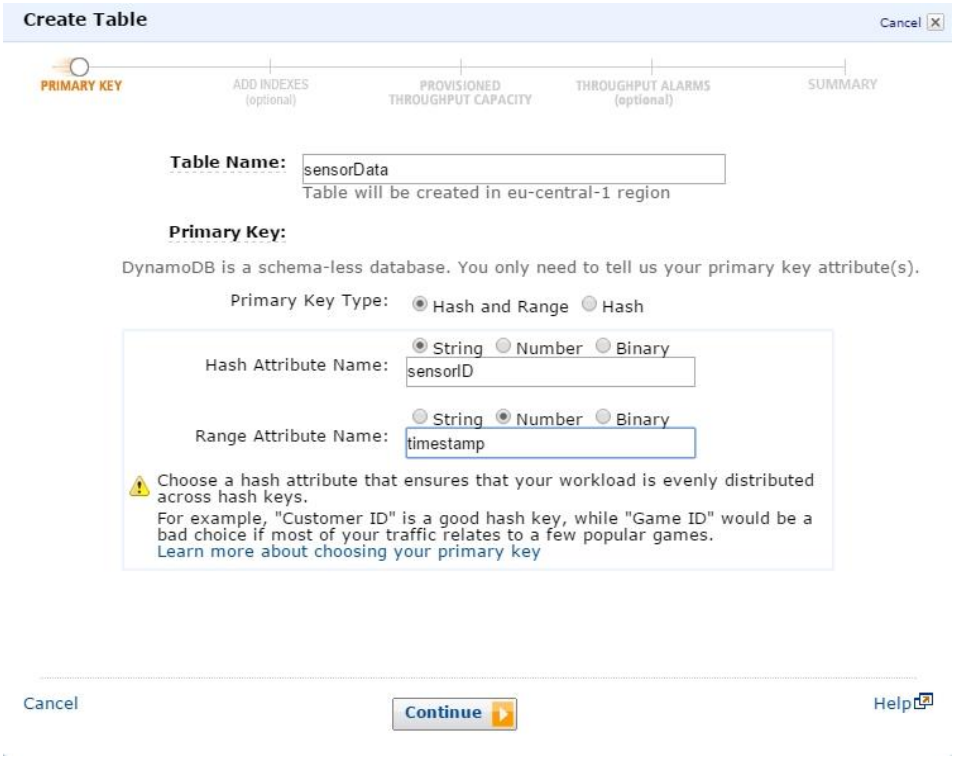


A continuación, aparece una ventana que invita a introducir ciertos parámetros de configuración de la nueva tabla. Cabe mencionar el significado de dos nuevos conceptos que aparecen en este punto, como son *Hash Attribute* y *Range Attribute*. Tal y como indican sus nombres, se trata de atributos de la tabla final que funcionan como identificadores unívocos de los datos.
En el caso de seleccionar dentro de *Primary Key Type* la opción de trabajar solo con *Hash*,

habría un único identificador en la tabla que serviría para registrar datos de los diferentes sensores que se vayan añadiendo, sobrescribiendo el valor de la entrada cada vez que se introdujeran datos. Esta situación sería idónea en el caso de que se quisiera realizar un registro en tiempo real de la situación del sensor.

Por otro lado, si se seleccionara *Hash and Range* habría dos identificadores unívocos de los datos trabajando conjuntamente, de manera que uno de ellos podría ser el identificador del sensor y otro el timestamp de recepción del dato. Como cada vez que se registre un dato en un sensor el timestamp será diferente, no se sobrescribirán los datos, pudiéndose crear y registrar un histórico de datos, con la finalidad de poder evaluar el rendimiento de la plataforma a posteriori.

Por lo tanto, se utiliza el *Hash Attribute* para los identificadores de los sensores y el *Range Attribute* para el timestamp. La configuración final es la siguiente:



Tras hacer click dos veces en *Continue* (se omite la segunda ventana), hay que especificar el *throughput* de la base de datos, es decir, la capacidad de escribir y leer datos por segundo. Para poder realizar una estimación aproximada de los requisitos que se necesitan para llevar a cabo una evaluación equitativa entre plataformas, se puede consultar la siguiente tabla:

Unidades de capacidad requeridas para	**Cómo calcular**
Lecturas	Número de objetos leídos por segundo x 4 KB tamaño objeto
Escrituras	Número de objetos escritos por segundo x 1KB tamaño objeto.

La configuración elegida es: Lecturas = 10 y Escrituras = 5. Tras pulsar nuevamente en *Continue*, aparece una ventana para configurar el envío opcional de alarmas que permiten

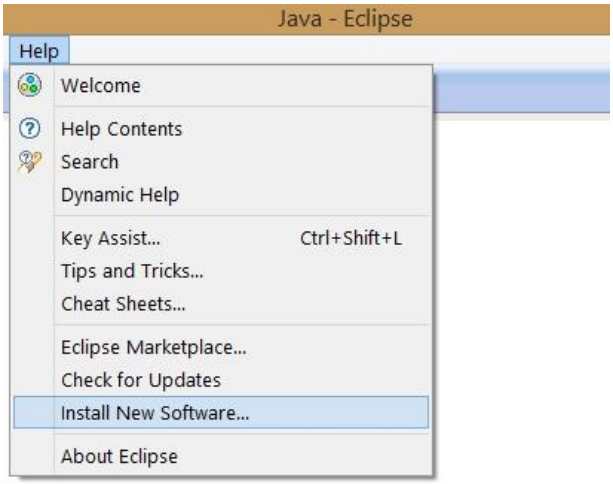
notificar si se excede un umbral de *throughput* configurado. Solamente hay que seleccionar el porcentaje límite y el correo electrónico al que se quiere enviar las notificaciones.

Finalmente, aparece una ventana resumen de la configuración realizada. Si se está conforme, se pulsa en *Create* para crear la tabla. Inicialmente el estado de la tabla es *Creating*.

Posteriormente, dicho estado cambiará a *Active*, momento en el que la tabla estará operativa.

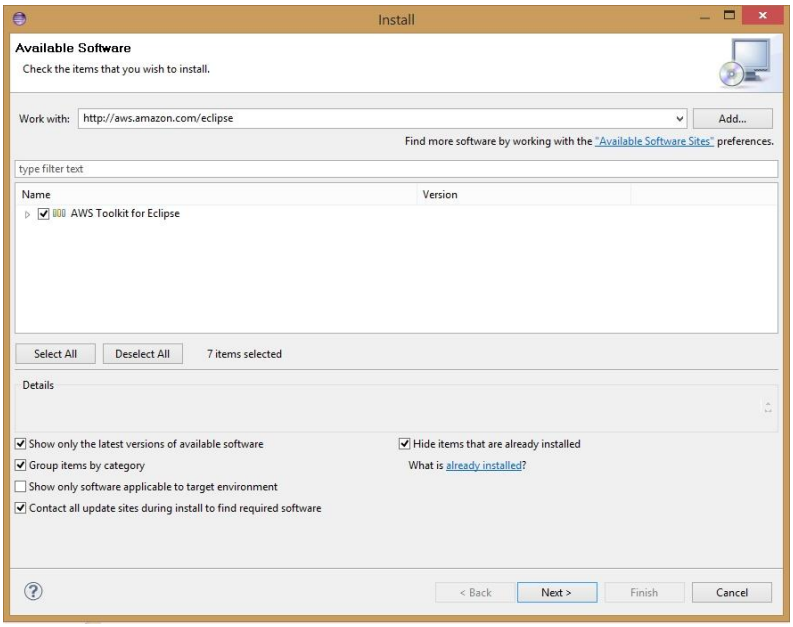
Anexo II.I.II. Instalación de AWS toolkit en Eclipse[3]

Para la instalación del conjunto de herramientas de AWS necesarias para trabajar en Eclipse se debe pulsar en *Help > Install New Software...*



A continuación, se introduce la siguiente dirección en el cuadro de texto *Work with* que aparece en la parte superior del menú. Seleccionar *AWS Toolkit for Eclipse* en la lista y hacer click en *Next* tal y como ilustra la siguiente imagen.

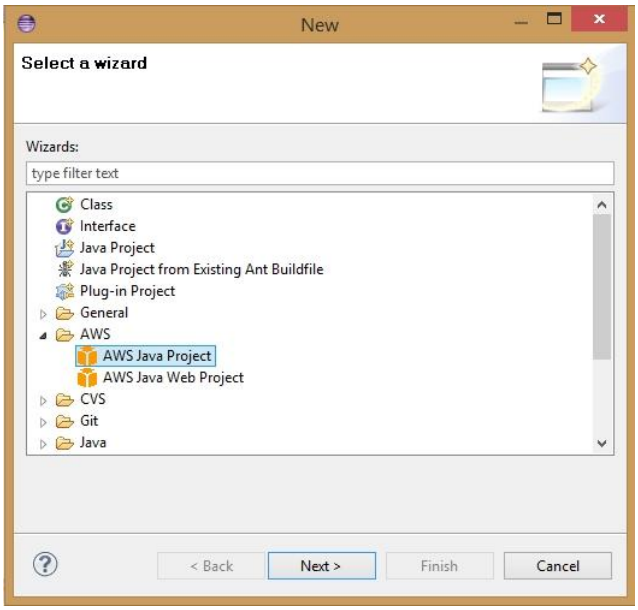
[3]Información extraída de http://aws.amazon.com/es/eclipse/



La parte final de la instalación de los paquetes es intuitiva ya que Eclipse ofrece un proceso guiado.

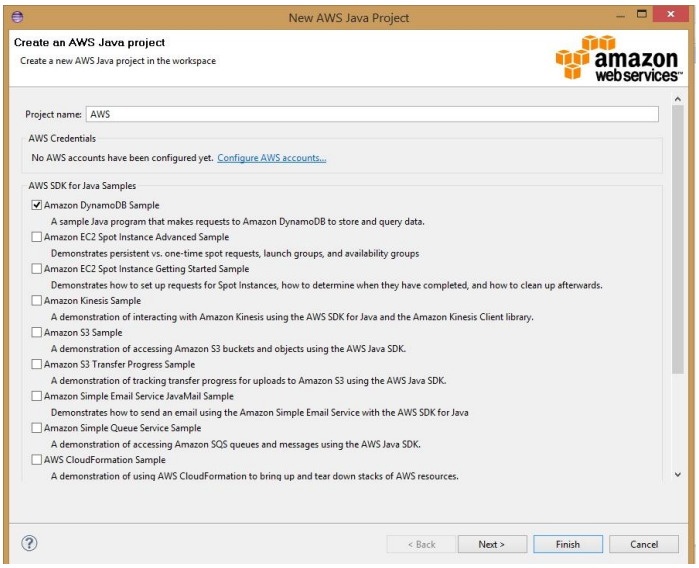
Anexo II.I.III. Creación de un nuevo proyecto AWS en Eclipse[4]

Suponiendo que ya se ha realizado la instalación de los paquetes necesarios en Eclipse, el siguiente paso para la puesta en marcha del servicio de Base de Datos de la plataforma es la creación de un nuevo proyecto. Así, desde el menú de Eclipse, click en *File* > *New* > *Other...* En el asistente seleccionar *AWS Java Project* y click en *Next.*

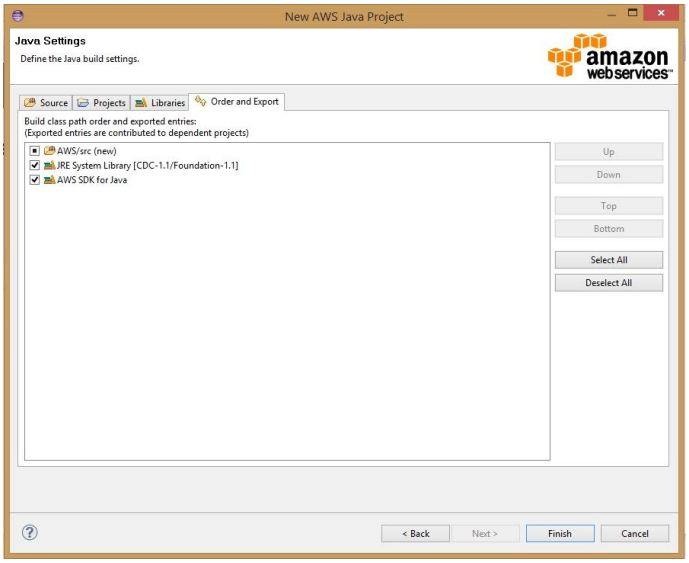


En la siguiente ventana se especifica el nombre del proyecto y opcionalmente, se realiza una selección de diferentes clases de ejemplo a añadir al nuevo proyecto. Seleccionar la clase *Amazon DynamoDB Sample*.

[4]http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/SettingUpTestingSDKJava.html



Hacer click en *Next* y dirigirse a la pestaña *Order and Export* para seleccionar *JRE System Library* y *AWS SDK for Java*. Finalmente pulsar en *Finish*.

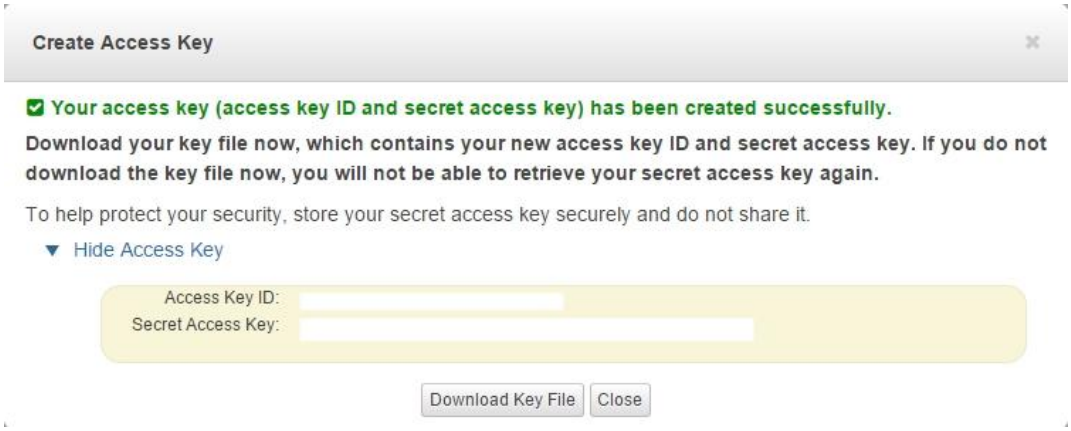


Anexo II.I.IV. Generación de credenciales

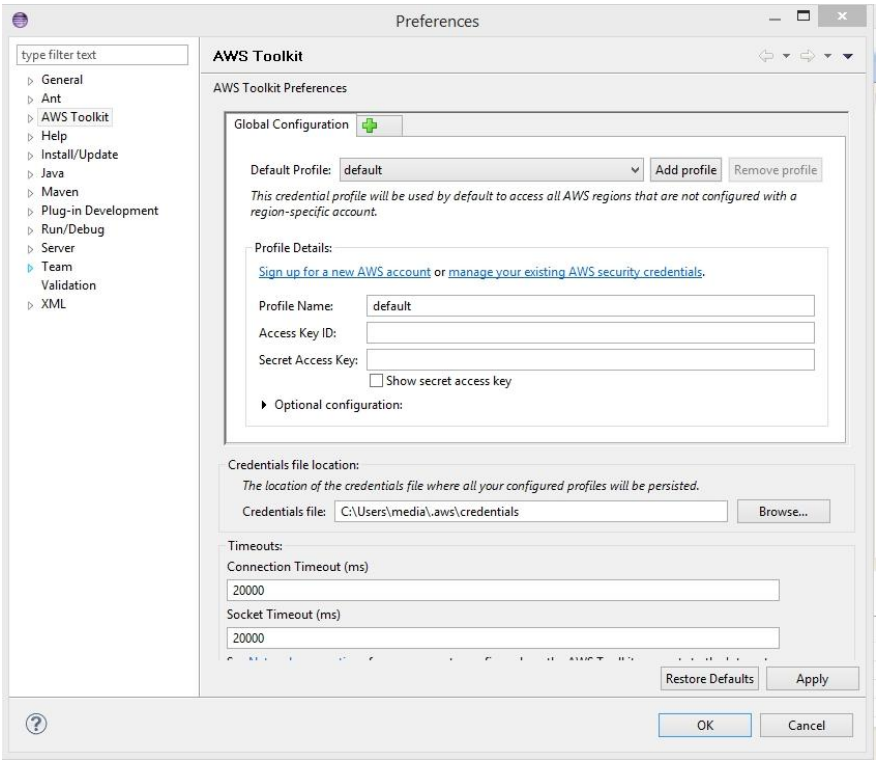
El último paso para la configuración e integración del servicio *DynamoDB* de AWS en Eclipse es la generación de credenciales para poder realizar la conexión a la base de datos. Amazon permite la autenticación mediante la creación de un archivo con las credenciales del usuario de la cuenta. Dichas credenciales son creadas por Amazon previa solicitud siguiendo un algoritmo, pudiendo ser bloqueadas y generadas tantas veces como se necesite siendo diferentes en cada proceso de generación.

Las credenciales se basan en dos parámetros fundamentales, *Access Key ID* y *Secret Access Key*. Para solicitar dicha información accederemos desde nuestra cuenta al apartado *Security Credentials* al hacer click en nuestro nombre dentro de la consola de Amazon.

A continuación, pulsar sobre *Create New Access Key* con lo que aparece una ventana que indica que las claves de acceso han sido creadas de manera satisfactoria. Si se hace click sobre *Show Access Key,* se muestran los parámetros mencionados anteriormente. Los valores de dichos parámetros son secretos y no deben de compartirse aunque se deben copiar para poder configurar Eclipse de manera que a través de ese par de claves se pueda tener acceso a la base de datos *DynamoDB* creada anteriormente.



Por último, resta introducir dicha información en Eclipse. Para ello, se hace click en *Window>Preferences*. A continuación, hacer click en *AWS Toolkit* y aparece una ventana de configuración con los parámetros explicados previamente. En los cuadros de texto adyacentes se debe introducir el par de claves obtenidas del procedimiento descrito en el párrafo anterior, además de dejar el *Profile Name* por configurado por defecto. Finalmente, pulsar en *Apply* y *OK*.



Anexo II.II. Detalles de implementación

Una vez realizado el proceso anterior, se pueden enviar/recibir datos entre Eclipse/AWS, entre otras muchas funcionalidades.

En el desarrollo realizado, la clase que se encarga de enviar datos a la nube se denomina **`AmazonDynamoDBSensorThreads.`** En dicha clase, se registran tantos clientes como se especifiquen y se envían datos bajo un periodo de tiempo determinado. Cada dato genera una marca temporal que se almacena en el *Range Attribute* creado anteriormente en la tabla de *DynamoDB*. Resumiendo, se envía un identificador del sensor, una marca de tiempo y el valor propiamente que genera el programa Eclipse. En primer lugar, los paquetes que se han importado en la clase son las siguientes:

```
import java.util.HashMap;
import java.util.Map;
import com.amazonaws.AmazonClientException;
import com.amazonaws.AmazonServiceException;
import com.amazonaws.auth.AWSCredentials;
import com.amazonaws.auth.profile.ProfileCredentialsProvider;
import com.amazonaws.regions.Region;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClient;
import com.amazonaws.services.dynamodbv2.model.AttributeValue;
import com.amazonaws.services.dynamodbv2.model.PutItemRequest;
import com.amazonaws.services.dynamodbv2.model.PutItemResult;
import java.util.Date;
import java.util.Random;
```

De la misma forma que en el programa implementado en Nimbits, la clase implementa el método *Runnable*, para habilitar el uso de *threads* que simulen el comportamiento de los sensores.

```
public class AmazonDynamoDBSensorThreads implements Runnable {
```

Por su parte, el constructor de la clase recibe el identificador del sensor y crea un *thread* asociado al mismo. Tras su creación lo lanza invocando al método **`run()`**.

```
    public AmazonDynamoDBSensorThreads(String sensorID){

            th = new Thread(this);
            th.setName(sensorID);
            System.out.println("Thread "+ sensorID  + " created!!");
            th.start();

    }
```

El método **`main()`** conecta con la base de datos a través de las credenciales previamente configuradas en Eclipse y ejecuta un bucle de una longitud igual a **`nClientes`** donde, en cada iteración, se crea un cliente distinto que simula un sensor. Entre creación de clientes el hilo se duerme 5 segundos, tal y como se puede observar en la línea **`Thread.Sleep(5000)`**.

```
    public static void main(String[] args) throws Exception {

        init();
        int nClientes = 10;
        String sensorID = null;

       for (int i=0; i<nClientes; i++){
          sensorID = Integer.toString(i);
          AmazonDynamoDBSensorThreads ADDBST = new AmazonDynamoDBSensorThreads(sensorID);
          Thread.sleep(5000);
              }

    }
```

Antes de describir el método **`run()`**, es necesario explicar dos métodos utilizados en el proceso de envío de datos. Estos son los métodos **`init()`** y **`newItem()`**.

- **`init():`** este método accede a las credenciales configuradas y establece la conexión con *DynamoDB*. La única información necesaria para crear un cliente son las credenciales de seguridad, consistentes en los parámetros *AWS Acces Key ID* y *Secret Access Key*. Todas las demás configuraciones, como los puntos de datos finales, son realizadas automáticamente.
  La ruta por defecto que utiliza Eclipse para almacenar las credenciales es *C:\\Users\\media\\.aws\\credentials*, utilizando el perfil por defecto que crea Eclipse.

```
   private static void init() throws Exception {
       AWSCredentials credentials = null;
       try {
           credentials = new ProfileCredentialsProvider("default").getCredentials();
       } catch (Exception e) {
           throw new AmazonClientException(
        "Cannot load the credentials from the credential profiles file. " +
         "Please make sure that your credentials file is at the correct " +
         "location (C:\\Users\\media\\.aws\\credentials), and is in valid format.",
                   e);
       }
       dynamoDB = new AmazonDynamoDBClient(credentials);
       Region eucentral1 = Region.getRegion(Regions.EU_CENTRAL_1);
       dynamoDB.setRegion(eucentral1);
   }
```

general, la cuenta en AWS, basta con acceder a consola. La región aparece al lado del nombre de usuario.

- **`newItem():`** este método tiene como atributos los datos que se desean insertar en la base de datos según la estructura de la tabla creada. Así, en el caso de estudio, el método **`newItem`** recoge el identificador del sensor en una variable de tipo **`String`**, un **`timestamp`** en otra variable del mismo tipo y por último, el valor del sensor en ese momento almacenado en variable tipo **`double`**. El método devuelve una estructura de tipo **`Map`** con un par **`<String, AttributeValue>`**. El primero de estos atributos es el nombre de la columna en la que se insertan los datos, mientras que el segundo atributo es el valor de la misma.

  La estructura de la tabla *sensorData* se compone por una columna llamada *sensorID* que recoge el identificador del sensor, otra columna llamada **`timestamp`** con la marca temporal del dato y una última columna denominada **`Value`** que recoge el valor de ese sensor en ese momento. Así, el código del método para la inserción de datos en la tabla *sensorData* es el siguiente:

```
    private static Map<String, AttributeValue> newItem(String sensorID, String
timestamp, double Value) {

        Map<String, AttributeValue> item = new HashMap<String, AttributeValue>();
        item.put("sensorID", new AttributeValue(sensorID));
        item.put("timestamp", new AttributeValue().withN(timestamp));
        item.put("Value", new AttributeValue().withN(Double.toString(Value)));

        return item;
    }
```

Una vez explicados estos métodos, resta comentar el método **`run()`** de la clase, encargado de iterar de manera concurrente con el fin de insertar datos a la base de datos.

Lo más relevante del método es la utilización de las siguientes variables:

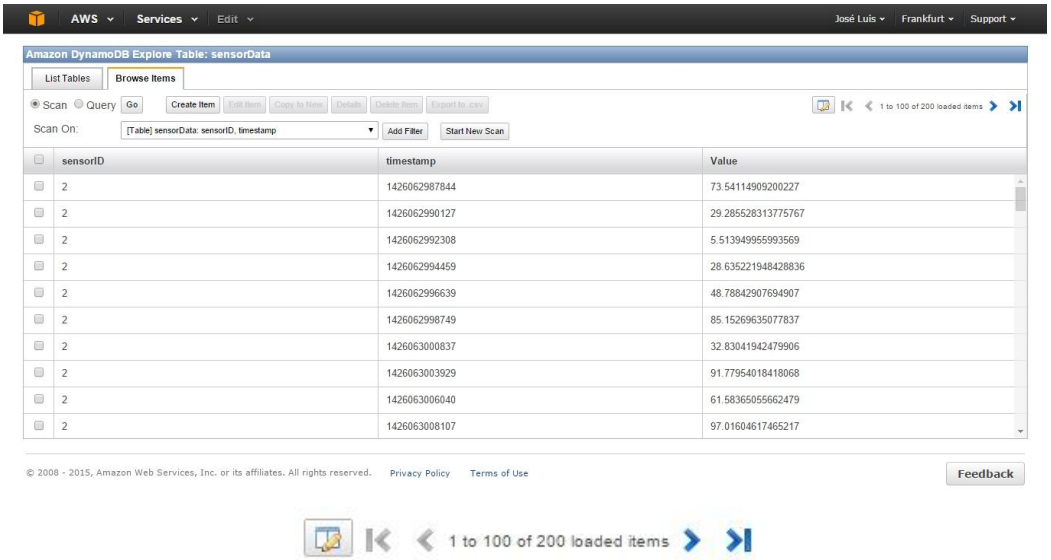
- **`Map<String, AttributeValue> item`**: utiliza el método **`newItem()`** explicado anteriormente. Especifica los valores de cada una de las columnas de la tabla.
- **`PutItemRequest putItemRequest`**: asocia la variable anterior a una tabla determinada dentro de la base de datos. Es decir, enlaza la variable **`item`** a la tabla *sensorData*.
- **`PutItemResult putItemResult`**: inserta el **`putItemRequest`** en *dynamoDB*.

Además de estas variables, se debe de crear una de tipo **`Random`** que crea valores aleatorios y otra de tipo **`String`** que almacena el nombre del *thread* que gobierna dicho sensor. Una vez que se tengan estas variables, sólo queda iterar en un bucle que especifica el número de datos que se quieren insertar en la base de datos.

```
public void run(){
        Map<String, AttributeValue> item;
        PutItemRequest putItemRequest;
        PutItemResult putItemResult;
        Random r = new Random();
        String sName = th.getName();
         String tableName = "sensorData";
         // Add an item
          for(int i=0 ;i<20;i++){
             try{
              Date date = new Date();
              double valor = r.nextDouble() * 100;

              item = newItem(sName, String.valueOf(date.getTime()), valor);
              putItemRequest = new PutItemRequest(tableName, item);
              putItemResult = dynamoDB.putItem(putItemRequest);
              System.out.println("item "+ sName +": "+ valor);
              th.sleep(2000);
             }
             catch (Exception e){
              System.out.println(e);
             }
```

Entre dato y dato existe un periodo de 2 segundos. Tras la ejecución del código, se puede comprobar que los datos se han almacenado correctamente en *DynamoDB*. Para verificar el correcto funcionamiento, acceder a la consola principal de AWS haciendo click en *DynamoDB* y doble click sobre la tabla *sensorData*.



Tal y como se puede observar, hay un total de 200 datos en la tabla. Esto quiere decir que el código implementado funciona correctamente ya que se han simulado 10 sensores (con identificadores del 0 al 9) y cada uno ha enviado 20 datos a la plataforma AWS. Así, el número de datos enviados desde Eclipse coincide con el número de datos almacenados en la *DynamoDB* de la plataforma AWS (200 datos).

Referencias

[1] Thiago Teixeira, Sara Hachem, Valérie Issarny and Nikolaos Georgantas, 2011. "*Service Oriented Middleware for the Internet of Things: A Perspective*".

[2] Project CHOReOS – "*Large Scale Choreographies for the Future Internet*". http://www.choreos.eu/

[3] IERC European Research Cluster on the Internet of Things. http://www.internet-of-things-research.eu/index.html

[4] What exactly is the *"Internet of Things"*? http://postscapes.com/what-exactly-is-the-internet-of-things-infographic

[5] Explicación y definición de Internet de las Cosas. http://www.quees.info/que-es-internet-de-las-cosas.html

[6] Internet of Things Protocols & Standards. http://postscapes.com/internet-of-things-protocols

[7] Internet of Things Hardware Round-up. http://postscapes.com/internet-of-things-hardware

[8] Internet of Things Software. http://postscapes.com/internet-of-things-software-guide

[9] An Internet of Things, some example applications. http://postscapes.com/internet-of-things-examples/

[10] PTC Product & Service advantage. http://es.ptc.com/axeda

[11] Wikipedia: Amazon Web Services. http://es.wikipedia.org/wiki/Amazon_Web_Services

[12] Web oficial de Eclipse. https://eclipse.org/org/

[13] Wikipedia: Eclipse. http://es.wikipedia.org/wiki/Eclipse_(software)

[14] Web oficial PuTTY. http://www.putty.org/

[15] Wikipedia PuTTY. http://es.wikipedia.org/wiki/PuTTY

[16] Wikipedia: WinSCP. http://es.wikipedia.org/wiki/WinSCP

[17] Introducing WinSCP. http://winscp.net/eng/docs/introduction#features

[18] Wikipedia: RStudio. http://en.wikipedia.org/wiki/RStudio

[19] Why RStudio? http://www.rstudio.com/about/

[20] RStudio IDE features. http://www.rstudio.com/products/rstudio/features/

[21] OpenVPN: HowTo. https://openvpn.net/index.php/open-source/documentation/howto.html

[22] Wikipedia: OpenVPN. http://es.wikipedia.org/wiki/OpenVPN

[23] Apache Maven Project. What is Maven? https://maven.apache.org/what-is-maven.html

[24] Wikipedia: Java. http://es.wikipedia.org/wiki/Java_%28lenguaje_de_programaci%C3%B3n%29

[25] What is R? http://www.r-project.org/about.html

[26] Asier Marqués. Conceptos sobre APIs REST. http://asiermarques.com/2013/conceptos-sobre-apis-rest/

[27] Wikipedia: Representational State Transfer. http://es.wikipedia.org/wiki/Representational_State_Transfer

[28] Intoducing JSON. http://json.org/

[29] What is Apache Hadoop? http://hadoop.apache.org/

[30] Web oficial Apache Hive https://cwiki.apache.org/confluence/display/Hive/Home

[31] Web oficial Apache Flume. https://flume.apache.org

[32] "*The Many Faces of Publish/Subscribe*". http://se.inf.ethz.ch/old/people/eugster/papers/manyfaces.pdf

[33] FI-WARE Consortium, 2011. "*FI-WARE High-level Description*".

[34] FI-WARE Data Context Management. https://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/Data/Context_Management_Architecture
[35] Enrique Hernández, Juan Antonio López, Andrés Iborra, 2014. "*Contribución a la Infraestructura FI-WARE. Integración de Dispositivos Empotrados de Bajo Coste en el Ecosistema*".
[36] Orion Context Broker GE. https://fiware-orion.readthedocs.org/en/develop/index.html
[37] Orion Context Broker GE. Arquitectura. https://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/FIWARE.ArchitectureDescription.Data.ContextBroker
[38] Cosmos GE. Análisis Big Data. https://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/FIWARE.ArchitectureDescription.Data.BigData#Big_Data_Analysis_GE
[39] Cosmos GE. Open RESTful API Specification. http://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/BigData_Analysis_Open_RESTful_API_Specification
[40] Conector Cygnus. https://github.com/telefonicaid/fiware-cygnus
[41] Using the command line to manage files on HDFS. http://hortonworks.com/hadoop-tutorial/using-commandline-manage-files-hdfs/
[42] How to enable EPEL Repository on CentOS for Yum Package Management. http://www.thegeekstuff.com/2012/06/enable-epel-repository/
[43] OpenVPN. http://www.redeszone.net/redes/openvpn/
[44] Fedora Project – OpenVPN. https://fedoraproject.org/wiki/Openvpn
[45] Proyecto SICORI - INICIO http://www.dsie.upct.es/proyectos/web_sicori/index.html
[46] J. A. López Riquelme, F. Soto, J. Suardíaz, A. Iborra *"Red de Sensores Inalámbrica para Agricultura de Precisión."*
[47] Proyecto SICORI – PROYECTO SICORI - Red de Sensores http://www.dsie.upct.es/proyectos/web_sicori/proyecto_red_de_sensores.html
[48] C. Albadalejo Pérez, F. Soto, J.A. López Riquelme, A. Iborra "Diseño de una red de sensores inalámbrica para un sistema de observación costero"
[49] P.J. Navarro Lorente, F. Soto Valles, J.M. Molina Martínez "Estación web agroclimática para la adquisición y procesado de datos en tiempo real"