Memoria Trabajo Fin de Grado

Procesamiento en tiempo real de log de actividad en sistemas de Receta Electrónica y uso de técnicas avanzadas de clustering en entornos Hadoop

Autor:
Daniel Cores Costa

Tutores:
Tomás Fernández Pena
Adolfo Sanz Anchelergues

Septiembre 2015

HP – Universidad Santiago de Compostela

D. Tomás Fernández Pena, Profesor do Departamento de Electrónica e Computación da Universidade de Santiago de Compostela, e **D. Adolfo Sanz Anchelergues**, Consultor Senior Business Intelligence de Hewlett Packard Customer Delivery Services,

INFORMAN:

Que a presente memoria, titulada *Procesamiento en tiempo real de log de actividad en sistemas de Receta Electrónica y uso de técnicas avanzadas de clustering en entornos Hadoop*, presentada por **D. Daniel Cores Costa** para superar os créditos correspondentes ao Traballo de Fin de Grao da titulación de Grao en Enxeñaría Informática, realizouse baixo nosa dirección no Departamento de Electrónica e Computación da Universidade de Santiago de Compostela.

E para que así conste aos efectos oportunos, expiden o presente informe en Santiago de Compostela, a 03/09/2015:

O director,	O codirector,	O alumno,
(Tomás Fernández Pena)	(Adolfo Sanz Anchelergues)	(Daniel Cores Costa)

Contenido

1. INTRODUCCIÓN 5

1.1. CONTEXTO 5

1.1.1. HP 5

1.1.2. Análisis de log de actividad en sistemas de receta electrónica con Hadoop 7

1.2. OBJETIVOS DEL PROYECTO 7

1.3. ORGANIZACIÓN DEL DOCUMENTO 8

2. ESPECIFICACIÓN DE REQUISITOS 10

2.1. CASOS DE USO 10

2.1.1. Diagrama de casos de uso 10

2.1.2. Descripción de Actores 10

2.1.3. Descripción de casos de uso 11

2.2. REQUISITOS DE DIFUSIÓN DE LA INFORMACIÓN 13

2.3. REQUISITOS DE CARGA Y ALMACENAMIENTO 14

2.4. REQUISITOS DE ARQUITECTURA Y PLATAFORMA 15

2.5. MATRIZ DE TRAZABILIDAD REQUISITOS-CASOS DE USO 16

2.6. CRITERIOS DE VALIDACIÓN 17

2.6.1. Exclusiones y asunciones del proyecto 17

2.6.2. Entregables del proyecto 18

3. GESTIÓN DEL PROYECTO 19

3.1. METODOLOGÍA DE TRABAJO 19

3.2. PLANIFICACIÓN 20

3.3. ESTIMACIÓN DE COSTES 23

3.4. ANÁLISIS DE RIESGOS 25

3.4.1. Control y seguimiento de riesgos 29

3.5. GESTIÓN DE LA CONFIGURACIÓN 29

3.5.1. Elementos e configuración 29

4. ANÁLISIS TECNOLÓGICO 32

4.1. BIG DATA 32

4.1.1. Las siete V´s del Big Data 32

4.1.2. Características comunes 34

4.1.3. Modelos de computación 35

4.2. HADOOP 37

4.2.1. Hadoop Distributed File System 38

4.2.2. Hadoop MapReduce 42

4.2.3. Distribuciones 47

4.2.4. Ecosistema Hadoop 49

4.3. SISTEMA DE DISPENSACIÓN ELECTRÓNICA 62

4.3.1. Arquitectura 63

4.3.2. Formato de los logs 63

4.4. VIRTUALIZACIÓN 65

4.4.1. Docker 65

5. DISEÑO 69

5.1. ARQUITECTURA GENERAL DEL SISTEMA 69

5.2. ARQUITECTURA DEL SISTEMA DE PROCESAMIENTO DE LOG 69

5.2.1. Arquitectura Lambda 69

5.2.2. Arquitectura alternativa 74

5.2.3. Software de procesamiento 80

5.3. NODOS SISTEMA DE DISPENSACIÓN ELECTRÓNICA 81

5.4. NODOS DEL CLÚSTER HADOOP 83

5.5. SISTEMA DE MONITORIZACIÓN 84

5.6. APLICACIÓN WEB 84

5.6.1. Operaciones por unidad de tiempo 86

5.6.2. Uso del sistema por operación 86

6. IMPLEMENTACIÓN 88

6.1. DESPLIEGUE Y CONFIGURACIÓN DEL CLÚSTER 88

6.1.1. Permitir reanudar el clúster 88

6.1.2. Soporte para realizar imágenes sobre el estado actual del clúster 90

6.1.3. Automatización de la instalación de los servicios en el clúster 90

6.1.4. Instalación y configuración de servicios adicionales 91

6.2. SISTEMA DE DISPENSACIÓN ELECTRÓNICA 95

6.3. SISTEMA DE PROCESAMIENTO 97

6.4. DEFINICIÓN DE INFORMES 99

7. PRUEBAS 104

7.1. DEFINICIÓN DEL ESCENARIO DE PRUEBAS 104

7.1.1. Equipo de pruebas 104

7.1.2. Datos de prueba 104

7.2. CAPA DE RECOLECCIÓN 105

7.3. CAPA DE PROCESAMIENTO 108

7.4. RESULTADOS 108

8. CONCLUSIONES 113

8.1. AMPLIACIONES 114

9. ANEXOS 116

9.1. INSTALACIÓN DOCKER 116

9.1.1. Prerrequisitos 116

9.1.2. Instalación 116

9.2. ANONIMIZAR LOG REALES 118

9.3. DESPLIEGUE DEL SISTEMA 120

10. BIBLIOGRAFÍA 122

1. Introducción

En el presente documento se recoge la memoria del Trabajo Fin de Grado de los estudios de Grado en Ingeniería Informática realizados en la Escuela Técnica Superior de Ingeniería de la Universidad de Santiago de Compostela. Este proyecto nace como una ampliación al Trabajo Fin de Grado presentado en septiembre de 2014 por Andrea Vila Rodríguez titulado “Análisis de logs de actividad en sistemas de Receta Electrónica con Hadoop”.

1.1. Contexto

El desarrollo del proyecto está ligado al Observatorio Tecnológico HP-USC que surge de un convenio de colaboración entre la empresa Hewlett-Packard y la Universidad de Santiago de Compostela. Como ya hemos comentado, este proyecto pretende ampliar las capacidades de un trabajo presentado previamente que también ha sido realizado bajo el amparo del Observatorio Tecnológico.

La colaboración con la empresa ha sido de vital importancia puesto que el presente proyecto está íntimamente ligado con el ámbito funcional de la Receta Electrónica, en el que HP presta servicios a varias Comunidades Autónomas. En concreto, nos centraremos en el sistema de Dispensación Electrónica (DISEL) que está actualmente implantado en el Servicio Galego de Saúde (SERGAS) y con cuyo equipo de desarrollo mantendremos un contacto constante. Los sistemas de Receta Electrónica permiten a los facultativos realizar prescripciones y dispensar fármacos de manera electrónica, eliminando de este modo el uso de la receta en papel y descongestionando la actividad asistencial y el proceso de citación en los centros de salud.

Además, podemos clasificar el sistema desarrollado dentro del mundo del Big Data, un sector en plena expansión con un gran potencial por explotar y en el que HP cuenta con un gran número de profesionales que han servido de apoyo en la realización del proyecto.

1.1.1. HP

Hewlett-Packard es un proveedor de soluciones tecnológicas para consumidores, empresas e instituciones de todo el mundo. Las ofertas de la empresa abarcan las áreas de infraestructura de tecnologías de la información (TI), informática personal y dispositivos de acceso, servicios globales y tratamiento de imágenes e impresión. El objetivo fundamental de HP es crear, diseñar y ofrecer soluciones tecnológicas que impulsen el valor empresarial, creen valor social y mejoren las vidas de los clientes. HP vende sus productos y servicios en más de 170 países y cuenta con aproximadamente 304.000 empleados en todo el mundo. Las oficinas centrales de HP se encuentran en Palo Alto, California.

HP cuenta con tres grupos de negocio principales, cada uno de los cuales se encarga de impulsar el desarrollo y la venta de productos y servicios relacionados con mercados concretos:

- El Grupo de Sistemas Personales (GSP) saca al mercado PCs para uso personal y empresarial, dispositivos de computación móviles y estaciones de trabajo de HP.
- El Grupo de Impresión y Tratamiento de Imágenes (IPG) aplica su saber hacer a la impresión por inyección de tinta, LaserJet y comercial, suministros de impresión, fotografía digital y entretenimiento.

- HP Enterprise Business dispone de una cartera de clase mundial de productos empresariales incluyendo servidores, almacenamiento, productos de red, servicios de externalización y software.

Estas unidades comparten funciones centrales, como investigación y desarrollo (I+D), cuentan con una estructura flexible para aprovechar eficazmente las oportunidades conjuntas, y están vinculadas por comunicaciones y procesos comunes que permiten prestar un servicio perfecto y ofrecer un mensaje uniforme a los clientes.

El desarrollo de este proyecto está ligado al área HP Analytics and Data Management como parte del grupo de negocio HP Enterprise Services.

HP Enterprise Services es una organización mundial de consultoría que ofrece servicios de TI, abarcando productos y tecnología tanto de HP como de terceros. Los consultores de HP Enterprise Services tienen probada experiencia en el diseño de soluciones extremo a extremo adecuadas para cada cliente. El hecho de que inviertan el tiempo necesario en entender los requerimientos del cliente, de sus negocios y de sus mercados, hace que sus esfuerzos tengan resultados eficaces, con mínimos riesgos y en el menor tiempo posible.

HP ha desarrollado un completo set de servicios que abarca todo el ciclo de TI, desde el diseño y la planificación hasta la implementación, mantenimiento y operación. El diverso portfolio de servicios de HP Enterprise Services incluye tanto soluciones de negocio como servicios tecnológicos dentro del nuevo estilo de TI y está organizado entorno a las siguientes siete prácticas que se muestran en la Figura 1.1:

Practice	1. Workload & Cloud Solutions	2. Projects & App Services	3. Enterprise Security	4. Analytics & Data Management	5. Business Process Services	6. Mobility & Workplace Solutions	7. Industry Solutions
What we do	Define the optimal environment for critical app workload and enable network, storage and communication	Manage integrated app practice and leverage center of excellence for project capabilities & transformation roadmaps	Understand gaps in security operations and protect information as it moves in and out of the organization	Deploy analytics to identify business opportunities and efficiently run your business, while streamlining data complexity	Partner with process experts and drive results with economic models based on continuous improvement	Simplify and automate traditional end-user support and enable workforce productivity in a mobile world	Leverage specialized intellectual properties and platform expertise around specific industry segments
Example customer issues	• *Accelerate cloud deployment* • *Optimize data center*	• *Transform legacy applications to cloud* • *Introduce new app service delivery models*	• *Manage security and compliance* • *Mitigate impact of security breaches*	• *Deliver insights with powerful analytics* • *Create big data roadmap*	• *Drive process and industry best-practices* • *Leverage economies of scale via common platforms*	• *Manage SW and HW assets* • *Virtualize end-user applications*	• *Airline (e.g., flight operations, cloud)* • *Communications and Media Solutions (CMS)*

Figura 1.1: HP Enterprise Services

En Galicia cuenta con un Centro de Excelencia de BI en Santiago de Compostela y colabora con la Universidad de Santiago de Compostela a través del Centro Singular de Investigación en Tecnoloxías da Información (CiTIUS) y con acuerdo de tutela de Proyectos Fin de Carrera del Grado y Master en Ingeniería Informática.

HP Analytics and Data Management (A&DM) forma parte de HP Enterprise Services. Los servicios A&DM facilitan el alineamiento entre las personas, los procesos y las tecnologías más adecuadas para crear las analíticas avanzadas y las estrategias de gestión de datos que necesitan los clientes. Los consultores proporcionan el consejo y el soporte necesario para transformar, gestionar y desarrollar iniciativas de Big Data que permitan a los clientes identificar y explotar nuevas

oportunidades que aporten un valor añadido a sus organizaciones. El portfolio específico de A&DM se estructura en las siguientes líneas estratégicas:

- **Consejo:** allanar el camino hacia la innovación con sesiones de trabajo personalizadas sobre Big Data e inteligencia de negocio, evaluaciones de la situación actual y entornos de descubrimiento de oportunidades.
- **Transformación:** transformar la información en conocimiento orientado a la acción mediante el desarrollo y despliegue de analíticas avanzadas.
- **Gestión:** gobernanza y gestión de los datos como un activo estratégico a lo largo de todo el ecosistema de información, mediante métodos, procesos y tecnologías de eficacia probada.

El portfolio de servicio de HP A&DM está además soportado por el conocimiento de tecnologías específicas tanto de HP (HP HAVEn, HP Vertica, HP Autonomy IDOL), como de sus partners y del ecosistema Free Open Source Software.

1.1.2. Análisis de log de actividad en sistemas de receta electrónica con Hadoop

El presente proyecto surge como una ampliación del Trabajo Fin de Grado presentado, por Andrea Vila, en septiembre de 2014. En este trabajo se hacía una primera aproximación al problema y se proponía una solución que, partiendo de grandes cantidades de datos de log, era capaz de generar informes acerca de la actividad del sistema en un periodo de tiempo determinado.

No se contemplaba el análisis en tiempo real a medida que se producen nuevos eventos, sino que partía de un conjunto de ficheros previamente generados. Este sistema permitía la realización de análisis periódicos en momentos en los que, por ejemplo, la carga del sistema era baja. De este modo, se añadía un retardo importante frente a un sistema de procesamiento en tiempo real puesto que una situación anómala no sería detectada hasta que se terminase el ciclo actual y se realizase el siguiente análisis. El aumento en las capacidades de cómputo disponibles hace asumible el desarrollo de un sistema de procesamiento en tiempo real, sobre todo, teniendo en cuenta los beneficios que puede aportar.

La solución propuesta consistía, básicamente, en el acceso a los ficheros de log almacenados en HDFS (Hadoop Distributed File System) de forma estructurada a través de Hive. De este modo, se permitía el acceso a los datos mediante un lenguaje similar a SQL (HiveQL) que se utilizaba para la realización de consultas desde la herramienta de generación de informes Eclipse BIRT. Discutiremos estas tecnologías en la sección de análisis tecnológico.

Esta primera aproximación, así como el gran análisis realizado, tanto del problema en concreto, como de un gran número de tecnologías disponibles suponen un excelente punto de partida para la realización de este proyecto.

1.2. Objetivos del proyecto

El objetivo de este proyecto es proporcionar una herramienta capaz de analizar los logs de actividad de un sistema de Receta Electrónica en tiempo real, en concreto, del sistema de Dispensación Electrónica implantado en el SERGAS, de tal forma que los usuarios puedan tener a su disposición una serie de informes que pueden actualizar utilizando los parámetros que deseen.

Se busca una solución que ofrezca un sistema robusto y con unos tiempos de respuesta reducidos con el fin de poder ofrecer un servicio en tiempo real y de alta disponibilidad prestando especial atención a la posibilidad de que se produzca pérdida de datos. En este sentido, las etapas de análisis tecnológico y de diseño de la arquitectura del sistema cobrarán especial relevancia a la hora de poder confeccionar un sistema de estas características.

Surgen un gran número de cuestiones inherentes al procesamiento en tiempo real que se deben solventar y que en el proyecto anterior no se tenían en cuenta. Por un lado, necesitamos establecer un mecanismo de comunicación robusto con el sistema de Receta Electrónica que no sea sensible al estado del sistema o de la red. Por otro lado, se imponen unas fuertes restricciones sobre los tiempos de procesamiento, en este sentido, cabe destacar que cuando nos referimos a procesamiento en "*tiempo real*", realmente, nos estamos refiriendo a sistemas de procesamiento *NRT ("near real time")*, se trata de sistemas capaces de procesar y analizar los datos a medida que son recibidos ofreciendo unos tiempos de respuesta muy bajos pero que, realmente, tienen un pequeño retardo con respecto a los datos originales.

1.3. Organización del documento

El presente documento se organiza de la siguiente manera:

Capítulo 1: se realiza una primera introducción al problema y se contextualiza el proyecto dentro del marco de un Trabajo Fin de Grado desarrollado en colaboración con la empresa Hewlett Packard.

Capítulo 2: se identifican y describen los casos de uso del sistema y se recoge el catálogo de requisitos completo. Se proporciona con esto una primera definición formal del sistema cuyo desarrollo se detalla a lo largo de todo el documento.

Capítulo 3: se documentan todos los procesos propios de la gestión de proyectos llevados a cabo. Detallaremos la planificación inicial del proyecto y realizaremos tanto un análisis de costes como un análisis de los riesgos inherentes al proyecto y propondremos un mecanismo de gestión de la configuración.

Capítulo 4: en este capítulo se realiza un análisis profundo de todas las tecnologías utilizadas en el desarrollo del sistema final así como las alternativas que se han barajado para ser utilizadas en su lugar.

Capítulo 5: en este capítulo nos centraremos en el diseño del sistema, propondremos tanto un diseño a nivel de arquitectura global del sistema como un diseño mucho más detallado para cada una de sus partes.

Capítulo 6: a partir de los diseños descritos en el capítulo anterior desarrollaremos el producto final. En este capítulo se recogen todas las cuestiones de más bajo nivel relacionadas con la implementación del sistema.

Capítulo 7: someteremos al sistema implementado a una serie de pruebas de validación, mostraremos los resultados que hemos obtenido y comprobaremos si cumplen con los requisitos previamente descritos.

Capítulo 8: expondremos las conclusiones resultado de la realización del presente proyecto y propondremos una serie de ampliaciones al mismo.

Anexos: al final de este documento se recogen una serie de anexos. En el primer anexo se recoge el proceso por el que se ha conseguido anonimizar los datos reales con el fin de ser utilizados en este Trabajo Fin de Grado. En el segundo anexo describiremos el proceso de instalación de Docker, necesario para poder desplegar el clúster que utilizaremos. En el último anexo, se expone un breve resumen con los pasos necesario para instalar y desplegar todos los componentes necesarios.

2. Especificación de requisitos

En este capítulo se recoge el diagrama de casos de uso así como los requisitos que definen el nuevo sistema. Se trata de una sección fundamental puesto que representa los criterios que debe cumplir el desarrollo final para ser validado y aceptado. En este sentido, al final de la sección se recoge un apartado en el que se especifican los criterios de validación del producto que serán utilizados en la fase de pruebas.

2.1. Casos de uso

2.1.1. Diagrama de casos de uso

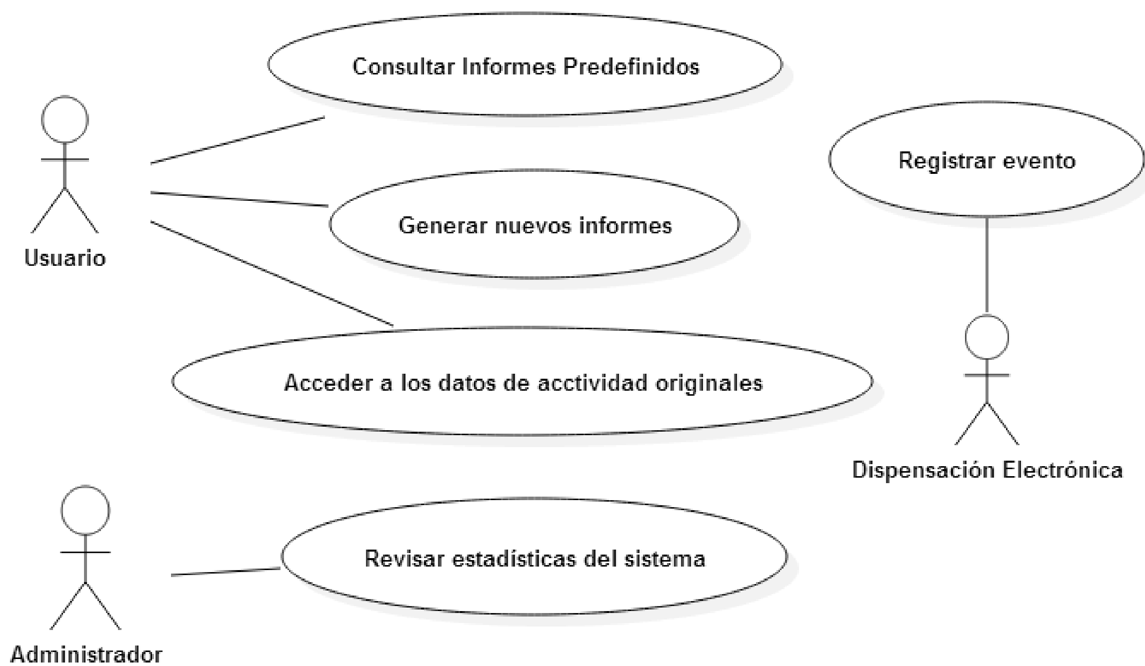


Figura 2.1: Diagrama de casos de uso

2.1.2. Descripción de Actores

Se han identificado los siguientes actores que interactúan de una u otra manera con el sistema:

- **Administrador:** este actor representa el rol encargado de supervisar que el sistema funcione de forma correcta manteniendo unas tasas de procesamiento aceptables en función de las tasas de recepción de nuevos datos.
- **Dispensación Electrónica:** este actor representa al sistema de Dispensación Electrónica que interactuará con nuestro sistema enviando las líneas de log a medida que se producen.
- **Usuario:** este actor representa al usuario final que interactuará con el sistema, fundamentalmente, a través de la aplicación web visualizando los informes disponibles. Además, también tendrá acceso a una herramienta de generación de informes *ad hoc* que no estaban previamente contemplados en la aplicación web. Este rol representa a los miembros del equipo encargado del mantenimiento del sistema de Dispensación Electrónica, con lo que se trata de un tipo de usuario avanzado con grades conocimientos

acerca del funcionamiento de este sistema, de este modo, damos la posibilidad al usuario de acceder directamente a través de la consola a las tablas de la base de datos que contienen la información original a partir de la que se crean los informes.

2.1.3. Descripción de casos de uso

Identificador	CU-01
Nombre	Consultar informes predefinidos
Descripción	El usuario obtiene informes acerca de la actividad del sistema de Receta Electrónica
Actores implicados	Usuario
Precondiciones	-
Postcondiciones	-
Escenario principal	1- El usuario accede a la aplicación web 2- El sistema muestra los informes disponibles 3- El usuario selecciona el informe que desea obtener 4- El sistema muestra un formulario en el que se piden los parámetros del informe 5- El usuario introduce los parámetros que le interesen 6- El sistema genera y muestra el informe
Escenario alternativo	6a- El sistema no puede generar el informe solicitado: 1- Se muestra un mensaje al usuario informando del error

Identificador	CU-02
Nombre	Generar nuevos informes
Descripción	El usuario puede generar nuevos informes, diferentes a los que se habían previsto en el desarrollo de la aplicación web
Actores implicados	Usuario
Precondiciones	En la aplicación Pentaho Report Designer se encuentra correctamente configurada una fuente de datos adecuada para conectarse a nuestra base de datos
Postcondiciones	-
Escenario principal	1- El usuario accede a la aplicación Pentaho Report Designer 2- La aplicación muestra el editor de informes con la conexión a la base de datos previamente configurada

	3- El usuario selecciona la fuente de datos y genera el informe

Identificador	CU-03
Nombre	Acceder a los datos de actividad originales
Descripción	El usuario puede acceder a los datos de log originales almacenados en una base de datos
Actores implicados	Usuario
Precondiciones	La base de datos está creada y disponible
Postcondiciones	-
Escenario principal	1- El usuario accede a la consola del sistema de base de datos 2- El sistema muestra una consola que permite realizar consultas SQL 3- El usuario realiza las consultas que le interesan

Identificador	CU-04
Nombre	Revisar estadísticas del sistema
Descripción	El administrador puede ver el número de eventos que se están registrando por unidad de tiempo y el ratio al que se están procesando para comprobar que todo está correcto.
Actores implicados	Administrador
Precondiciones	-
Postcondiciones	-
Escenario principal	1- El administrador inicia la herramienta que permite visualizar la actividad pasando la ruta en la que se encuentran los ficheros que genera el sistema 2- El sistema muestra información acerca del procesamiento de forma gráfica
Escenario alternativo	2a- No se puede acceder a los ficheros en la ruta indicada: 1- Se muestra un mensaje de error

Identificador	CU-05
Nombre	Registrar evento
Descripción	El sistema de Dispensación Electrónica puede registrar las nuevas líneas de log a medida que

	son generadas para que sean enviadas al clúster y, posteriormente, procesadas.
Actores implicados	Dispensación Electrónica
Precondiciones	Los componentes de envío de datos se encuentran activos en los nodos del sistema de Dispensación Electrónica
Postcondiciones	En caso de que se envíen los datos quedan almacenados en el clúster.
Escenario principal	1- El sistema de Dispensación Electrónica registra una nueva línea de log 2- Nuestro sistema recoge esta línea y la envía al clúster para ser procesada
Escenario alternativo	2a- No se puede establecer la conexión con el clúster pero el buffer no está lleno 1- Se almacenan los datos en un buffer temporal 2- Se reintenta el envío 2b- No se puede establecer la conexión con el clúster y el buffer está lleno 1- Los datos son descartados

2.2. Requisitos de difusión de la información

Se definen los mecanismos de difusión que se van a utilizar para hacer llegar la información a los usuarios así como el conjunto inicial de componentes de difusión de información que estarán disponibles.

Requisitos RDI_001

- **Título:** creación de informes predefinidos como medio de difusión de la información a través de la aplicación web
- **Descripción:** los usuarios tendrán acceso a la información a través de informes previamente definidos que estarán disponibles a través de una aplicación web. Se trata del principal punto de interacción entre el usuario y el sistema.

Requisito RDI_002

- **Título:** posibilidad de crear nuevos informes
- **Descripción:** los usuario podrán crear nuevos informes diferentes a los previamente definidos, en todo caso, estos informes deberán poder utilizar toda la información presente en los datos originales de tal modo que en ningún momento del procesado puede haber pérdida de información.

Requisito RDI_003

- **Título:** acceso a los datos originales sin procesar

- **Descripción:** los usuario podrán acceder a los campos de las líneas de log originales disponibles de forma estructurada en una base de datos a través de consultas SQL.

A continuación se detallan los elementos de difusión de la información que estarán de forma predeterminada en el sistema:

Requisito RDI_004
- **Título:** informe 1: número de operaciones por franja horaria en el último día
- **Descripción:** estará disponible un informe en el que se muestre una gráfica en la que se puede ver el número de operaciones por franja horaria durante el último día con el fin de identificar tendencias que puedan llevar el sistema a una situación de colapso o poder verificar si un fallo ha podido ser debido a un pico de uso inesperado.

Requisito RDI_005
- **Título:** informe 2: número de operaciones por minuto durante la última hora
- **Descripción:** estará disponible un informe en el que se muestre una gráfica en la que se puede ver el número de operaciones por minuto durante la última hora, se trata de un informe que complementa al informe definido en el requisito RDI_004.

Requisito RDI_006
- **Título:** informe 3: uso del sistema desglosado por operación
- **Descripción:** estará disponible un informe en el que se recoge el tiempo de procesamiento y el número de peticiones que se han realizado por cada tipo de operación en un periodo de tiempo determinado, que podrá seleccionar el usuario.

Además de los informes generados acerca de la actividad del sistema de Dispensación Electrónica estará disponible información acerca de las estadísticas de ejecución de nuestro sistema:

Requisito RDI_007
- **Título:** porcentaje de uso de los buffers
- **Descripción:** se proporcionará un informe en el que se recoja la evolución de ocupación de los buffers de la capa de carga con el fin de poder detectar si se han llenado y, por lo tanto, ha habido pérdida de datos.

Requisito RDI_008
- **Título:** número de eventos recibidos
- **Descripción:** se debe mantener el número de eventos (líneas de log) que se están recibiendo con el fin de poder depurar el sistema y ver que coincide con el número de líneas procesadas.

2.3. Requisitos de carga y almacenamiento

A continuación se describen los requisitos que definen la forma en que se cargarán los nuevos datos en el sistema y la forma en la que se deben almacenar:

Requisito RCA_001
- **Título:** origen de la información

- **Descripción:** el origen de la información serán las líneas de log que envía el sistema de Dispensación Electrónica en tiempo real a medida que se generan.

Requisito RCA_002
- **Título:** no debe haber pérdida de datos con periodos de fallos del clúster inferiores a un día
- **Descripción:** cuando los nodos del sistema de Dispensación Electrónica no puedan enviar los datos a los nodos del clúster Hadoop para ser procesados deberán ser almacenados y serán reenviados en el momento en el que se reestablezca la conexión. El periodo máximo en el que se almacenarán los datos para ser reenviados será de un día, a partir de este momento serán descartados. Conseguimos así que los nodos del sistema de Dispensación Electrónica no tengan que disponer de una gran capacidad de almacenamiento disponible para los datos de actividad y evitamos la pérdida de datos en la mayor parte de las ocasiones en las que pueda ocurrir un problema.

Requisitos RCA_003
- **Título:** simular el sistema de Dispensación Electrónica
- **Descripción:** a los efectos de este TFG se debe implementar una aplicación que simule el funcionamiento del sistema de Dispensación Electrónica a partir de los ficheros de log originales. De este modo, esta aplicación leerá los ficheros originales y enviará las líneas del mismo modo que lo haría el sistema original.

Requisito RCA_004
- **Título:** utilizar datos anonimizados
- **Descripción:** los ficheros de log contienen datos sensibles protegidos por la Ley Orgánica de Protección de Datos con lo que, a los efectos de este TFG, los datos deben ser anonimizados.

Requisito RCA_005
- **Título:** mantener todo el histórico
- **Descripción:** se debe mantener todo el histórico de los datos con lo que no se programarán borrados de datos.

2.4. Requisitos de arquitectura y plataforma

Requisitos RAP_001
- **Título:** log4j
- **Descripción:** la aplicación que simula el sistema de Dispensación Electrónica debe utilizar el framework log4j para registrar los nuevos eventos, el envío de estos datos al clúster Hadoop debe definirse a través de los ficheros de configuración de log4j.

Requisito RAP_002
- **Título:** Hadoop
- **Descripción:** el sistema de procesamiento de logs debe desplegarse en un clúster Hadoop.

Requisito RAP_003

- **Título:** Pentaho Report Designer
- **Descripción:** se debe proporcionar un mecanismo de configuración de la aplicación Pentaho Report Designer para que los usuarios puedan generar informes utilizando esta aplicación a partir de los datos presentes en el clúster Hadoop.

Requisito RAP_004
- **Título:** Docker
- **Descripción:** se desplegará el clúster en contenedores Docker en lugar de en máquinas virtuales convencionales con el fin de explorar las ventajas y desventajas que puede aportar este método de virtualización.

Requisito RAP_005
- **Título:** memoria 8GB
- **Descripción**: la plataforma en la que se ejecutará el clúster Hadoop y el software desarrollado en el presente proyecto contará con un máximo de 8GB de memoria RAM con el fin de exprimir al máximo el sistema en entornos con pocos recursos disponibles.

Requisito RAP_006
- **Título:** utilización de software gratuito
- **Descripción:** además de las tecnologías impuestas por los requisitos anteriores, se podrán utilizar otro tipo de soluciones para diferentes propósitos siempre y cuando cuenten con un tipo de licencia gratuita.

2.5. Matriz de trazabilidad requisitos-casos de uso

A continuación mostraremos una matriz de trazabilidad en la que se relacionan los requisitos identificados con los casos de uso que hemos descrito anteriormente, esta relación se especifica con un “1” en la intersección de la columna del caso de uso con la fila del requisito. En esta matriz no hemos representado los requisitos de arquitectura y plataforma puesto que se trata de requisitos no funcionales con lo que no están directamente relacionadas con las funcionalidades que debe soportar el sistema (que vienen dadas por los casos de uso). Esto nos permitirá identificar fácilmente los cambios que supone una modificación en algunas de las funcionalidades.

	CU-01: Consultar informes predefinidos	CU-02: Generar nuevos informes	CU-03: Acceder a los datos de actividad originales	CU-04: Revisar estadísticas del sistema	CU-05: Registrar evento
RDI_001: creación de informes predefinidos como medio de difusión de la información a través de la aplicación web	1				
RDI_002: posibilidad de crear nuevos informes		1	1		1
RDI_003: acceso a los datos originales sin procesar		1	1		1
RDI_004: informe 1: número de operaciones por franja horaria en el último día	1				1
RDI_005: informe 2: número de operaciones por minuto durante la última hora	1				1
RDI_006: informe 3: uso del sistema desglosado por operación	1				1
RDI_007: porcentaje de uso de los buffers				1	
RDI_008: número de eventos recibidos				1	
RCA_001: origen de la información					1
RCA_002: no debe haber pérdida de datos con periodos de fallos del clúster inferiores a un día					1
RCA_003: simular el sistema de Dispensación Electrónica					1
RCA_004: utilizar datos anonimizados					1
RCA_005: mantener todo el histórico		1	1		

2.6. Criterios de validación

A continuación se definen los criterios que se utilizarán para validar el resultado del presente proyecto así como las exclusiones y asunciones que se aplicarán en el desarrollo del mismo por no estar incluidas en la definición de los requisitos. Además, al final de esta sección, se definen los entregables del proyecto.

2.6.1. Exclusiones y asunciones del proyecto

A los efectos de este Trabajo Fin de Grado, no se tendrán en cuenta cuestiones de seguridad y control de acceso al clúster Hadoop al entenderse que se trata de configuraciones propias del entorno de producción en el que se desplegará el sistema y que por tanto están definidas por las políticas de seguridad de la empresa. La configuración de un clúster independiente tiene como objetivo la realización de las pruebas y el análisis acerca de la utilización de contenedores Docker cómo sistema de virtualización en el marco del presente proyecto.

El clúster sobre el que se despliegue el analizador de log en producción debe tener correctamente instalados y configurados todos los servicios necesarios que se describen en la presente memoria. Este despliegue en producción no se incluye en el desarrollo de este proyecto.

El sistema cumplirá con las restricciones recogidas en los requisitos previamente descritos siempre y cuando la carga del sistema sea igual o inferior a la impuesta por los ficheros de log de ejemplo de los que se dispone para su desarrollo.

2.6.2. Entregables del proyecto

Como resultado de la realización del presente proyecto se producirán los siguientes entregables:

- La presente memoria en la que se recoge toda la documentación acerca del desarrollo y gestión del proyecto así como las conclusiones y resultados obtenidos.
- Todos los ficheros de configuración y scripts utilizados para la instalación del entorno así como un manual que forma parte de la memoria.
- El software encargado de realizar el procesamiento en tiempo real, incluido el código fuente.
- Una aplicación web que se conecta al clúster Hadoop y produce una serie de informes, código fuente incluido.

Describiremos en mayor detalle todos los ficheros que conforman los entregables del proyecto en la sección de gestión de la configuración.

3. Gestión del proyecto

En este capítulo se detallan todos los procesos propios de la gestión de proyectos que sirven de marco de trabajo para abordar el desarrollo del proyecto con garantías. En primer lugar, definiremos una metodología de trabajo fundamentada en la propuesta por HP para el desarrollo de proyectos dentro del ámbito del *business intelligence*, esto nos proporcionará una base sólida y testada por un gran número de profesionales del sector. A continuación, propondremos una planificación temporal del trabajo basada en dicho ciclo de vida. Por último, realizaremos una estimación de costes y un análisis de riesgos.

Como resultado, obtendremos una visión global del proyecto, sobre todo, centrándonos en temas de vital importancia como los plazos de finalización y los costes derivados de su desarrollo así como los riesgos asociados que pueden llevar a que el proyecto no finalice con éxito. Se trata de información de vital importancia a la hora de decidir si el proyecto es viable o, por lo contrario, está abocado al fracaso.

3.1. Metodología de trabajo

Como ya hemos comentado, uno de los primeros pasos a la hora de afrontar un nuevo proyecto consiste en definir una metodología de trabajo clara que se llevará a cabo durante todo el desarrollo del mismo. A continuación, describiremos en detalle la que hemos seleccionado para la realización de este proyecto.

Utilizaremos la metodología desarrollada por HP para llevar a cabo proyectos relacionados con el mundo de la inteligencia de negocio [1] denominada **HPGM-BII (HP Global Method for BI Implementation)**. Se trata de una metodología basada en la experiencia de HP a nivel mundial gestionando información del orden de terabytes, ofreciendo un enfoque repetitivo e iterativo, de manera coherente y aplicando las mejores prácticas para resolver problemas complejos. En la figura 3.1 se puede ver un esquema acerca de las fases de esta metodología.

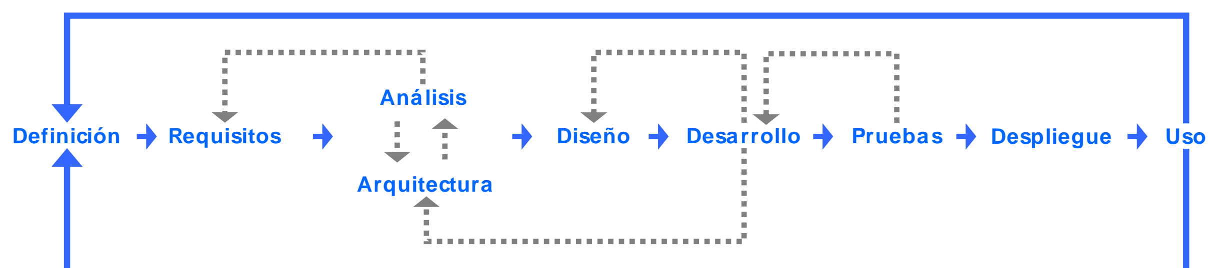


Figura 3.1: Metodología HPGM-BII

Las fases que describe esta metodología se organizan de forma similar a un ciclo de vida en cascada, no obstante, no se realizan de manera secuencial sino que se ejecutan de forma iterativa solapando, en muchos casos, unas fases con otras. En la figura 3.2 se ofrece una visión general de esta metodología:

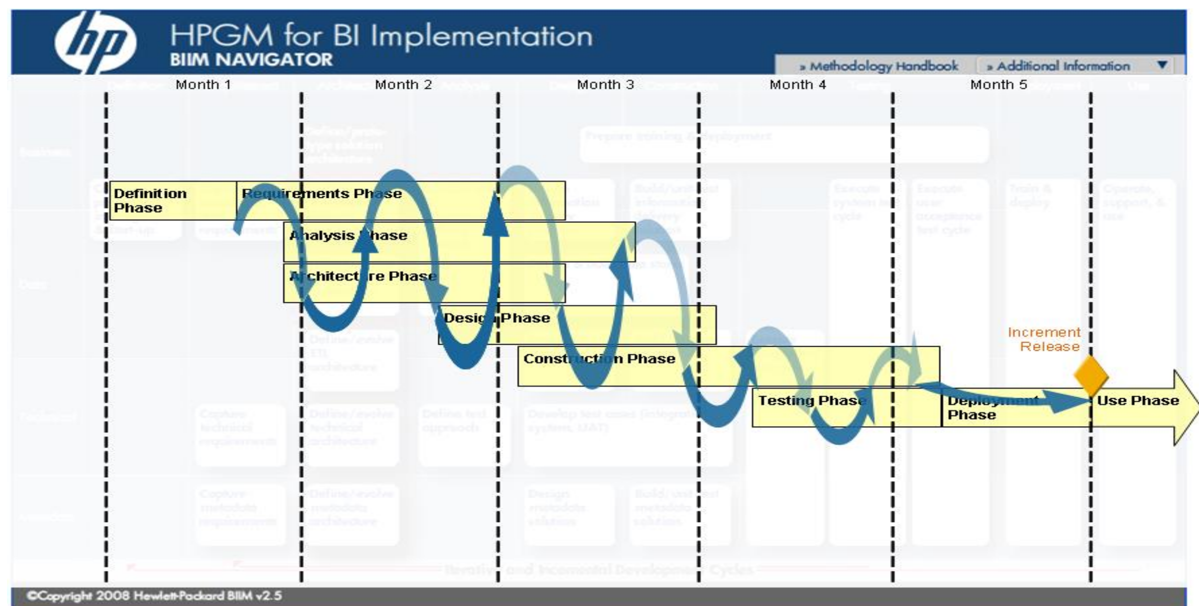


Figura 3.2: HPGM-BII

A continuación se detallan las características fundamentales de este modelo:

- **Claridad en los objetivos funcionales:** para asegurar la entrega de valor a la organización se definen unos estatutos de proyecto que proporcionan una definición clara del proyecto, objetivos, alcance, hipótesis, etc.
- **Desarrollo iterativo:** el desarrollo se realiza de manera iterativa para asegurar la calidad de los requisitos capturados. En cada iteración se valora el estado del proyecto con el cliente y se marcan los siguientes pasos a seguir para cumplir con los requisitos.
- **Implicación del cliente:** el cliente está profundamente involucrado en el desarrollo del proyecto. La creación de prototipos implica a los usuarios desde las fases iniciales del desarrollo.
- **Pruebas de aceptación:** el desarrollo se realiza centrado en pruebas de aceptación de usuario para verificar el alineamiento con las expectativas. De esta forma se garantiza en todo momento el cumplimiento de las funcionalidades requeridas, evitando posteriores errores de implementación.

3.2. Planificación

A continuación describiremos la planificación que hemos realizado para el desarrollo del proyecto. Al tratarse de un Trabajo Fin de Grado se dan una serie de circunstancias muy especiales que no son comunes en el desarrollo de proyectos informáticos. En primer lugar, el grueso del trabajo lo realiza una única persona que hace todas las tareas de gestión y desarrollo en lugar de un equipo de trabajo completo. Por otro lado, los tutores del proyecto juegan un doble papel, por un lado hacen el rol del cliente y por otro lado supervisan todas las tareas que se están llevando a cabo ejerciendo de jefes de proyecto.

Todas estas circunstancias hacen que sea necesario adaptar nuestra metodología a esta situación tan especial. En todo caso, mantendremos las líneas maestras que la definen. Realizaremos **pruebas de aceptación** durante todas las fases del proyecto por parte de los tutores. La **implicación del cliente** en el desarrollo del proyecto está ligado al punto anterior puesto que, como ya comentamos, ambos roles recaen sobre los tutores. Con respecto a **la claridad en los objetivos**, en la

presente memoria se detallan tanto los objetivos como los requisitos que debe cumplir el sistema, también se añade información acerca de las restricciones y las asunciones que se han realizado a la hora de desarrollar el proyecto.

Sin embargo, el solapamiento de tareas que propone HPGM-BII se hace especialmente complicado debido a que la mayor parte de las mismas las realiza una única persona. Además, el proyecto desarrollado en el presente Trabajo Fin de Grado se puede considerar, realmente, como un incremento dentro de un proyecto mayor cuyo primer incremento ha sido el Trabajo Fin de Grado ya comentado que se ha presentado en septiembre de 2014. Esto ha permitido que tanto la definición de los objetivos como la especificación de los requisitos que debería cumplir este nuevo incremento fuese muy clara desde el inicio.

Las principales fases en las que se ha dividido el trabajo son las siguientes:

- **Definición del proyecto:** en esta primera etapa se sientan las bases del proyecto fijando los objetivos generales que se pretenden abordar, en este caso se toma como base el Trabajo Fin de Grado anterior con el fin de identificar las nuevas características que se quieren añadir al sistema. Una de las principales misiones de esta fase es la elaboración del anteproyecto que servirá de guía en la posterior realización del proyecto.
- **Análisis del proyecto:** en esta etapa se realizan todas las tareas de análisis incluyendo la especificación de requisitos, el análisis de costes y el análisis de riesgos. Además, se estudian las tecnologías que se barajarán como alternativas a la hora de implementar el sistema.
- **Diseño:** en base al análisis previo realizaremos una propuesta de diseño tanto a nivel de arquitectura del sistema como a un nivel de detalle más bajo en el que se definen como serán implementados todos los módulos del sistema.
- **Instalación del entorno:** esta fase se dedica a la instalación del entorno y a crear un manual que permita reproducir esta instalación en el futuro.
- **Desarrollo del sistema:** en esta etapa se desarrolla el producto final definido previamente en las etapas de análisis y diseño.
- **Pruebas:** llevaremos a cabo una serie de pruebas que nos permitan validar el desarrollo realizado en la etapa anterior frente a los requisitos previamente definidos. Esta etapa supone la ejecución del sistema mediante la utilización de una serie de datos de prueba que nos permitan ver que realmente se obtienen los resultados esperados.
- **Documentación:** en esta etapa se realiza la memoria del proyecto, cabe destacar que a pesar de que aparezca al final de forma independiente, esta tarea es transversal a todas las demás, con el fin de mantener la lista de tareas siguiente los más simple y legible posible la recogeremos al final como una única tarea. No obstante, hay que tener en cuenta que, efectivamente, gran parte del esfuerzo se realiza al final del proyecto puesto que se integran todas las partes ya redactadas y se prepara para la entrega final.

A continuación se puede ver la planificación completa en la que se muestran todas las tareas que se han llevado a cabo a lo largo del proyecto:

Nombre de tarea	Duración	Comienzo	Fin
Definición del proyecto	**11 días**	**lun 02/02/15**	**lun 16/02/15**
Reunión inicial	1 día	lun 02/02/15	lun 02/02/15
Elaboración anteproyecto	5 días	mar 03/02/15	lun 09/02/15

Validación anteproyecto	5 días	mar 10/02/15	lun 16/02/15
Entregar anteproyecto	0 días	lun 16/02/15	lun 16/02/15
Análisis del proyecto	**24 días**	**mar 17/02/15**	**lun 23/03/15**
Análisis de requisitos	3 días	mar 17/02/15	jue 19/02/15
Análisis de costes	2 días	vie 20/02/15	lun 23/02/15
Análisis de riesgos	2 días	mar 24/02/15	mié 25/02/15
Análisis tecnológico	**17 días**	**mié 25/02/15**	**lun 23/03/15**
Estudio de las tecnologías	12 días	jue 26/02/15	vie 13/03/15
Selección tecnológica	5 días	lun 16/03/15	lun 23/03/15
Análisis finalizado	0 días	mié 25/02/15	mié 25/02/15
Diseño	**14 días**	**mar 24/03/15**	**mar 14/04/15**
Definir arquitectura del sistema	4 días	mar 24/03/15	vie 27/03/15
Diseño de la capa de obtención de datos	2 días	lun 30/03/15	mar 31/03/15
Diseño de la capa de procesamiento	1 día	mié 01/04/15	mié 01/04/15
Diseño de la capa de almacenamiento	1 día	lun 06/04/15	lun 06/04/15
Diseño de la aplicación web	1 día	mar 07/04/15	mar 07/04/15
Validar diseños	5 días	mié 08/04/15	mar 14/04/15
Diseño finalizado	0 días	mar 14/04/15	mar 14/04/15
Instalación del entorno	**10 días**	**mié 15/04/15**	**mar 28/04/15**
Instalación y configuración de Docker	2 días	mié 15/04/15	jue 16/04/15
Instalación y configuración del clúster Hadoop	5 días	vie 17/04/15	jue 23/04/15
Instalación de servicios necesarios en el clúster	3 días	vie 24/04/15	mar 28/04/15
Instalación finalizada	0 días	mar 28/04/15	mar 28/04/15
Desarrollo del sistema	**22 días**	**mié 29/04/15**	**lun 01/06/15**
Anonimización de los logs de prueba	4 días	mié 29/04/15	mar 05/05/15
Desarrollo del sistema de procesamiento	9 días	mié 06/05/15	mar 19/05/15
Desarrollo del sistema de almacenamiento	**3 días**	**mié 20/05/15**	**vie 22/05/15**
Scripts de creación de vistas	2 días	mié 20/05/15	jue 21/05/15
Consultas SQL para la generación de informes	1 día	vie 22/05/15	vie 22/05/15
Aplicación web	**6 días**	**lun 25/05/15**	**lun 01/06/15**
Generación de definición de los informes	2 días	lun 25/05/15	mar 26/05/15
Conexión con la definición de los informes	2 días	mié 27/05/15	jue 28/05/15
Interfaz de usuario	2 días	vie 29/05/15	lun 01/06/15
Desarrollo finalizado	0 días	lun 01/06/15	lun 01/06/15
Pruebas	**13 días**	**mar 02/06/15**	**vie 26/06/15**
Configuración entorno de pruebas	5 días	mar 02/06/15	lun 08/06/15
Programación de ejecución	1 día	mar 09/06/15	mar 09/06/15
Análisis de resultados	2 días	mié 10/06/15	jue 11/06/15
Validación de Resultados	5 días	vie 12/06/15	vie 26/06/15
Pruebas finalizadas	0 días	vie 26/06/15	vie 26/06/15
Documentación	**17 días**	**lun 29/06/15**	**mar 21/07/15**
Redacción de la memoria	12 días	lun 29/06/15	mar 14/07/15
Elaboración de la presentación pública	1 día	mié 15/07/15	mié 15/07/15
Corrección memoria	5 días	mié 15/07/15	mar 21/07/15
Documentación finalizada	0 días	mar 21/07/15	mar 21/07/15

A continuación se puede ver una gráfica en la que se muestra la disponibilidad de los participantes en el proyecto tras la asignación de las tareas anteriores teniendo en cuenta una jornada laboral de 8 horas. Ambos tutores (Tomás Fernández Pena y Adolfo Sanz Anchelergues) mantienen la misma disponibilidad puesto que se les han asignado las mismas tareas de supervisión, en todo caso, esta disponibilidad se mantiene alta con un mínimo al final del desarrollo del proyecto coincidiendo con las tareas de corrección de la memoria y validación del proyecto. En caso del alumno (Daniel Cores Costa) se ha respetado un periodo en el que se ha asignada una menor carga de trabajo (manteniendo una disponibilidad más alta) debido a la realización de las prácticas en empresa. Además, durante el mes de mayo se ha reducido la carga de trabajo coincidiendo con la época de exámenes. De este modo, se pretende buscar la menor interferencia posible con las demás tareas del grado en las que el alumno está involucrado.

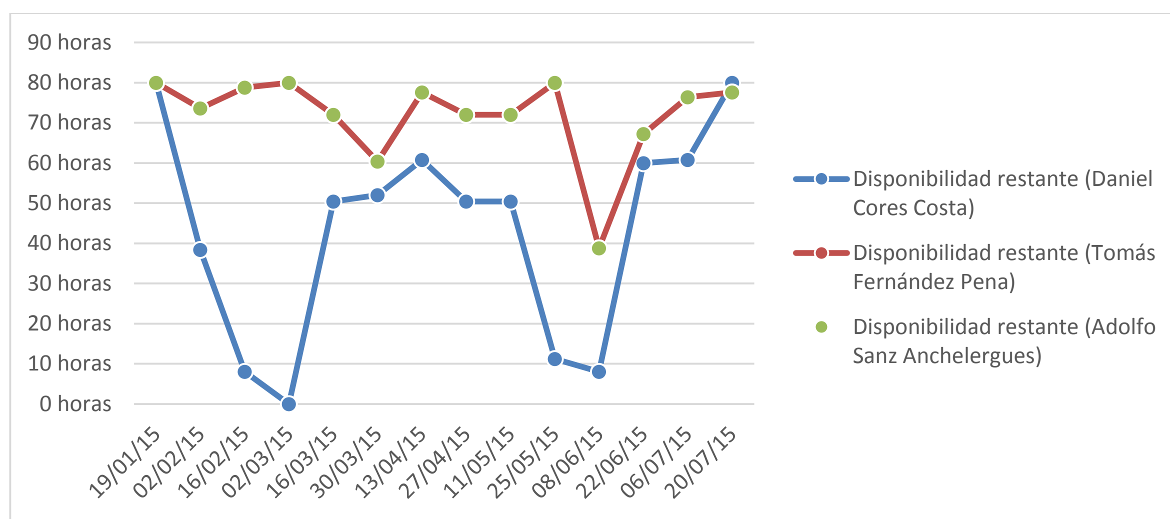


Figura 3.3: Disponibilidad del equipo

3.3. Estimación de costes

A continuación mostraremos la estimación de costes del proyecto desglosada por costes de personal, costes de materiales y otros costes asociados al proyecto.

Costes de personal: la estimación de costes de personal se ha realizado en base a un estudio realizado por la empresa "*Vitae Consultores*" acerca de la situación salarial del sector TIC en Galicia en los años 2014-2015 del que obtendremos los salarios de cada uno de los roles que intervienen en el proyecto. En el caso del alumno se le ha asignado un rol de analista programador con una experiencia de entre 0 y 2 años, si cogemos el salario medio de esta categoría, obtenemos 16.000€/año, lo que supone 8.33€/hora. Por otro lado hemos asignado el rol de Jefe de Proyecto con más de 8 años de experiencia a los tutores. Obtenemos un salario medio de 40.000€/año, lo que supone 20,83€/hora. Cabe destacar que se trata, en ambos casos, de salarios brutos. A mayores debemos añadir los gastos asociados a la cuota de la Seguridad Social que debe abonar la empresa y que supone un 30% del salario de cada empleado, obteniendo unos valores finales de 27€/hora en el caso de los Jefes de proyecto y de 10€/hora en el caso del analista programador.

Personal				
Recurso	**Categoría**	Coste hora	Nº de horas	Importe
Adolfo Sanz Anchelergues	Jefe de Proyecto	27	26	702,00
Tomás Fernández Pena	Jefe de Proyecto	27	26	702,00
Daniel Cores Costa	Analista/Programador	10	428	4.280,00
Total Personal			**480**	**5.684,00**

Figura 3.4: Tabla de costes de personal

A continuación se puede ver la variación del coste asociado al personal, vemos como la fase de análisis supone una gran inversión puesto que no se posee experiencia en las tecnologías manejadas. Además, todo el análisis tecnológico y las propuestas de diseño del sistema componen las partes de mayor valor para la empresa, más allá de las soluciones software concretas desarrolladas.

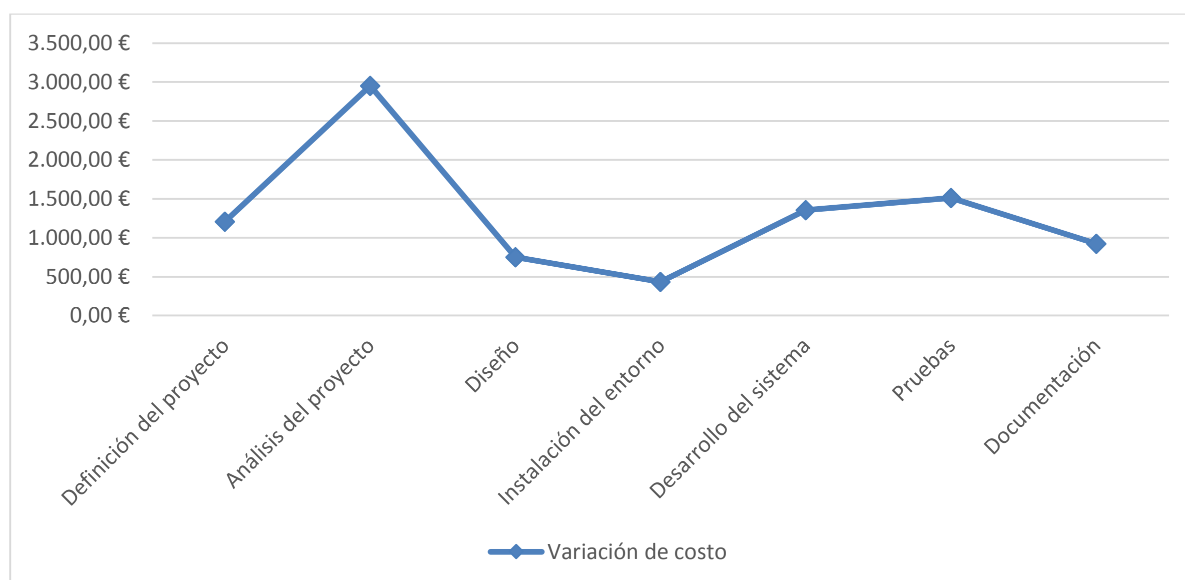


Figura 3.5: Costes asociados a cada etapa

Costes de materiales: en los costes de materiales hemos incluido un ordenador en el que se realizará el desarrollo y en el que se ejecutarán las pruebas, por el requisito **RAP_005** el sistema se debe ejecutar en un equipo con un máximo de 8GB de memoria principal con lo que debemos adquirir un equipo con estas características. Además añadimos los costes asociados a la bibliografía que hemos utilizado en las etapas de análisis y los costes de las instalaciones que hemos utilizado para llevar a cabo el proyecto, incluyendo todos los servicios básicos tales como agua, luz o conexión a internet.

Compras			
Tipo de Operación	**Precio Ud.**	**Unidades**	**Importe**
PC	803,00	1	803,00
Bibliografia	212,14	1	212,14
Instalaciones	150,00	1	150,00
Total Compras			**1.165,14**

Figura 3.6: Tabla de costes por compras

Otros costes asociados al proyecto: a mayores, en esta sección hemos incluido el fondo de reserva para riesgos como parte del plan de contingencia fruto del análisis realizado en la sección *3.4 Análisis de Riesgos*.

Otros Gastos Directos Proyecto			
Tipo de Gasto	**Precio Ud.**	**Unidades**	**Importe**
Fondo de riesgos	1.524,00	1,00	1.524,00
Total Otros Gastos Directos			**1.524,00**

Figura 3.7: Tabla de gastos directos del proyecto

A continuación se puede ver un resumen final de los costes asociados al desarrollo del proyecto así como el coste final tras aplicar un margen de beneficio del 30%.

TOTAL GASTOS	8.373,14
Margen:	30,00%
COSTE FINAL DEL PROYECTO	10885,08

Figura 3.8: Resumen costes

3.4. Análisis de riesgos

En esta sección detallaremos los riesgos que pueden poner en peligro la correcta consecución del proyecto, la gestión de los riesgos consiste en una tarea que se debe llevar a cabo durante todo el ciclo de vida del proyecto realizando las tareas de control pertinentes. Además, una correcta gestión y documentación de los riesgos tiene beneficios más allá del presente proyecto ayudando a la organización en futuros proyectos evitando que se repitan situaciones no deseadas que han ocurrido en el pasado.

Con el fin de poder medir el nivel de exposición al riesgo se ha utilizado la siguiente matriz de probabilidad- impacto:

Nivel de exposición al riesgo							
			Impacto				
			Muy Alto	Alto	Medio	Bajo	Muy Bajo
			0,8	0,4	0,2	0,1	0,05
Probabilidad	Muy Alta	0,9	0,720	0,360	0,180	0,090	0,045
	Alta	0,7	0,560	0,280	0,140	0,070	0,035
	Media	0,5	0,400	0,200	0,100	0,050	0,025
	Baja	0,3	0,240	0,120	0,060	0,030	0,015
	Muy Baja	0,1	0,080	0,040	0,020	0,010	0,005

Figura 3.9: Matriz probabilidad - impacto

Dónde:

- **Probabilidad:** representa la probabilidad de que el riesgo se haga efectivo.
- **Impacto:** representa el efecto de la ocurrencia del riesgo sobre el desarrollo del proyecto, bien sea en esfuerzo, coste, tiempo de desarrollo o cualquier otra métrica utilizada a la hora de medir el éxito del proyecto.
- **Nivel de exposición al riesgo:** consiste en el producto *probabilidad*impacto* con lo que nos ofrece una visión global del nivel de exposición del proyecto a un determinado riesgo. Se trata del parámetro fundamental a la hora de evaluar los diferentes riesgos.

Para cada uno de los riesgos proporcionaremos una descripción así como un análisis del impacto que puede suponer sobre el proyecto, propondremos acciones de prevención que puedan minimizar la exposición al riesgo, bien reduciendo la probabilidad de ocurrencia, bien reduciendo el impacto. Además, definiremos una serie de acciones a llevar a cabo en caso de que el riesgo se materialice.

RSG001: Incumplimiento de la planificación variando las fechas de finalización

- **Descripción:** a la falta de experiencia en la gestión de proyectos informáticos podemos sumar la poca experiencia en las tecnologías que se utilizarán en el desarrollo del sistema, esto puede suponer que las duraciones estimadas en la planificación inicial no se ajusten a las reales retrasando la fecha de terminación del proyecto.
- **Exposición:** ***Alto*** (Probabilidad: *Alta* * Impacto: *Alto*)
- **Coste:** (suponemos un mes de retraso con dedicación completa por parte del analista programador) 2400€ * 0,56 = 672€
- **Acciones de prevención:** la planificación inicial será revisada y validada por los tutores aportando una visión con mayor experiencia.
- **Exposición tras las acciones de prevención:** ***Alto*** (Probabilidad: *Media* * Impacto: *Alto*)
- **Costes tras las acciones de prevención:** 2400€ * 0,2 = 480€
  Los costes asociados a las acciones de prevención no se suman debido a que ya se repercuten en la planificación puesto que los tutores deben realizar acciones de supervisión de todas las etapas.
- **Acciones de corrección:** modificación de la planificación cuando se detecte una desviación.

RSG002: Falta de disponibilidad por parte de los tutores

- **Descripción:** debido a que las tareas de validación que realizan los tutores son imprescindibles para poder confirmar que el desarrollo del proyecto va por el buen camino y a que, en muchos casos, son necesarias para poder continuar con la siguiente fase, son de vital importancia. Al tratarse de dos profesionales pertenecientes a dos organizaciones diferentes y con localizaciones geográficas diferentes pueden surgir conflictos a la hora de llevar a cabo estas tareas de corrección de forma coordinada.
- **Exposición:** ***Medio*** (Probabilidad: *Alta* * Impacto: *Medio*)
- **Coste:** (suponemos un mes de retraso del analista programador a la espera de los resultados de las validaciones)1600€ * 0,14 = 224€
- **Acciones de prevención:** se ha realizado una planificación en la que a las tareas de supervisión se le ha incrementado su tiempo y se ha reducido su dedicación al 20% de tal forma que la dedicación a este proyecto se reduce al mínimo intentando no interferir con otras tareas de otros proyectos.
- **Exposición tras las acciones de prevención:** ***Medio*** (Probabilidad: *Media* * Impacto: *Medio*)
- **Costes tras las acciones de prevención:** 1600€ * 0,14 = 160€
- **Acciones de corrección:** se intentará modificar la planificación en base al progreso en el momento en el que se materialice el riesgo de tal forma que se pueda programar otra tarea que impida el bloqueo en el avance. En caso de que no se pueda, simplemente se modificará la planificación retrasando todas las tareas implicadas.

RSG003: Indisponibilidad por parte del analista programador

- **Descripción:** indisponibilidad del alumno para el desarrollo del proyecto, puesto que la mayor parte del trabajo recae sobre él se trata de una pieza fundamental cuya indisponibilidad puede poner en riesgo el proyecto.
- **Exposición:** ***Medio*** (Probabilidad: *Muy Baja* * Impacto: *Muy Alto*)
- **Coste:** (Suponemos una baja de 1 mes) 1600€ * 0,08 = 128€
- **Acciones de prevención:** la instalación del entorno se replicará e un PC portátil que permita al alumno realizar el trabajo desde una localización diferente.
- **Exposición tras las acciones de prevención:** ***Bajo*** (Probabilidad: *Muy Baja* * Impacto: *Medio*)
- **Costes tras las acciones de prevención:** 1600€ * 0,08 = 32€
- **Acciones de corrección:** modificar la planificación inicial teniendo en cuenta los desfases que pueda haber provocado la materialización del riesgo.

RSG004: Definición errónea de requisitos

- **Descripción:** una mala especificación de los requisitos puede ocasionar que el resultado del proyecto no sea el esperado provocando modificaciones durante el desarrollo del proyecto que ocasionen retrasos con respecto a la planificación inicial.
- **Exposición:** ***Alto*** (Probabilidad: *Media* * Impacto: *Alto*)
- **Coste:** (suponemos mes y medio extra de trabajo para el analista programador) 2400€ * 0,2 = 480€
- **Acciones de prevención:** la toma de requisitos se realizará en base a los resultados obtenidos en el Trabajo Fin de Grado realizado previamente con lo que tenemos desde el principio una idea muy clara de lo que buscamos con este incremento.
- **Exposición tras las acciones de prevención:** ***Bajo*** (Probabilidad: *Muy Baja* * Impacto: *Alto*)
- **Costes tras las acciones de prevención:** 2400€ * 0,04 = 96€
- **Acciones de corrección:** se deberán evaluar los cambios que son necesarios para cumplir los nuevos requisitos y modificar la planificación en consecuencia.

RSG005: Selección tecnológica errónea

- **Descripción:** a causa de la poca experiencia en el manejo de las tecnologías utilizadas se puede dar el caso en que se seleccione una tecnología para resolver un problema determinado en las etapas de análisis y diseño pero que, finalmente, en las fases de implementación y pruebas se compruebe que realmente no es una solución válida.
- **Exposición:** ***Alto*** (Probabilidad: *Alta* * Impacto: *Alto*)
- **Coste:** (suponemos un mes extra de trabajo para el analista programador) 1600€ * 0,28 = 448€
- **Acciones de prevención:** los resultados de las etapas de análisis y diseño son validadas por los tutores, que cuentan con mayor grado de experiencia en algunas de las tecnologías utilizadas.
- **Exposición tras las acciones de prevención:** ***Alto*** (Probabilidad: *Media* * Impacto: *Alto*)
- **Costes tras las acciones de prevención:** 1600€ * 0,2 = 320€

- **Acciones de corrección:** es necesario volver a la etapa de análisis tecnológico con el fin de evaluar alternativas a las tecnologías que no han funcionado y volver a realizar una planificación en consecuencia.

RSG006: Indisponibilidad de datos para realiza las pruebas

- **Descripción:** los datos de log reales no estarán disponibles hasta la fase de pruebas del proyecto. El retraso en la disponibilidad puede obligar a retrasar las pruebas.
- **Exposición:** ***Medio*** (Probabilidad: *Media* * Impacto: *Medio*)
- **Coste:** (suponemos un retraso de dos semanas en el caso del analista programador) 800€ * 0.1 = 80€
- **Acciones de prevención:** por motivos externos al proyecto no tenemos los datos reales disponibles, pero disponemos de un pequeño conjunto de datos de prueba de un entorno de preproducción. Utilizaremos estos datos para realizar una primera validación del sistema y poder corregir algunos errores hasta que estén disponibles los datos reales.
- **Exposición tras las acciones de prevención:** ***Medio*** (Probabilidad: *Media* * Impacto: *Bajo*)
- **Costes tras las acciones de prevención:** 800€ * 0.05 = 40€
- **Acciones de corrección:** será necesario volver a planificar el trabajo restante en consecuencia.

RSG007: Falta de experiencia en el ámbito funcional de la Receta Electrónica

- **Descripción:** el proyecto se realiza en un ámbito muy concreto y desconocido para el alumno como es el de la Receta Electrónica que puede llevar a realizar una mala interpretación de los ficheros de log disponibles.
- **Exposición:** ***Medio*** (Probabilidad: *Alta* * Impacto: *Medio*)
- **Coste:** (suponemos un retraso de dos semanas en el caso del analista programador) 800€ * 0.14 = 112€
- **Acciones de prevención:** las prácticas en empresa del alumno se realizarán simultáneamente con el Trabajo Fin de Grado dentro del equipo de desarrollo del sistema de Dispensación Electrónica de Galicia con lo que se adquirirá cierta experiencia en este sistema.
- **Exposición tras las acciones de prevención:** ***Medio*** (Probabilidad: *Baja* * Impacto: *Medio*)
- **Costes tras las acciones de prevención:** 800€ * 0.06 = 48€
- **Acciones de corrección:** en el momento en el que se detecte que se ha hecho una mala interpretación de los ficheros, bien por parte del alumno, bien por parte de los tutores será necesario replanificar todo el trabajo que se ha realizado en base a uno supuestos incorrectos.

RSG008: Posibles retrasos debido a la carga de trabajo de las demás tareas propias del grado

- **Descripción:** al mismo tiempo que se está realizando el Trabajo Fin de Grado se están cursando asignaturas del grado y se están realizando las prácticas en empresa. Esto puede interferir en algunos momentos impidiendo que se cumplan los plazos marcados en la planificación.
- **Exposición:** ***Alto*** (Probabilidad: *Alta* * Impacto: *Alto*)

- **Coste:** (suponemos un retraso de dos meses en el caso del analista programador) 3200€ * 0.28 = 896€
- **Acciones de prevención:** en la planificación se tiene en cuenta este hecho asignando una dedicación inferior en los periodos en los que se prevé una mayor carga de trabajo en estas tareas.
- **Exposición tras las acciones de prevención:** ***Medio*** (Probabilidad: *Baja* * Impacto: *Alto*)
- **Costes tras las acciones de prevención:** 3200€ * 0.12 = 348€
- **Acciones de corrección:** será necesario realizar una nueva planificación teniendo en cuenta los retrasos acumulados y las futuras interferencias que se puedan reproducir.

3.4.1. Control y seguimiento de riesgos

Durante el desarrollo del proyecto se han materializado los riesgos RSG005 (“Selección tecnológica errónea”) y RSG006 (“Indisponibilidad de datos para realiza las pruebas”) que han supuesto la toma de las medidas de corrección pertinentes.

RSG005: en la sección de diseño de la arquitectura del sistema se detalla más en profundidad este aspecto. En una primera versión se ha seleccionado Hive como tecnología de acceso a datos. Finalmente, en la etapa de desarrollo se han encontrado problemas técnicos que han impedido la implementación del diseño previo, suponiendo un rediseño de una parte de la arquitectura y su posterior implementación. La ocurrencia de este riesgo supuso incrementar la fase de implementación en una semana a tiempo completo y retomar las tareas de diseño y análisis suponiendo otra semana a mayores, lo que conlleva un sobrecoste de 800€.

RSG006: los ficheros de log que se utilizarán para la realización de las pruebas no han estado disponibles en la fecha prevista. Se han aplicado las acciones de corrección previamente descritas utilizando el subconjunto de datos de preproducción para realizar una primera batería de pruebas. La finalización de las pruebas ha tenido que retrasarse una semana en la que se han realizado tareas de documentación con lo que no ha supuesto un retraso en la fecha de finalización.

La fecha de finalización del proyecto se ha retrasado al **4 de agosto de 2015** con un sobrecoste de 800€ a cargado de la reserva para contingencia de riesgos prevista. Esto supone un incremento en los beneficios puesto que de la reserva para riesgos no se ha recurrido a 724€ del total reservado lo que supone un 6,6% de los costes totales del proyecto.

3.5. Gestión de la configuración

Al igual que ocurría en el caso de la planificación, el hecho de que se trate de un Trabajo Fin de Grado determina en gran medida este proceso. La realización del proyecto por una única persona permite aligerar todo el proceso de gestión de la configuración puesto que no tendremos varios miembros del equipo accediendo de forma concurrente a los elementos de configuración.

3.5.1. Elementos e configuración

El primer paso consiste en identificar los elementos de configuración en nuestro proyecto:

- Código Fuente
  - Proyecto con el código del sistema de procesamiento de log
  - Proyecto con la aplicación Web
  - Proyecto con el sistema de simulación del sistema de Dispensación Electrónica

    - Proyecto con la aplicación de monitorización de los agentes Flume
    - Scripts de anonimizado de los datos originales
- Memoria del proyecto
- Scripts *bash* para desplegar el clúster
- Imágenes Docker
    - Imagen base para los contenedores del clúster
    - Imagen del sistema de simulación del sistema de Dispensación Electrónica
    - Imagen para el contenedor en el que se desplegará la aplicación web

A continuación, identificaremos el árbol de directorios de nuestro repositorio principal en el que se mantendrán todos los elementos de configuración identificados anteriormente y que coincidirá con la estructura de fichero de los entregables adjuntos a este proyecto. A lo largo de la presente memoria, sobre todo, en la sección de implementación se describirán estos elementos en un mayor nivel de detalle.

Elemento	Descripción
TFG_030915	Carpeta raíz en la que se incluyen todos los ficheros
MEM_TFG_030915.pdf	Documento que recoge la presente memoria
anonimizar.py	Script Python utilizado para anonimizar los datos
imgDISEL	Carpeta que contiene los ficheros necesarios para la generación de la imagen que simula el funcionamiento del sistema de Dispensación Electrónica
Dockerfile	Fichero de definición de la imagen Docker
flume.sh	Fichero que lanza los agentes Flume
flumeDisel.conf	Fichero de configuración del agente Flume
disel.jar	Aplicación Java que simula el funcionamiento del sistema de Dispensación Electrónica
apache-flume-1.6.0-bin	Distribución de Flume que utilizaremos
imgReporting	Carpeta que contiene los ficheros necesarios para la generación de la imagen que contiene la aplicación web de generación de informes
Dockerfile	Fichero de definición de la imagen Docker
Reporting.war	Aplicación web
hdp-docker	Carpeta en la que se incluyen todos los ficheros relacionados con el clúster Hadoop
imagen	Carpeta que contiene los ficheros necesarios para la generación de la imagen del clúster Hadoop
Dockerfile	Fichero de definición de la imagen Docker
hdp-cluster.sh	Script que contiene el código necesario para automatizar el despliegue del clúster
compartida	Carpeta cuyo contenido se debe copiar a /usr/hdp y que será utilizada por os contenedores como almacenamiento compartido
resources	Carpeta con ficheros de configuración

config.properties	Fichero de configuración del sistema de procesamiento de log
ambari-shell.sh	Script que ejecuta el los comandos recogidos en *ambari-script* en la consola de Ambari para automatizar la instalación del clúster
ambari-script	Script con comandos Ambari para la instalación de todos los componentes
hdp-blueprint	Blueprint con la definición del clúster
join-cluster.sh	Script que reconfigura el agente Serf para volver a unir el agente al clúster
spark-install.sh	Script de instalación de Spark en el clúster
analizador_1.0.jar	Aplicación de procesamiento de log en tiempo real
codigo	Carpeta en la que se encuentra el código fuente
procesamientoLog	Carpeta que contiene el código del sistema de procesamiento de log
aplicaciónWeb	Carpeta que contiene el código de la aplicación web
simulacionDISEL	Carpeta que contiene el código que simula el funcionamiento del sistema de Dispensación Electrónica
monitoringFlume	Carpeta que contiene el código de la aplicación que utilizaremos para monitorizar los agentes Flume.

4. Análisis tecnológico

En el siguiente apartado realizaremos una contextualización tecnológica del proyecto así como un estudio de las diversas tecnologías que se han barajado a la hora de implementar el sistema final. Este estudio servirá de punto de partida para la realización, posteriormente, de una selección tecnológica teniendo en cuenta, por un lado, las características de cada una de estas tecnologías y, por otro lado, las necesidades específicas de nuestro proyecto.

Las tecnologías seleccionadas así como la forma en que se interconectan y el rol que jugarán en la implementación final se describirán en el apartado de diseño, concretamente, en la sección de "*5.2. Arquitectura del sistema de procesamiento de log*".

4.1. Big Data

El concepto de **Big Data** [2] se refiere a toda aquella información que, debido a su gran tamaño y complejidad, no puede ser procesada o analizada utilizando procesos y herramientas tradicionales. La cantidad de datos generados, así como su naturaleza, ha cambiado radicalmente desde la época de las primeras mainframes hasta la actualidad. Este gran cambio se puede atribuir, en gran medida, al auge de la computación ubicua así como a la aparición de nuevos paradigmas como la web 2.0, las redes sociales, la computación en la nube o la popularización de las arquitecturas basadas en SaS (Software-as-a-Service). Cabe destacar, que el concepto de Big Data no se refiere exclusivamente a grandes cantidades de información sino que también se centra en la complejidad de los conjuntos de datos manejados, normalmente con estructuras muy heterogéneas (incluso textos en lenguaje natural carentes de estructura) que dificultan su procesamiento.

La información que poseen las organizaciones es de vital importancia como herramienta principal a la hora de realizar la toma de decisiones y establecer nuevas líneas de negocio, el reto radica en recolectar, almacenar y procesar las grandes cantidades de información que se están generando de forma continua para poder aportar valor añadido a la organización. Todo ello, debe realizarse utilizando sistemas escalables y con un coste asumible para estas organizaciones, no es factible aumentar drásticamente los recursos hardware en cortos periodos de tiempo, en los que el volumen y complejidad de los datos manejados es muy probable que si se incremente [3].

4.1.1. Las siete V´s del Big Data

Como ya hemos comentado, Big Data no se refiere exclusivamente a grandes cantidades de datos, sino que entran en juego otros conceptos como son la **variedad** y la **velocidad** que forman junto con el **volumen** lo que se dio a conocer como las *3 V´s del Big Data* [4]. Posteriormente, se han añadido otros dos conceptos fundamentales: **valor** y **veracidad**, dando lugar a las *5 V´s*.

- **Volumen:** se refiere a las grandes cantidades de datos que las organizaciones deben manejar. Cabe destacar que el ritmo al que crece la generación de nuevos datos se ha disparado en los últimos años, de tal forma, que se estima que el 90% del total de los datos existentes en el mundo han sido creados en los últimos 2 años.

  Según los datos publicados por YouTube cada minuto se suben unas 300 horas de vídeo a este sitio web. Por otro lado, según www.internetlivestats.com el tráfico total de Internet asciende a 29.000 GB por segundo. En las estadísticas publicadas por el CERN se recoge que el LHC genera más de 30 PB de datos al año y se mantienen un total de 100 PB almacenados de forma permanente en cinta.

- **Variedad:** en el pasado, la mayor parte de los datos que manejábamos eran casi exclusivamente estructurados, entendiendo como tales todos aquellos conjuntos de datos cuya estructura viene definida por un esquema, y encajaban perfectamente en un modelo de base de datos relacional. En la actualidad, más de un 80% de los datos son no estructurados (texto, imágenes, video, etc.). De este modo, se presentan los sistemas Big Data, capaces de manejar datos estructurados, no estructurados y semiestructurados, como una alternativa a los sistemas tradicionales. Además, los sistemas Big Data no solo soportan todos estos tipos de datos, sino que permiten trabajar simultáneamente con varios de ellos, probablemente, procedentes de fuentes muy variadas.

  Un ejemplo de datos no estructurados lo tenemos en los textos en lenguaje natural, según los datos ofrecidos por la propia Twitter se publican unos 500 millones de Tweets por día, esto da una idea de la cantidad de datos no estructurados que posee Twitter. Otro claro ejemplo lo tenemos en Facebook, en abril de 2014, se estimaba que esta red social contaba con una base de datos de 300 PB entre textos e imágenes y generaba un tráfico total de 600 TB/día.

- **Velocidad:** la cantidad de datos que se genera por segundo y que las organizaciones tienen que almacenar, procesar y dar respuesta puede llegar a ser muy alta, esto supone un reto para todos aquellos sistemas que tengan necesidades en tiempo real.

  En *www.internetlivestats.com* publican que se realizan unas 50.000 búsquedas en Google por segundo a las que el motor de búsqueda debe dar una respuesta prácticamente inmediata.

- **Veracidad:** hace referencia a la fiabilidad asociado a ciertos tipos de datos. Las técnicas de Big Data deben aceptar la incertidumbre asociada a determinados datos como pueden ser las previsiones meteorológicas, los factores económicos o las intenciones de compra de futuros clientes. Es fundamental poder averiguar la calidad y la coherencia de los datos manejado con el fin de poder identificar la validez de los resultados obtenidos a partir de los mismos.

  Por ejemplo, una herramienta que analice el correo electrónico que recibe un usuario debe tener en cuenta que una parte de este correo es spam. En la web *www.internetlivestats.es* también ofrecen estadísticas acerca del uso global del correo electrónico, calculan que de los alrededor de 2.500.000 emails que se envían por segundo el 67% es spam.

- **Valor:** se trata del objetivo último de estos sistemas, extraer valor a partir de un conjunto de datos. De nada vale poder acceder de forma eficiente a una gran cantidad de datos totalmente heterogéneos si no es posible extraer alguna conclusión de los mismos.

  Un claro ejemplo lo tenemos en el modelo de negocio de Facebook, que a partir de los datos recolectados a cerca de sus usuarios es capaz de ofrecer una audiencia segmentada, atendiendo a multitud de criterios, a la que las empresas pueden dirigir sus anuncios. En los datos de facturación de 2014, Facebook ha destacado que su facturación por publicidad ha ascendido hasta 3.590 millones de dólares durante el último trimestre del año fiscal, un 58,4% más que en todo 2013.

Algunos autores añaden a estas cinco dimensiones las dos siguientes completando así lo que se conoce como las 7 V´s del Big Data:

- **Variabilidad:** atiende al significado que puedan tener los datos según el contexto, uno de los ejemplos más claros lo tenemos en los sistemas de procesamiento de lenguaje natural, ya que el significado de cada palabra está fuertemente influenciado por el contexto en el que se utiliza.
- **Visualización:** normalmente, una vez los datos han sido analizados y procesados, es necesario mostrar los resultados de forma clara y comprensible. Un punto clave es el control de acceso, ya que permitir visualizar ciertos datos sensibles para la organización a personal no autorizado puede tener consecuencias nefastas.

4.1.2. Características comunes

Las siguientes características son comunes a la mayoría de técnicas utilizadas en sistemas Big Data [5]:

4.1.2.1. Los datos se encuentran distribuidos en una gran cantidad de nodos

Uno de los objetivos de las técnicas basadas en Big Data es la utilización de hardware de propósito general que permita aprovechar la infraestructura de procesamiento de datos con la que ya cuentan las organizaciones (*commodity hardware*). Cada uno de estos servidores, por sí solo, carece de la capacidad de almacenamiento y de la potencia necesaria para hacer frente al procesamiento de grandes cantidades de datos en un tiempo razonable. Esto impone la necesidad de adoptar una arquitectura distribuida en la que todos los servidores disponibles puedan ofrecer un sistema de procesamiento y almacenamiento conjunto. De este modo, se ofrece una capa lógica que permite al programador abstraerse de toda la gestión de la concurrencia y centrarse únicamente en el desarrollo de los algoritmos de procesamiento. Se consiguen minimizar los riesgos con respecto a un enfoque tradicional automatizando todas las tareas de distribución y fusión de los datos así como el tratamiento de fallos parciales.

En la mayoría de los casos, sería impensable que una sola máquina tuviera la capacidad de almacenamiento necesaria para almacenar la totalidad de los datos, no obstante, este no es el único beneficio de adoptar una arquitectura distribuida. También tenemos la posibilidad de duplicar bloques de datos en diferentes máquinas con el fin de asegurar alta disponibilidad de los datos y tolerancia a fallos. Además, permite el procesamiento en paralelo de los datos, cada nodo procesa un subconjunto de los datos totales consiguiendo de este modo una mejora significativa en el rendimiento total.

4.1.2.2. Se mueven las aplicaciones hacia los datos en lugar de al contrario

Dada una aplicación en la que los datos residen en varias máquinas, desde un punto de vista tradicional, se podría optar por una arquitectura en la que una aplicación que residente en un servidor consulta a través de la red los datos necesarios que se encuentran en las demás máquinas. Se trata de una visión centralizada de la capa de aplicación a pesar de que los datos se encuentran distribuidos, este enfoque se centra en la aplicación tomando los datos como un elemento auxilia.

En un entorno Big Data, el coste de enviar los datos por la red puede no ser asumible, en su lugar, es la propia aplicación la que se transmite hacia el nodo en el que se encuentran los datos

donde serán procesados. Además de perjudicar el rendimiento, la transmisión de grandes cantidades de datos puede colapsar la red originando fallos en el sistema. Se añade, de este modo, una cierta complejidad, ya que además de las aplicaciones es necesario mover todas las librerías con las que la aplicación tenga alguna dependencia. La mayor parte de los sistemas Big Data permiten desplegar las aplicaciones de forma centralizada ocultando toda esta complejidad asociada.

4.1.2.3. En la medida de lo posible, los datos se procesan localmente

Se trata de una consecuencia directa de las dos características anteriores, nos encontramos en un entorno distribuido en el que las aplicaciones se mueven hacia los nodos en los que se encuentra los datos. Si además, tenemos en cuenta que los accesos a disco son mucho más eficientes que los accesos a través de la red obtenemos un sistema en el que las aplicaciones se mueven a los nodos para realizar todo el procesamiento de datos de forma local.

No obstante, no siempre es posible acceder a todos los datos de forma local, las tareas se planifican de tal modo que se minimicen los accesos a través de la red. Un claro ejemplo lo tenemos en un gran número de aplicaciones en las que es necesario procesar los resultados intermedios producidos en paralelo por cada nodo para producir un resultado final, es necesario, por lo tanto, transmitir todos estos resultados parciales por la red. Este proceso, en la mayoría de los casos, supone un tiempo muy inferior al consumido por el resto del procesamiento en los nodos.

4.1.2.4. Se priman los accesos secuenciales sobre los accesos aleatorios

Los accesos a disco son más eficientes que los accesos a través de la red, pero no todos los accesos a disco suponen el mismo tiempo. Los discos duros poseen un cabezal lector que se debe posicionar en la dirección en la que se empezará la lectura, una vez hecho esto se realiza una lectura secuencial de los datos. El tiempo de posicionamiento del cabezal no es para nada despreciable sino que supone una penalización importante sobre los tiempos de acceso. Tenemos, por lo tanto, que cuantos más posicionamientos sean necesarios (accesos aleatorios) tendremos un tiempo de acceso peor, mientras que si realizamos accesos secuenciales obtendremos un mejor rendimiento.

En algunos sistemas en los que se busca ofrecer el máximo rendimiento se opta por realizar accesos totalmente secuenciales de disco y, posteriormente, se filtran los datos en la memoria principal.

4.1.3. Modelos de computación

A continuación describiremos los modelos de computación más comunes dentro de las técnicas basadas en Big Data [5].

4.1.3.1. Massively Parallel Processing (MPP) Database System

Los sistemas que utilizan una aproximación basada en MPP dividen los datos en subconjuntos que puedan ser procesados de forma independiente y por lo tanto en paralelo en diferentes nodos. El punto clave de este sistema radica en cómo se dividen los datos para crear los diferentes conjuntos.

Se trata de un parámetro de diseño que debe ser especificado de antemano dependiendo de cada caso.

El criterio utilizado para dividir y repartir los datos entre los nodos del clúster determina, en gran medida, el rendimiento que puede alcanzar cada consulta ya que restringe el número de nodos que podrán trabajar en paralelo de forma simultánea. Se puede dar el caso en que un criterio de división de los datos haga que una consulta se ejecute de forma muy eficiente mientras que otra consulta se vea gravemente perjudicada. Con el fin de minimizar este efecto negativo se duplican los datos y se dividen bajo diferentes criterios: en cada caso se seleccionan los datos particionados bajo el criterio que mejores resultados pueda ofrecer.

Entre los principales productos comerciales que implementan este concepto tenemos IBM Netezza y EMC Greenplum.

4.1.3.2. Sistemas de bases de datos "en memoria"

Desde un punto de vista funcional, se trata de un sistema prácticamente idéntico al caso de los sistemas MPP. La principal diferencia radica en la implementación, estos sistemas se ejecutan en nodos con grandes cantidades de memoria principal en la que se precargan los datos. SAP HANA es un ejemplo de sistema cuyo funcionamiento se basa en este principio

4.1.3.3. MapReduce

Se trata del modelo de programación utilizado en Hadoop con lo que lo describiremos en mayor detalle en los siguientes apartados en los que se analiza este framework en profundidad.

Desde un punto de vista funcional, MapReduce necesita que el usuario defina dos procesos: *map* y *reduce*. Cada nodo planifica una tarea *map*, esta accede a los datos y produce pares clave/valor que serán procesados en la etapa *reduce*. Un número preestablecido de nodos ejecutan la función *reduce* combinando (mediante una función previamente definida) los valores intermedios asociados a cada clave. Finalmente, la salida se escribe en almacenamiento persistente.

MapReduce sigue una arquitectura maestro/esclavo en la que el nodo maestro se encarga de todas las tareas de monitorización y control mientras que los nodos esclavos se encargan de la ejecución propiamente dicha. Profundizaremos en esta arquitectura en la descripción de la implementación que se realiza en Hadoop de este modelo.

Uno de los principales problemas del modelo MapReduce radica en la poca idoneidad para ejecutar algoritmos iterativos, esto cobra especial importancia si tenemos en cuenta que la mayor parte de los algoritmos de análisis de datos pertenecen a este grupo. Este modelo aplicado a algoritmos iterativos requiere que cada una de las iteraciones se ejecute como una tarea *MapReduce* independiente, cada una de estas tareas coge los datos de almacenamiento persistente y vuelca los resultados, de nuevo, en almacenamiento persistente. Esto, en la mayoría de los casos, genera un gran número de accesos innecesarios a disco ya que, potencialmente, los datos que una iteración recibe como entrada se pueden corresponder con la salida de la anterior.

4.1.3.4. Bulk Synchronous Parallel (BSP)

Se trata de un concepto similar a MapReduce. BSP añade una barrera de sincronización en la que todos los procesos (similares a los procesos *map* de MapReduce) envían datos al nodo maestro e intercambian información relevante sobre la ejecución. Una vez que todos los procesos han terminado (se ha completado una iteración) el nodo maestro indica a cada nodo de procesamiento que puede continuar con la siguiente iteración.

La utilización de barreras de sincronización es muy común dentro de los paradigmas de computación paralela y, en este caso, se utiliza con el fin de paliar las deficiencias de rendimiento que presenta MapReduce ante algoritmos iterativos. Podemos encontrar proyectos dentro de la Fundación Apache como Hama o Giraph que implementan este modelo.

4.2. Hadoop

Apache Hadoop [6] es un framework para el desarrollo de sistemas capaces de procesar grandes cantidades de datos utilizando modelos simples de programación. Fue diseñado para ofrecer un sistema escalable, capaz de ejecutarse en grandes centros de procesamiento, aislando a los desarrolladores de las complejidades asociadas a la programación paralela. Está inspirado en los documentos de Google sobre MapReduce [7] y Google File System (GFS) [8], un sistema de ficheros distribuido diseñado por Goolge.

Hadoop fue creado por Doug Cutting, creador de Apache Lucene, una de las librerías más utilizadas para búsqueda e indexado de textos. En concreto, Hadoop surge en 2005 como parte de Apache Nutch, un motor de búsqueda web que formaba parte del proyecto Lucene. En 2006 Doug Cutting entra a formar parte de Yahoo!, empresa que dedica un equipo completo al desarrollo de Hadoop. Es en 2008 cuando se consolida como un proyecto independiente dentro de la Fundación Apache y empieza a ser utilizado por empresas como Last.fm, Facebook o The New York Times, además de la propia Yahoo! [9].

Alrededor del proyecto Apache Hadoop han surgido numerosos nuevos proyectos formando lo que se conoce como el "*Ecosistema Hadoop*". Se trata de iniciativas que aportan soluciones concretas en diversos ámbitos de aplicación, en su mayoría, desarrollados bajo el amparo de la Apache Software Fundation. No obstante, también surgen iniciativas de terceros como puede ser Cloudera Impala (que se discutirá más adelante), todos estos proyectos conforman un ecosistema muy rico y en constante crecimiento.

Las primeras versiones de Hadoop estaban compuestas por dos plataformas software: Hadoop Distributed File System (HDFS) como sistema de almacenamiento y MapReduce como sistema de procesamiento.

- **Hadoop Distributed File System (HDFS):** se trata de un sistema de archivos distribuido que permite que un fichero no se almacene en una única máquina sino que replica la información en varias máquinas consiguiendo así proporcionar alta disponibilidad y mayor rendimiento con accesos en paralelo.
- **MapReduce:** se trata de un framework que permite al programador abstraerse de las tareas propias de la programación paralela. Es decir, permite que un programa escrito de forma convencional se pueda ejecutar en un clúster Hadoop, aprovechando los recursos aportados por todos los servidores que conforman el clúster de forma eficiente.

En la actualidad, esta primera versión coexiste con una nueva versión en la que se ha incluido una arquitectura renovada mucho más flexible que permite una integración más simple con nuevos

componentes software, incluyendo alternativas al framework de procesamiento *MapReduce* original. Esta flexibilidad hace que esta nueva versión, Hadoop 2.0, tenga un mayor rango de casos de uso que la versión original, pensada para el procesamiento *batch* de grandes cantidades de información. Además, esta nueva arquitectura permite una mejor gestión de los recursos disponible proporcionando una mejor escalabilidad con respecto al número de nodos que forman el clúster [5]. En la figura 4.1 se muestra una primera aproximación de esta profunda evolución.

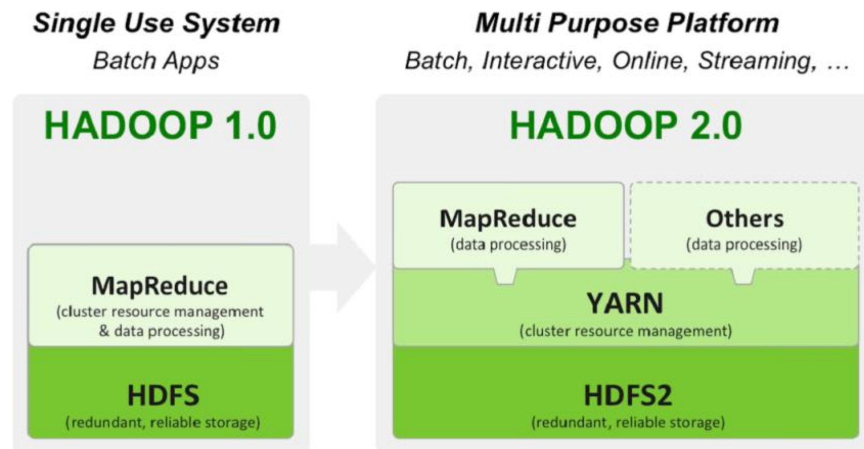


Figura 4.1: Hadoop 2.0

4.2.1. Hadoop Distributed File System

Hadoop Distributed File System (HDFS) consiste en un sistema de archivos distribuido diseñado para dar respuesta a la necesidad de almacenar ficheros de gran tamaño utilizando hardware de propósito general, abaratando de este modo los costes del sistema.

Realmente, HDFS consiste en una capa de abstracción (figura 4.2) que se sitúa sobre el sistema de archivos nativo ofreciendo, como veremos a continuación, alta disponibilidad, tolerancia a fallos y accesos eficientes a ficheros distribuidos en un clúster Hadoop.

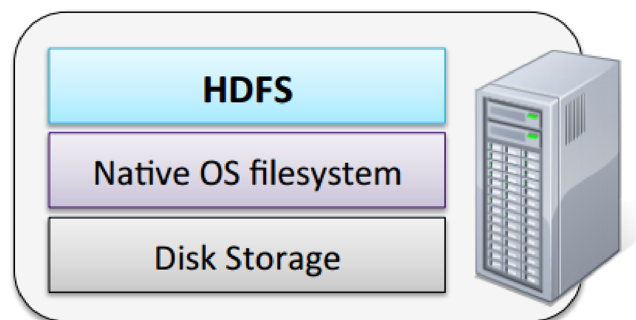


Figura 4.2: Abstracción HDFS

HDFS está diseñado siguiendo una arquitectura maestro/esclavo en la que únicamente existe un maestro que se encarga de mantener el espacio de nombres del sistema de archivos y de gestionar el acceso de los clientes a los datos. Por otro lado, existen una serie de nodos esclavos, generalmente uno por cada nodo del clúster, que se encargan de almacenar los datos.

HDFS está soportado por los siguientes demonios que ofrecen todos los servicios que lo conforman:

- **NameNode**: se ejecuta en el nodo maestro, mantiene el espacio de nombres y el árbol que define su estructura así como todos los metadatos acerca de cada uno de los ficheros y carpetas que lo forman. Esta información se almacena en dos ficheros: *fsimage* y el fichero de log *edits*. Además, el NameNode mantiene información acerca de que DataNode contiene cada bloque, sin embargo, esta información no se mantiene en almacenamiento persistente sino que se construye cuando se inicia el sistema. Se trata de uno de los puntos que será modificado en la versión Hadoop 2.0 por tratarse de un cuello de botella por el que tiene que pasar todas las peticiones de los clientes además de tratarse de un punto único de fallo.
- **Secondary NameNode**: a pesar de que su nombre puede sugerir lo contrario, no se trata de un NameNode de respaldo. Como ya hemos comentado, en el NameNode se mantiene un fichero *fsimage* en el que se almacena el estado del sistema, además, cada vez que se realiza una operación se registra en el fichero de log *edits*. Este último fichero puede llegar a ser demasiado grande cuando el clúster lleve mucho tiempo operativo puesto que únicamente se elimina cuando se reinicia el sistema. El papel del Secondary NameNode es realizar estas operaciones de vaciado del fichero *edits* de forma periódica aplicando los cabios en el fichero *fsimage*, se trata de un proceso que puede llegar a ser muy pesado con lo que se recomienda ejecutar este servicio en un nodo diferente al que se está ejecutando el NameNode.
- **DataNode**: se ejecuta en los nodos esclavos del clúster, se encargan de almacenar los datos y, como veremos más adelantes, atender las peticiones de lectura/escritura de los clientes.

Los ficheros se almacenan físicamente en el clúster Hadoop divididos en bloques, originalmente, cada bloque contaba con un tamaño de 64MB siendo cada vez más frecuente una configuración de 128MB. Es importante destacar que si el fichero no posee un tamaño múltiplo del tamaño de bloque, el último bloque simplemente es más pequeño, esto es de especial importancia teniendo en cuenta los tamaños de bloque que estamos manejando. Una aproximación en la que todos los bloques tienen un tamaño fijo podría suponer desperdiciar una gran cantidad de espacio de almacenamiento rellenando los últimos bloques de cada fichero con datos inservibles.

Cada bloque se almacena replicado en los DataNode del clúster con el fin de proporcionar tolerancia a fallos, el factor de replicación por defecto en Hadoop es 3 y sigue el siguiente esquema: se almacena un bloque en un nodo del rack local en el que se está ejecutando la aplicación cliente que ha solicitado el almacenamiento (si la aplicación cliente no se ejecuta en el clúster se selecciona al azar), el segundo bloque replicado se almacena en un nodo perteneciente a un rack diferente al del primer nodo y, por último, se almacena la tercera copia en un nodo del mismo rack al que pertenece este segundo nodo. Esto permite reducir el tráfico entre diferentes racks asegurando un mayor rendimiento puesto que, en la mayor parte de los casos, el ancho de banda disponible entre nodos del mismo rack es mayor que entre diferentes racks. Además, se apoya en el hecho de que el fallo de un rack es mucho menos frecuente que el fallo de un nodo. El valor de replicación puede variar dependiendo de cada sistema, se trata de un compromiso claro entre rendimiento y alta disponibilidad, así en sistemas en los que tengamos algún mecanismo Hardware que proporcione resistencia a fallos como puede ser una configuración en RAID podemos reducir este factor.

Como ya hemos visto, prima la localidad de los datos, esto se traduce en que cuando se ejecuta un algoritmo en el clúster es la aplicación la que se mueve a los DataNode en los que se encuentran los datos y no al revés.

4.2.1.1. Proceso de escritura en HDFS

En la figura 4.3 se puede ver el proceso de escritura en detalle, en general, HDFS no soporta actualizaciones sobre ficheros con lo que el proceso de escritura lleva implícito la creación del fichero. No obstante, se soportan las escrituras sobre ficheros existentes en modo *"append"* pensado para ser usado por HBase, pero no se recomienda su uso en clientes de propósito general.

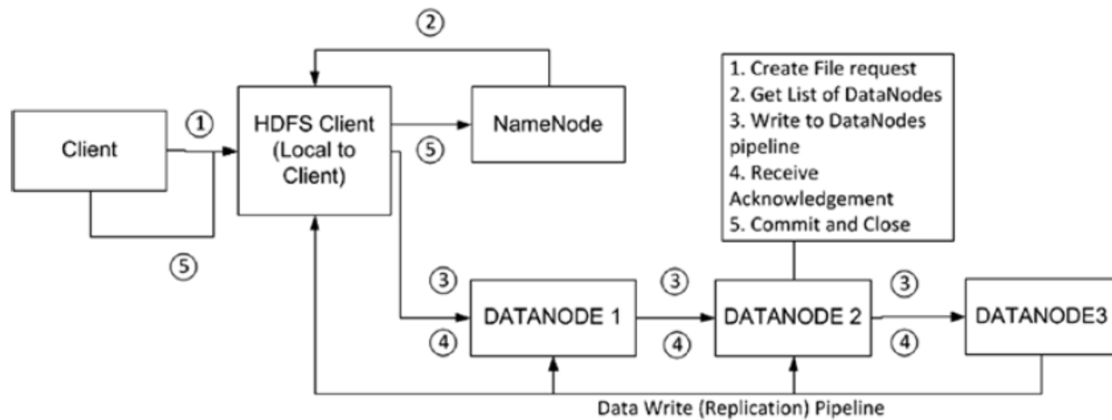


Figura 4.3: Escritura en HDFS

1. El cliente empieza a copiar el contenido del fichero a un fichero temporal en su sistema de archivos local antes de contactar con el NameNode.
2. Cuando el fichero temporal alcanza el tamaño de bloque, el cliente contacta con el NameNode.
3. El NameNode crea una entrada para el fichero en la estructura de HDFS y devuelve al cliente información acerca del identificador del bloque y la localización del DataNode, además, también se incluye información acerca de los DataNode dónde se replicará el bloque.
4. El cliente utiliza la información recibida por el NameNode para enviar el fichero local al primer DataNode y se crea el nuevo fichero en el sistema de archivos local del DataNode.
   4.1. El primer DataNode recibe los datos en pequeños paquetes (normalmente 4 KB), a medida que los recibe empieza a escribirlos en disco y a enviarlos al segundo DataNode.
   4.2. El segundo DataNode realiza una operación similar, a medida que recibe los bloques de datos los almacena en disco y los reenvía al tercer DataNode.
   4.3. El último DataNode simplemente escribe los datos a disco, de este modo se consigue el factor de replicación 3.
   4.4. Cada uno de los DataNode envía un paquete de confirmación al anterior, por último, el primero se lo envía al cliente.
   4.5. Cuando el cliente recibe la confirmación, todos los datos se han almacenado correctamente con el factor de replicación adecuado. En este momento, el cliente envía otro paquete de confirmación al NameNode.
   4.6. Si algún DataNode falla en el proceso de escritura, los datos pueden haber quedado almacenados en otros DataNode, en esta situación, se informa al NameNode que tomará las acciones pertinentes para intentar conseguir el factor de replicación adecuado.
   4.7. Se genera un *checksum* por cada bloque en HDFS que se utilizará para comprobar su integridad cuando sea leído.

5. Por último, el NameNode confirma que el nuevo fichero se ha creado correctamente y lo hace visible para los clientes. Si se produce un fallo en el NameNode antes de que se confirme el nuevo fichero se pierden todos los datos.

4.2.1.2. Proceso de lectura en HDFS

A continuación detallaremos el proceso de lectura (figura 4.4) de un fichero que se encuentra almacenado en HDFS.

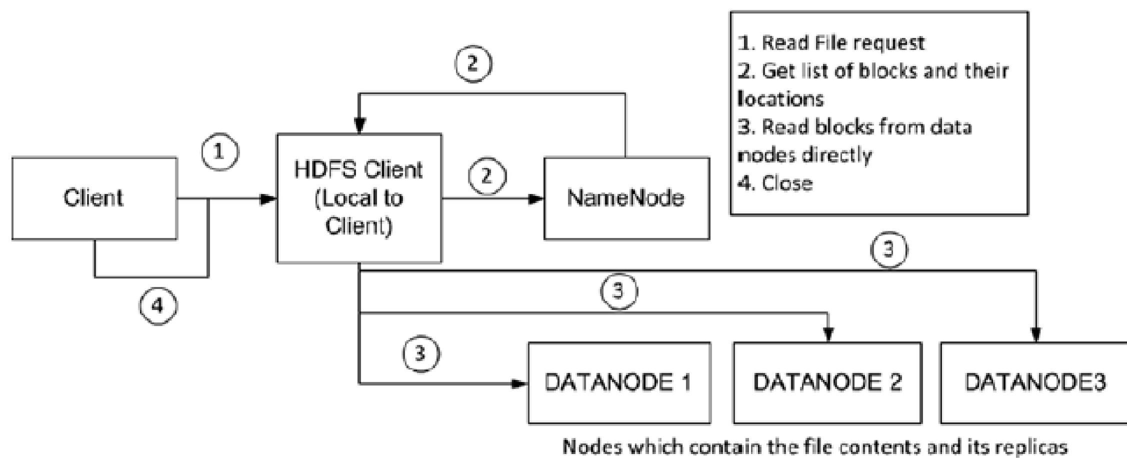


Figura 4.4: Lectura de HDFS

1. El cliente establece una conexión con el NameNode y recibe la lista de bloques y las localizaciones de todas las réplicas de cada bloque.
2. El cliente contacta con el DataNode para iniciar la lectura, en caso de que un DataNode no responda reintenta la conexión con un DataNode que mantenga una réplica.
3. Se calcula el *checksum* sobre el bloque que se acaba de leer y se compara con el que se había generado cuando se había escrito el bloque, si no coinciden se descarta y se lee de uno de los DataNode que contenga una réplica del bloque.

4.2.1.3. Borrado de un fichero en HDFS

El proceso de borrado es muy simple, a continuación se muestran los pasos que sigue el sistema para eliminar un fichero:

1. El NameNode actualiza los metadatos para indicar que el fichero se ha movido a la carpeta /trash, se trata de una operación muy rápida. El fichero permanecerá en esta carpeta durante un periodo de tiempo en el que se podrá recuperar fácilmente moviéndolo de la carpeta /trash a la carpeta deseada.
2. Una vez se ha terminado el periodo de tiempo en que el fichero se mantiene en la carpeta /trash, el NameNode elimina el fichero del espacio de nombres de HDFS.
3. Los bloques que formaban el fichero se liberan y, por lo tanto, se incrementa el espacio disponible.

4.2.1.4. HDFS alta disponibilidad

Como ya hemos visto, uno de los principales problemas de las primeras versiones de HDFS era la presencia de un único punto de fallo, el NameNode. Si el nodo que contenía este demonio sufría un fallo o necesitaba reiniciarse por tareas de mantenimiento, todo el clúster permanecía inaccesible hasta que volviera a estar operativo. Esta *alta disponibilidad* [10] se consigue añadiendo un segundo NameNode de tal forma que mientras uno se mantiene en estado activo, el otro se mantiene en *standby*. Si se produce un fallo en el NameNode activo, el otro pasa a estado activo y continua atendiendo las peticiones de los clientes. Este nuevo modelo tiene varias implicaciones adicionales:

- Los dos NameNodes necesitan un sistema de almacenamiento compartido de alta disponibilidad para poder compartir el fichero de log *edit*, como puede ser NFS.
- Los DataNode deben enviar los reportes acerca de los bloques que almacenan a ambos puesto que, como ya vimos, esta estructura se mantiene en memoria y no en disco.
- Los clientes deben configurarse de forma adecuada, si detectan que un NameNode no responde deben conectarse al otro de forma transparente al usuario.

4.2.1.5. HDFS Federation

En la implementación que hemos comentado el NameNode mantiene una estructura de datos en memoria en la que se almacenan las referencias a todos los ficheros y bloques presentes en el sistema de archivos, si aumentamos drásticamente el número de nodos y el número de ficheros que se almacenan podemos encontrarnos con que la memoria disponible en el NameNode marca el límite al que podemos escalar nuestro sistema. Con el fin de corregir esta circunstancia, en Hadoop 2.0, se ha incluido el concepto de **HDFS Federation** [10]. La nueva arquitectura se basa en la idea de introducir varios NameNode de tal forma que se encargue cada uno de ellos de manejar una porción del espacio de nombres.

Todos los DataNodes almacenan bloques manejados por todos los NameNodes. Se define un pool de bloques como un conjunto de bloques que pertenecen a un mismo espacio de nombres, de este modo, los DataNodes almacenan bloques de todos los pools de bloques del clúster. Se consigue así que el fallo de un NameNode no impida que todos los DataNode sigan sirviendo bloques a los clientes.

4.2.2. Hadoop MapReduce

MapReduce [5] es un modelo de progamación desarrollado originalmetne por Google y pensado para ejecutarse en grandes clúster y trabajar con grandes conjuntos de datos. Una de las principales fortalezas de este sistema radica en la capa de abstracción que incluye con respectoa a los sistemas distribuidos tradicionales. Los desarrolladores únicamente tienen que proporcionar la implementación para las funciones *map* y *reduce* y el sistema se encargará de realizar de forma totalmente transparente la planificación y la coordinación entre los diferentes nodos.

- **Map:** los datos se dividen en subconjuntos y son procesados de forma independiente en paralelo en los nodos del clúster.
- **Reduce:** esta etapa recibe como entrada la salida de *Map* y lleva a cabo una función de agregación sobre todos los subconjuntos de datos que están asociados en un único nodo. Este paso puede no ser necesario en algunas aplicaciones MapReduce.

Entre ambas etapas existe una fase intermedia denominada "**Sort and Shuffle"** que se encarga de asociar las salidas de los diferentes Map con el Reduce correspondiente. A continuación detallaremos todo el proceso por medio de un ejemplo clásico dentro del mundo del Big Data, contar el número de repeticiones de cada palabra en un fichero de texto. En la figura 4.5 vemos un esquema en el que se describe este ejemplo.

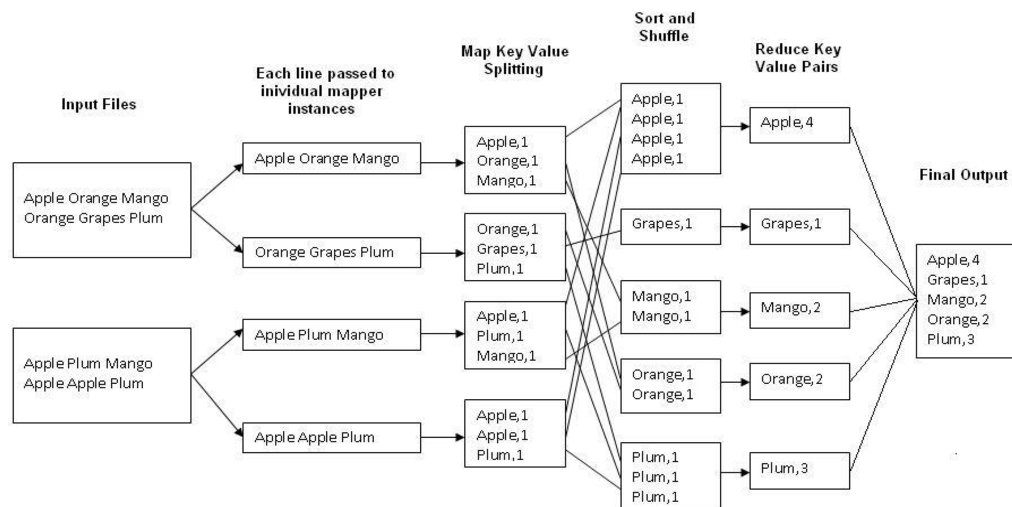


Figura 4.5: MapReduce

En este caso contamos con dos ficheros de texto que dividiremos en líneas, estas líneas serán las entradas de la operación *Map*. Cada proceso *Map* produce una serie de elementos clave valor en los que la clave es la palabra y asigna como valor un "1", siguiendo el esquema <{*palabra*}, 1>. Por último contamos con varios procesos Reduce que se encargan de agregar elementos del tipo <{*palabra*}, [1,1,…,1]> sumando todos lo valores asociados a la clave. Cabe destacar que antes de ejecutar la etapa Reduce se ha llevado a cabo la operación *Sort and Shuffle* que se ha encargado de ordenar todos los elementos por clave de tal forma que cada proceso *Reduce* unicamente recibe elementos asociados a una clave. Se podría eliminar esta etapa y contar con unico proceso *Reduce* que realizase las agragaciones sobre todas las claves, sin embargo, utilizando el esquema propuesto conseguimos que tanto la etapa *Map* como la etapa *Reduce* se pueda ejecutar en paralelo en los diferentes nodos del clúster.

Al igual que ocurría con el caso de HDFS, MapReduce siguen una arquitectura maestro/esclavo que se encuentra soportada por los siguientes demonios:

- **TaskTracker:** se ejecuta en cada uno de los nodos del clúster Hadoop y acepta peticiones individuales de ejecución de procesos *Map*, *Reduce* y *Shuffle*. Cada TaskTracker mantiene una serie de *slots*, normalmente uno por cada núcleo con los que cuenta el nodo. Cada *slot* ocupado se corresponde con una ejecución que se está llevando a cabo. Los TaskTracker envian de forma periódica mensajes *heartbeat* al JobTraker en los que, además de enviar información acerca del estado de "*salud*" del nodo se informa del número de *slots* libres, esta información se utilizará para planificar nuevas ejecuciones.

- **JobTracker:** es el encargado de lanzar y monitorizar los jobs MapReduce. A continuación, describiremos el proceso completo que se sigue a la hora de lanzar un job MapReduce y que podemos ver en la figura 4.6.

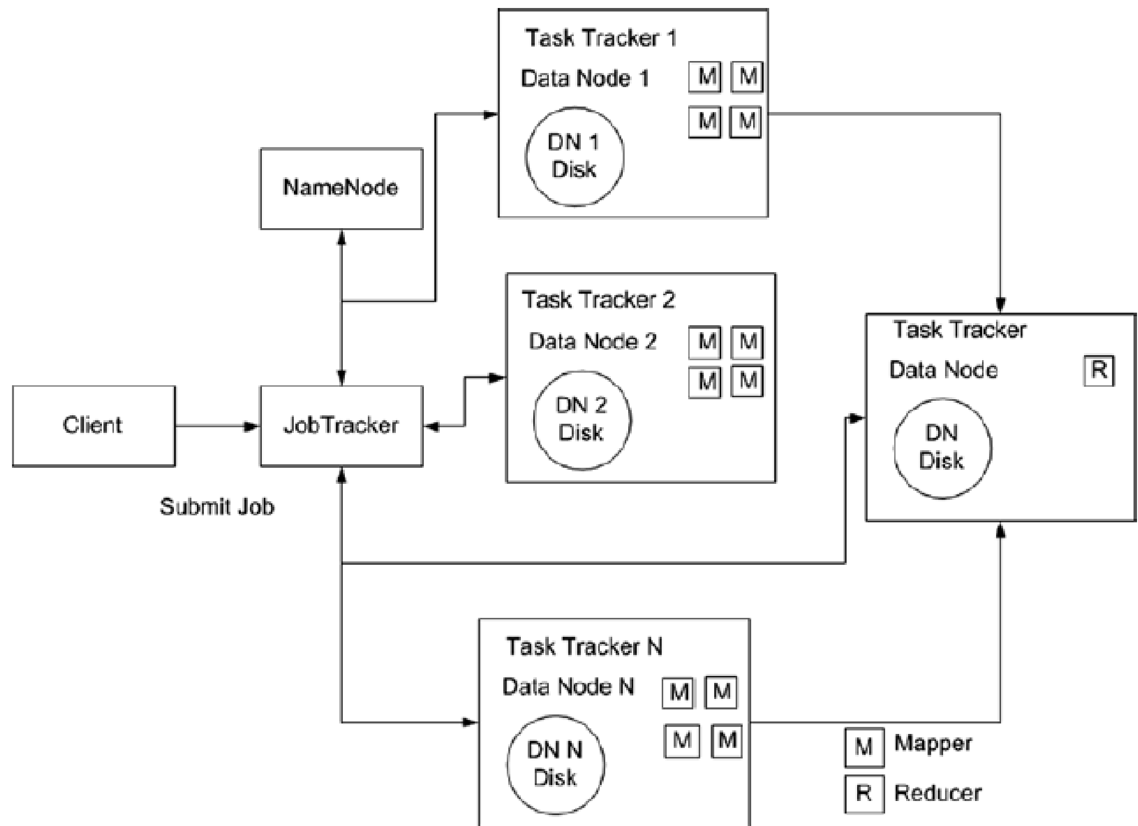


Figura 4.6: Ejecución de job MapReduce

1. El JobTacker recibe una solicitud de ejecución del cliente.
2. En el caso de que se necesite como entradas datos almacenados en HDFS el JobTraker consulta el NameNode acerca de los DataNode que contienen los bloques necesarios.
3. El JobTraker planifica la ejecución determinando el número de tareas *Map* y *Reduce* que se deben ejecutar, la planificación se adapta, en la medida de lo posible, a la forma en que los bloques estan almacenados.
4. El JobTraker envia las tareas correspondientes a cada TaskTracker para su ejecución. Se realiza una monitorización de los TaskTracker, en caso de que alguno falle, se replanifica el trabajo a un nodo diferente.
5. Cuando todas las tareas se han completado correctamente, el JobTracker actualiza el estado del job a finalizado con éxito, en caso de que algunas tareas fallen de forma repetida (el número de reintentos se puede configurar) se reportará un fallo en el job.
6. Los clientes pueden consultar el JobTracker acerca del estado del job.

Al igual que ocurria en el caso de HDFS el JobTrakcer se comporta como un único punto de fallo así como un cuello de botella dado que todas las ejecuciones que se realizan en el clúster pasan por este punto. Veremos como en Hadoop 2.0 esta arquitectura cambia radiclamente con el fin de solucionar estos problemas de las primeras versiones.

4.2.2.1. Hadoop MapReduce v2

Se trata de la capa que más cambios ha sufrido en Hadoop 2.0. Como ya adelantábamos en la sección anterior, se ha introducido una arquitectura totalmente nueva que confiere una mayor flexibilidad al entorno Hadoop permitiéndole ejecutar de forma nativa nuevos frameworks de procesamiento alternativos a MapReduce.

Arquitectura YARN

El JobTracker tenía una doble funcionalidad, por un lado realizaba tareas de gestión de los recursos y, por otro lado, planificaba y monitorizaba los *jobs* de MapReduce. En YARN se busca separar estas funcionalidades introduciendo un gestor de recursos global (*Resource Manager*) y un componente que se encargue de la ejecución de cada aplicación (Application Master). Tanto el JobTracker como el TaskTracker desaparecen en esta nueva arquitectura en favor de otros nuevos componentes que describiremos más adelante. En la figura 4.7 se puede ver el concepto de que los nuevos sistemas de procesamiento se ejecutan **en** Hadoop en lugar de **sobre** Hadoop.

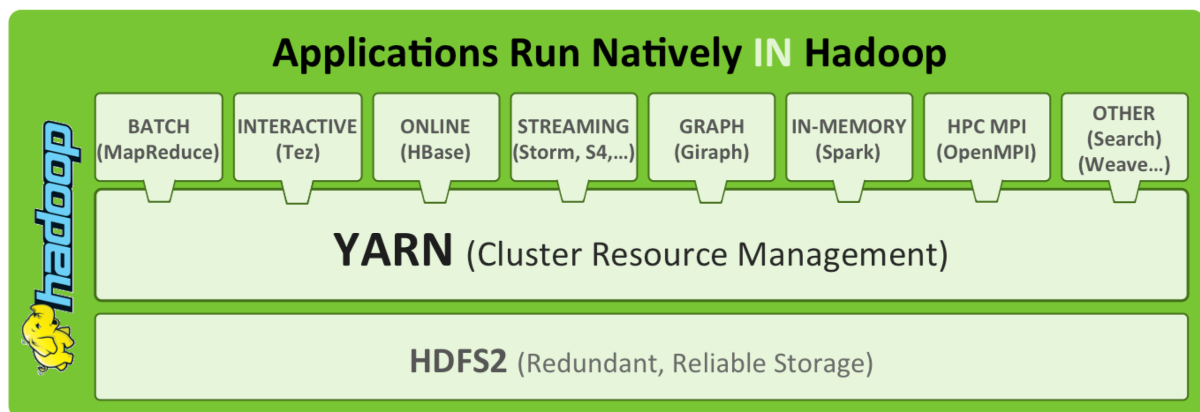


Figura 4.7: YARN

Los componentes que dan soporte a YARN son los siguientes:

- **Contenedores:** consiste en la unidad de computación que se maneja dentro de YARN, cada nodo cuenta con varios contenedores, pero estos no pueden pasar de un nodo a otro. Se trata de un concepto similar a los *slots* que manejaban los TaskTracker en MapReduce v1. Actualmente, los recursos que agrupan los contenedores son núcleos de CPU y megas de memoria principal.
- **ResourceManager:** es el encargado de asignar los recursos disponibles en el clúster a las diferentes aplicaciones de forma eficiente. Actualmente, las tareas de creación y monitorización de recursos se delegan en el NodeManager presente en cada uno de los nodos con el fin de conseguir mayor escalabilidad que el JobTracker de MapReduce que hemos visto.
- **NodeManager:** se trata del servicio que juega el papel de esclavo en la arquitectura maestro/esclavo, se ejecuta en cada uno de los nodos del clúster y recibe peticiones del ResourceManager acerca de cómo se reservan los contenedores para cada aplicación.
- **ApplicationMaster:** se trata de la principal diferencia con respecto a MapReduce v1, es un componente propio de cada framework de computación y se encarga de negociar con el ResourceManager la obtención de nuevos contenedores.

En la figura 4.8 se da una visión global del funcionamiento de esta arquitectura cuando lanzamos una nueva aplicación:

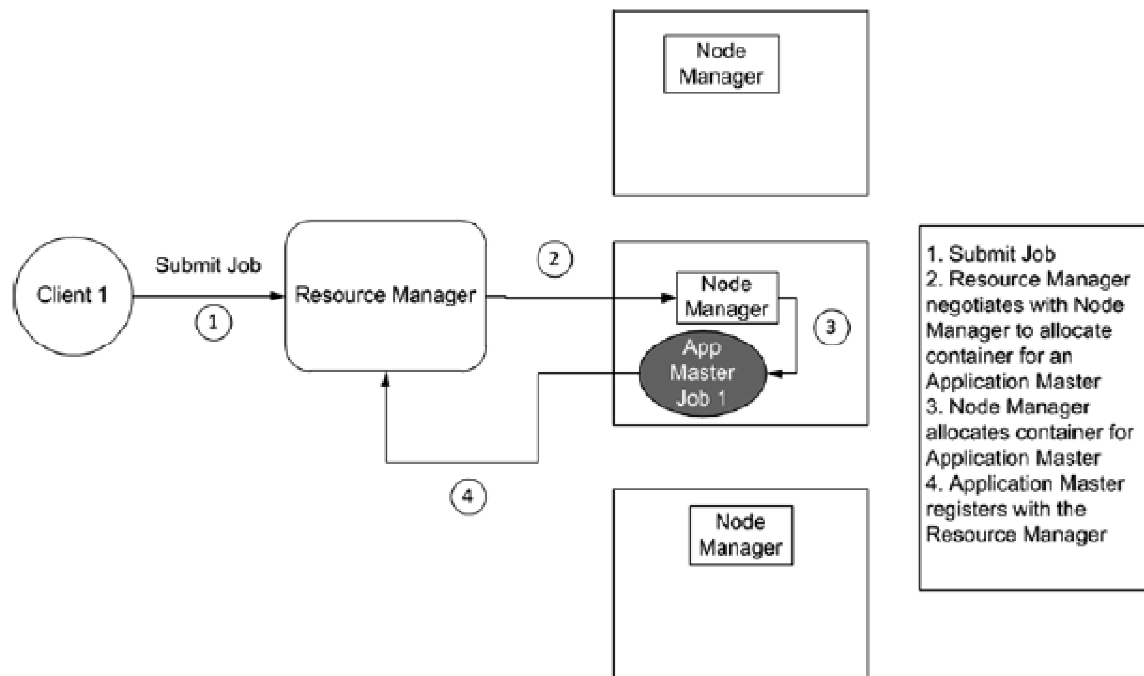


Figura 4.8: Ejecución de job en YARN

Los pasos a seguir son los siguientes:

1. El programa cliente envía una aplicación al clúster para su ejecución.
2. El ResourceManager reserva un contenedor en un nodo en el que ejecutará el AppliationMaster propio del tipo de aplicación que se ha enviado al clúster por parte del cliente.
3. El ApplicationMaster se registra en el ResourceManager con el fin de que el cliente pueda consultar el estado de la ejecución.
4. El ApplicationMaster negocia la utilización de recursos con el ResourceManager a través del concepto de *contenedor* que ya hemos comentado.
5. La aplicación se ejecuta en los contenedores reservados en el paso anterior y se comunica con el ApplicationMaster a través de un protocolo dependiente de cada aplicación con el fin de informar acerca del estado de la ejecución.
6. El programa cliente también se comunica con el ApplicationMaster a través de un protocolo dependiente de la aplicación. El cliente obtiene una referencia del ApplicationMaster a través del registro que se ha realizado en el paso 3 en el ResourceManager.

Esta arquitectura soluciona alguno de los problemas presentes en Hadoop 1.0 como la mejor escalabilidad del ResourceManager frente al JobTracker, sin embargo, el ResourceManager sigue siendo un único punto de fallo. Al igual que ocurría en el caso de HDFS con el NameNode, se despliegan dos ResourceManager, uno en estado activo y otro en *standby*, de este modo se consigue un mayor nivel de redundancia en este punto que resulta crítico para el funcionamiento del sistema.

4.2.3. Distribuciones

Como ya hemos visto, existen diferentes versiones de Hadoop que siguen siendo utilizadas de forma simultánea. Además de estas versiones principales, existen varias ramas de desarrollo independientes. El ecosistema Hadoop está en constante crecimiento, con un gran número de proyectos que cubren una gran variedad de casos de uso, todo ello hace que desplegar y configurar correctamente un clúster Hadoop pueda llegar a ser muy complejo. Con el fin de reducir esta complejidad surgen una serie de distribuciones que integran diversos servicios en Hadoop ofreciendo un único producto testado y con garantías. Además, los proveedores de estas distribuciones pueden añadir componentes propietarios. Esto puede suponer un problema a la hora de realizar una migración del sistema a una distribución diferente con lo que la decisión inicial cobra especial relevancia.

A continuación comentaremos las distribuciones de Hadoop más extendidas [10] y seleccionaremos la más adecuada para nuestro proyecto y que, posteriormente, desplegaremos en el entorno de pruebas.

4.2.3.1. Cloudera

Cloudera nace en marzo de 2009 con el principal objetivo de ofrecer soporte, servicios y formación acerca de Apache Hadoop y todo su ecosistema. Su principal producto se denomina **Cloudera Distribution of Hadoop (CDH)** [11] que consiste en una distribución de Hadoop con licencia 100% *open source*. Una de sus características más destacables es la inclusión de *Cloudera Manager*, una herramienta gráfica que facilita las labores de instalación y mantenimiento del clúster. Las características que ofrece esta aplicación forman junto con los diferentes servicios de soporte disponibles la principal diferencia entre las versiones gratuitas y las versiones que se distribuyen bajo subscripción anual.

En la siguiente figura se puede ver una visión general de la arquitectura de CDH correspondiente a la versión 5.4, la más reciente en el momento de realizar este proyecto.

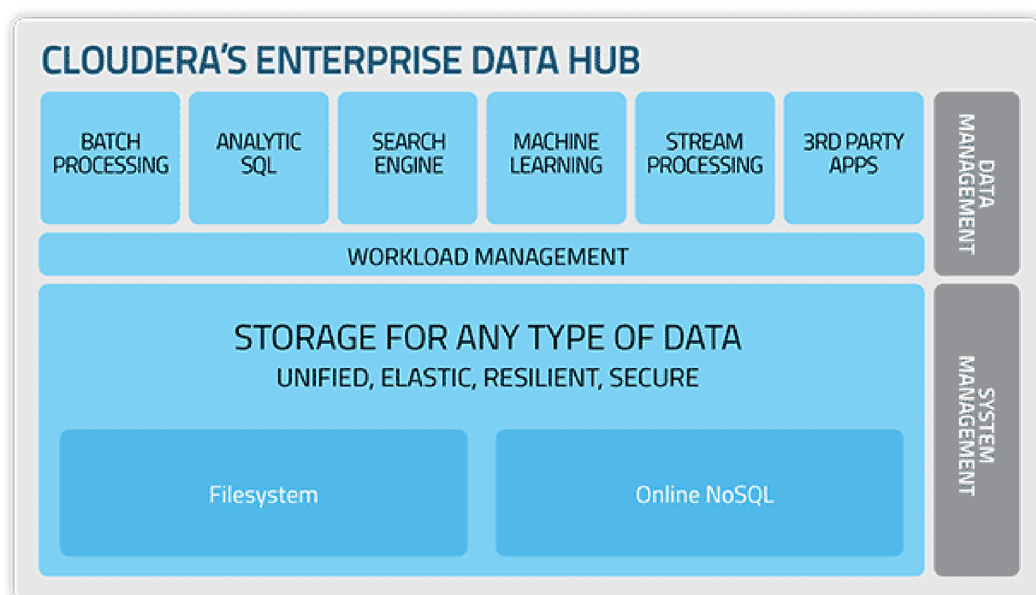


Figura 4.9: Arquitectura CDH

4.2.3.2. Hortonworks

Hortonworks fue fundado en 2011 por un grupo de ingenieros que pertenecían al grupo original de desarrollo de Hadoop dentro de Yahoo!. Su distribución recibe el nombre de **Hortonworks Data Platform (HDP)** [12], se trata de un producto totalmente gratuito pero están disponibles una serie de servicios de soporte y formación de pago. HDP incluye un sistema de gestión y administración del clúster similar a *Cloudera Manager* denominado *Apache Ambari*. A diferencia de *Cloudera Manager* no se trata de una herramienta propietaria aunque no ha alcanzado todavía el mismo nivel de madurez. En la figura 4.10 se muestra la arquitectura general de HDP.

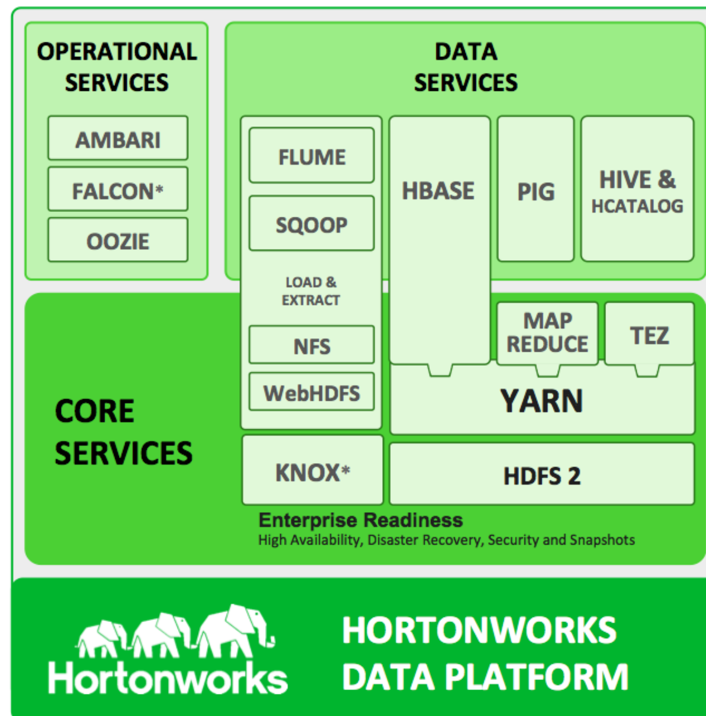


Figura 4.10: Arquitectura HDP

4.2.3.3. MapR

MapR [13] nace en 2009 centrado en prestar servicios empresariales sobre Hadoop. Esta distribución sustituye HDFS por un sistema de archivos basado en NFS compatible con POSIX. No obstante, se ofrecen sistemas para asegurar la compatibilidad con otras versiones incluyendo HDFS. MapR también ofrece un sistema de gestión propio.

Esta distribución está disponible bajo las siguientes nomenclaturas M3, M5 y M7. M5 se trata de la distribución comercial básica para empresas, M3 es la versión gratuita que carece de algunas características como alta disponibilidad. Por último, M7 consiste en un producto con características más avanzadas que incluye una versión revisada del API de HBase.

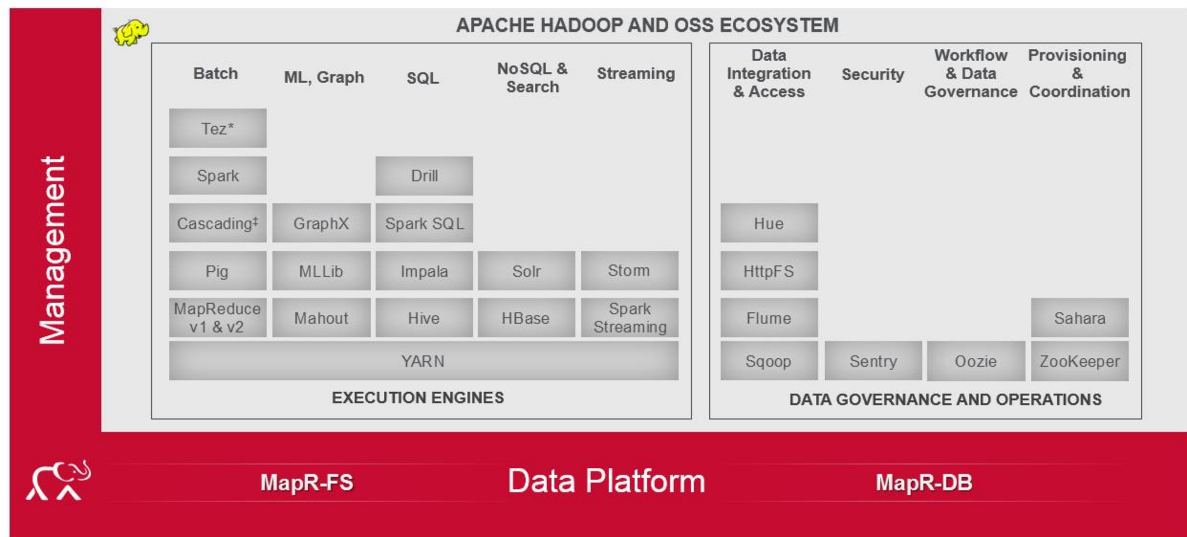


Figura 4.11: Arquitectura MapR

4.2.4. Ecosistema Hadoop

En esta sección se cubren todas las tecnologías que se han barajado a la hora de implementar nuestro sistema. En el capítulo de *Diseño* se pueden ver las tecnologías que finalmente se han seleccionado así como el rol que juegan dentro de la arquitectura final del sistema.

4.2.4.1. Apache Avro

Apache Avro [14] consiste en un sistema de serialización basado en esquemas, estos esquemas se almacenan junto a los datos de tal forma que no es necesario almacenar una etiqueta para cada uno de los valores reduciendo el overhead asociado. De este modo se consigue un sistema más eficiente y con un menor consumo de espacio en disco con respecto a otros sistemas de serialización como el estándar de Java. Estos esquemas se almacenan en formato JSON.

Para realizar las lecturas es necesario obtener primero el esquema asociado a los datos almacenados con el fin de poder interpretarlos correctamente.

En nuestra implementación, utilizaremos esta herramienta para la serialización de todos los datos desde el momento en que el sistema de Dispensación Electrónica los genera hasta que son recibidos por el sistema de procesamiento.

4.2.4.2. Apache Flume

Flume [15] es una herramienta distribuida, robusta y de alta disponibilidad para recolectar, agregar y mover grandes cantidades de datos de log generados por diversas fuentes. Se basa en una arquitectura simple y flexible que permite la retransmisión de datos en tiempo real incorporando mecanismos de recuperación de errores (figura 4.14). Los componentes principales dentro de Flume se denominan agentes, estos están formados por *fuente, canal* y *sumidero*. La *fuente* recibe eventos

de terceros sistemas que están generando los datos que se desean retransmitir, la *fuente* almacena estos eventos en el *canal* que actúa a modo de buffer intermedio que permite desacoplar los dos extremos del agente Flume. Por último, tenemos el *sumidero* que se encarga de entregar los eventos que se encuentran en el canal al sistema al que queremos que lleguen los datos. En la figura 4.12, a modo de ejemplo, los datos recibidos se almacenan en HDFS. Cabe destacar que este se trata de un ejemplo sencillo, pero en realidad, un mismo agente Flume puede contar con un número arbitrario de cada uno de sus componentes, de este modo, podemos tener múltiples *fuentes* que almacenan datos en uno o varios *canales* y que son consumidos por uno o varios *sumideros*.

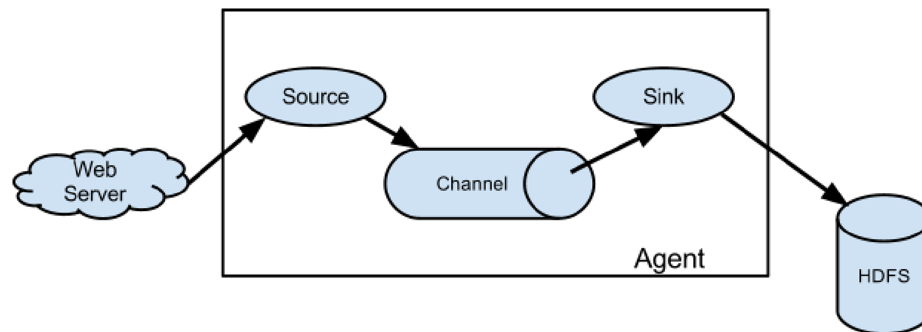


Figura 4.12: Agente Flume

Existen una gran variedad de tipos de *fuentes, canales* y *sumideros*, además de la posibilidad de crear nuevos tipos *ad hoc*. Uno de los tipos que resulta especialmente útil consiste en las *fuentes* y los *sumideros* que utilizan Avro. Estos serializan los datos que se retransmiten a través de esta herramienta. La utilización de Avro facilita la creación de arquitecturas multicapa de agentes Flume tal y como se muestra en la figura 4.13:

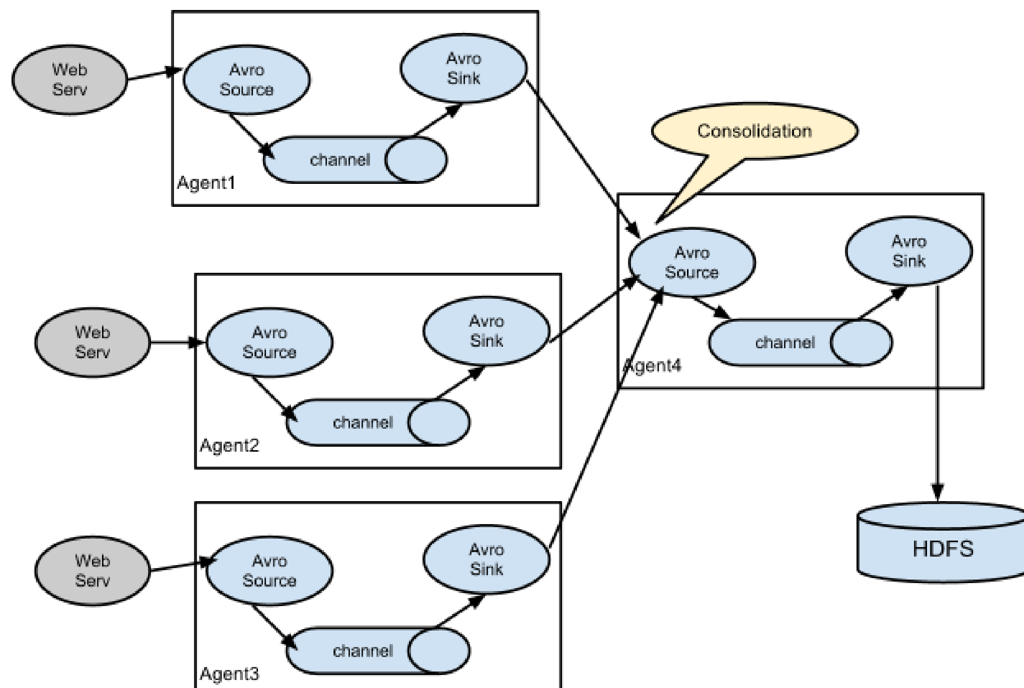


Figura 4.13: Arquitectura Flume multicapa

De este modo, se consigue *consolidar* en un único flujo de datos los eventos producidos por diversas fuentes y enviados a través de diferentes agentes Flume. Podemos pensar en un escenario en el que cada uno de los agentes se ejecuta en un nodo independiente. Un fallo en el nodo que alberga el algente *Agent1*, no supone ningún problema para ninguno de los demás agentes por lo que

los otros dos sistemas que están generando eventos podrán seguir retransmitiéndolos. A pesar de esto, tenemos un único punto de fallo en agente *Agent4,* puesto que si este agente falla por cualquier motivo se corta la comunicación con HDFS (en este caso los eventos se almacenan deserializados en HDFS). En todo caso, esto puede no suponer un problema ya que los nuevos datos generados se almacenan en los canales de los nodos de la primera capa hasta que el otro agente Flume se recupere. En algunos sistemas puede no ser aceptable que se interrumpa la comunicación a pesar de que no haya pérdida de datos. En estos casos necesitaríamos una segunda capa con mayor redundancia de agentes Flume o incluso un mayor número de capas en arquitecturas muy complejas en las que los datos provienen de múltiples sistemas totalmente heterogéneos en las que es necesario realizar una consolidación parcial en una capa intermedia.

En lo que se refiere a los canales, existen dos tipos fundamentales, *en memoria* y como un *archivo* [16]. Los canales que almacenan los datos en memoria cuentan con unos tiempos de acceso más reducidos otorgando mayor rendimiento al sistema, sin embargo, si se produce algún fallo en el nodo que alberga el agente Flume los datos se pierden. Además, este tipo de canales tienen una capacidad más reducida que un canal basado en un fichero en el que los eventos se almacenan en el sistema de ficheros del nodo. Cabe destacar que existen otros tipos de canales más complejos como los que se construyen sobre un clúster Kafka, tal y como se describe en el siguiente apartado, o la utilización de un conector JDBC para almacenar los eventos en una base de datos.

Utilizaremos Flume para retransmitir los eventos desde los nodos del sistema de Dispensación Electrónica a los nodos del clúster Hadoop. Hemos barajado Kafka como alternativa a la hora de implementar esta parte del sistema pero, como veremos en el siguiente apartado, Flume se adapta mejor a los requisitos que definen nuestro sistema.

4.2.4.3. Apache Kafka

Apache Kafka [17] consiste en una herramienta distribuida y con redundancia que ofrece las funcionalidades de un sistema de envío de mensajes. En la figura 4.14 se puede ver una visión de alto nivel de la arquitectura del sistema, Kafka mantiene los mensajes organizados en categorías denominadas *topics*, existen una serie de *productores* que publican los mensajes en el clúster Kafka, y por otro lado, también existen una serie de procesos que se suscriben a los *topics* y que se denominan *consumidores*.

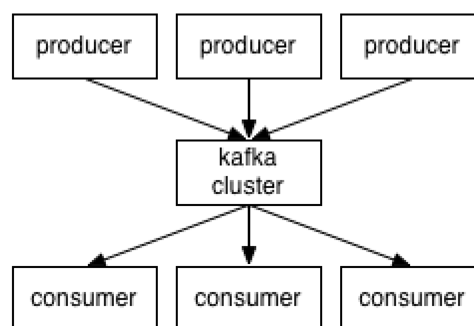


Figura 4.14: Arquitectura Kafka

Como ya hemos comentado, un *topic* consiste en una categoría bajo la que se agrupan los mensajes, cada uno de los *topic* se encuentra particionado dentro de clúster Kafaka tal y como se muestra en la figura 4.15.

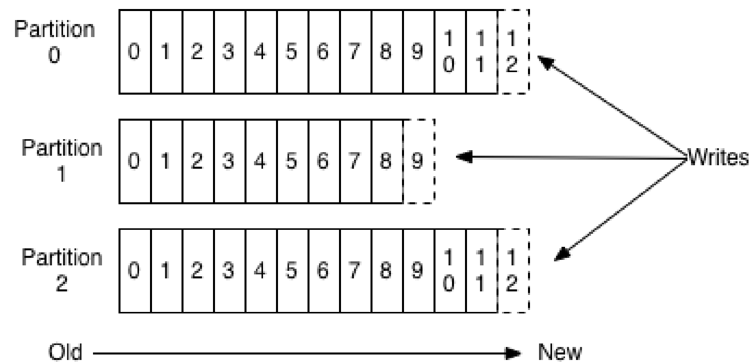


Figura 4.15: Topic de Kafka

Cada partición consiste en una secuencia ordenada de mensajes a la que se están continuamente añadiendo nuevos mensajes. Cada uno de los mensajes tiene asignado un id secuencial que lo identifica dentro de la partición. Kafka mantiene todos los mensajes que han sido publicados, hayan sido consumidos o no, durante un periodo de tiempo específico que se puede configurar, después de este periodo los mensajes se descartan para liberar espacio.

Cada consumidor mantiene la posición del último mensaje consumido, llamado “offset”. Habitualmente, esta posición se incrementa de forma lineal de tal forma que los mensajes se consumen por orden, no obstante, dado que la posición la controla el consumidor puede leer los mensajes en un orden arbitrario.

El uso de particiones permite escalar el tamaño de cada uno de los *topics* dividiendo los datos en múltiples nodos de forma redundante con el fin de conseguir tolerancia a fallos. Existe un nodo “líder” y un conjunto de nodos “seguidores”. El nodo líder maneja las peticiones de lectura y escritura mientras que los seguidores mantienen una copia del líder de forma automática. Todos los nodos actúan como líder para alguna de las particiones mientras que actúan como seguidores para otras.

Los consumidores pueden formar grupos de tal forma que cada uno de los mensajes de un *topic* únicamente se entrega a uno de los clientes de cada grupo. En la figura 4.16 podemos ver un *topic* que cuenta con cuatro particiones alojadas en los servidores *Server 1* y *Server 2*, además, tenemos dos grupos de consumidores. Podemos ver como en esta configuración es posibles entregar mensajes de forma simultánea y en paralelo a cada uno de los consumidores.

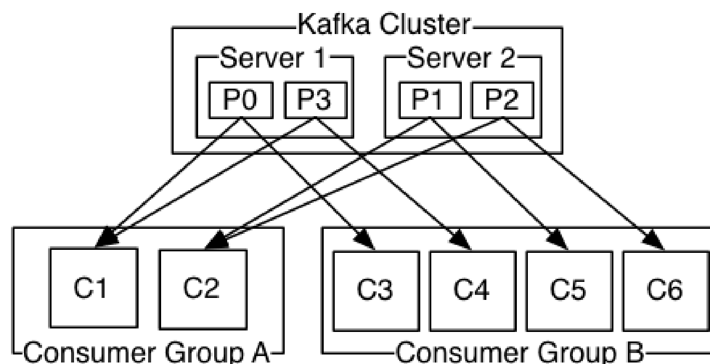


Figura 4.16: Grupos de consumidores

Como podemos ver, Kafka sería una alternativa válida a la hora de implementar la retransmisión de los datos convirtiéndose en una alternativa a Flume. Una de las principales ventajas de este sistema es la gran escalabilidad que presenta con respecto al número de productores y consumidores. En nuestro caso, tendremos un único consumidor que será nuestro sistema de procesamiento y ocho productores puesto que, como veremos en una sección posterior, el sistema de Dispensación Electrónica se encuentra formado por ocho procesos diferentes. No necesitaremos un sistema que ofrezca una gran escalabilidad en este sentido dado que tanto el número de consumidores como de productores se mantiene reducido y constante. Por otro lado, Flume nos permite configurar una arquitectura multicapa cuya primera capa se desplegará en los nodos del sistema de Dispensación Electrónica. Esta primera capa seguirá funcionando y almacenando eventos en los respectivos canales ante un fallo del clúster, lo que nos permitirá cumplir con el requisito RCA_002 que establece que ante esta situación de fallo no se deben perder datos. La posibilidad de dividir los datos en diferentes flujos (*topic)* que nos ofrece Kafka tampoco es aplicable a nuestro caso puesto que trataremos todas las líneas de log como un único flujo de datos. Todas estas circunstancias han hecho que nos decantemos por Flume a la hora de implementar esta funcionalidad.

4.2.4.1. Apache ZooKeeper

Zookeeper [18] es un sistema de coordinación de aplicaciones distribuidas que ofrece un conjunto simple de primitivas de alto nivel que permiten a las aplicaciones acceder a servicios de sincronización, configuración, mantenimiento y grupos de nombres.

Permite a los procesos distribuidos coordinarse a través de un espacio de nombres jerárquico organizado de manera similar a un sistema de archivos. Esta estructura de datos se mantiene en memoria con el fin de ofrecer alto rendimiento. Cada uno de los nodos se denomina *znode* y juegan un papel similar al de los directorios y los archivos en un sistema de archivos tradicional (figura 4.17).

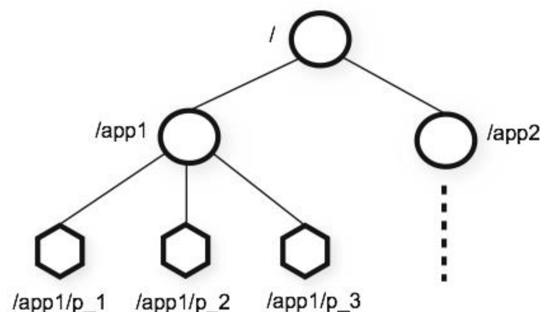


Figura 4.17: Espacio de nombres

A diferencia de los sistemas de archivos tradicionales, los *znode* mantienen un conjunto de datos asociados que pueden ser consultados y modificados por parte de las aplicaciones de forma atómica.

Además, ZooKeeper ofrece alta disponibilidad por medio de una arquitectura redundante que replica los datos a través de los nodos del clúster. Estos nodos mantienen la estructura en memoria además de una imagen de backup almacenada en disco junto a un conjunto de ficheros de log en los que se guardan las transiciones que convierten la imagen de backup al estado actual del sistema. Los clientes se conectan a un único nodo del clúster ZooKeeper, en caso de fallo en la conexión se reintenta con otro nodo diferente (figura 4.18).

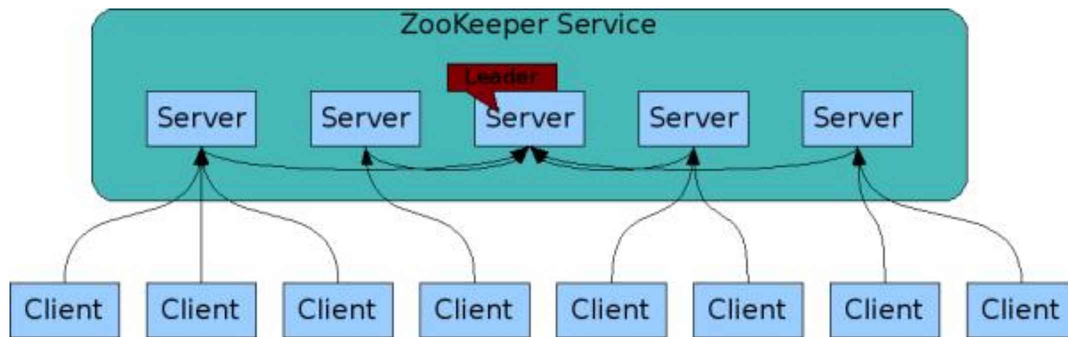


Figura 4.18: Clúster ZooKeeper

Necesitaremos contar con ZooKeeper instalado en el clúster puesto que, como veremos, HBase utiliza este servicio para la coordinación de los diferentes componentes en una arquitectura distribuida. En la implementación final utilizaremos HBase como sistema de almacenamiento con lo que tendremos que comunicarnos con este sistema a través de ZooKeeper desde nuestra aplicación de procesamiento de log.

4.2.4.2. Apache Hbase

HBase [19] es una base de datos distribuida y orientada a columnas que se ejecuta sobre HDFS y que es capaz de soportar lecturas y escrituras aleatorias en tiempo real sobre grandes conjuntos de datos. HBase se ha implementado siguiendo las publicaciones de Google: *"BigTable: A Distributed Storage System for Structured Data"* [20].

Apache Hbase se ha diseñado para ofrecer escalabilidad lineal con respecto al número de nodos del clúster. No se trata de una base de datos relacional y no soporta consultas SQL sino que ofrece un API propia de acceso a datos además de un Shell a través del cual se pueden realizar consultas con un lenguaje propio.

Los datos se almacenan en tablas, estas tablas están organizadas en filas y columnas. Cada una de las intersecciones entre las filas y las columnas se denomina celdas y se encuentran versionadas. Por defecto estas versiones se realizan en base a la fecha de inserción. En cada una de las celdas se mantiene un *array* de bytes que no se interpreta como un tipo de datos determinado. Cada fila cuenta con una clave que también se almacena como un conjunto de bytes con lo que, de forma teórica, se puede utilizar cualquier estructura de datos como clave, todas las filas de la tabla se encuentran ordenadas por su clave. Por otro lado, las columnas se encuentran organizadas formando *column families*, las familias en las que se agrupan las columnas se deben especificar en el momento de creación de la tabla, sin embargo, los miembros de cada familia pueden ser creados bajo demanda en cada una de las inserciones. En la figura 4.19 se puede ver un esquema en el que se muestra la estructura lógica de las tablas de HBase [21].

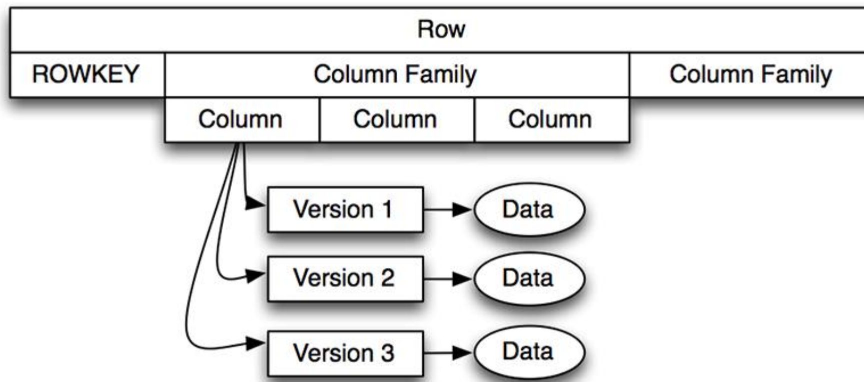


Figura 4.19: Formato de tablas en HBase

Las tablas se dividen en particiones a medida que se añaden nuevas filas, estas particiones se encuentran distribuidas entre los nodos del clúster. HBase se ha diseñado siguiendo una arquitectura maestro esclavo formada por uno o más *regionservers,* que mantienen las regiones que forman las tablas y atienden las peticiones de los clientes, y un *master,* que se encarga de repartir las regiones entre los *regionservers* y de realizar la gestión de los posibles fallos. HBase utiliza ZooKeeper para la implementación de esta arquitectura distribuida. Por último, HBase utiliza el API de acceso a datos de Hadoop para hacer los datos persistentes, con lo que el sistema de archivos más utilizado es HDFS ofreciendo alta disponibilidad y resistencia a fallos.

Surgen diversas consideraciones a la hora de diseñar una tabla en HBase. Una de las más importantes radica en el la elección de la clave puesto que determina la forma en que se ordenarán los registros y, por lo tanto, lo cerca que una fila está de otra [22]. Las filas cercanas pertenecerán a un mismo *regionserver* mientras que filas muy separadas pueden estar en nodos diferentes del clúster. Esto determina, por ejemplo, el número de consultas que se pueden atender en paralelo. Otro factor a tener en cuenta es el nombre de las columnas puesto que por cada celda se almacena la columna a la que pertenece así como la fecha de inserción, por lo tanto, se prefieren nombres cortos. Por último, también es importante prestar atención a como se almacenan los datos, como ya hemos comentado en HBase los datos se almacenan como secuencias de bytes sin ningún tipo asociado. Así, por ejemplo, una fecha se puede representar en un lenguaje como Java de al menos dos formas, como una cadena de caracteres o como un número que representa la fecha en milisegundos, sin embargo la segunda ocupa menos bytes que la primera (el tamaño de la cadena de caracteres depende del formato con el que se represente pero, en general, es muy superior al de un entero). De este modo al introducir una fecha en HBase es preferible no introducir la cadena de caracteres convertida directamente a bytes.

Como veremos en la descripción de la arquitectura final del sistema de procesamiento de log, utilizaremos HBase como sistema de almacenamiento. La principal característica que nos ha llevado a seleccionar esta herramienta ha sido el alto rendimiento que ofrece, tanto en lecturas como en escrituras aleatorias.

4.2.4.1. Apache Hive

Apache Hive [23] es una infraestructura de almacenamiento de datos construida sobre Hadoop que facilita la consulta y gestión de grandes conjuntos de datos que residen en almacenamiento distribuido. Incluye un lenguaje de consulta similar a SQL denominado HiveQL

Hive proporciona una capa de acceso a datos que permite la realización de consultas HiveQL sobre conjuntos de datos almacenados en HDFS. De este modo incluye soporte de un lenguaje estructurado similar a SQL, que está muy asentado en la industria, sobre Hadoop sin la necesidad de tener por debajo un sistema de base de datos estructurado. Además, ofrece una interfaz JDBC/ODBC con lo que ofrece una gran compatibilidad con la mayor parte de herramientas de bases de datos.

Hive utiliza un componente denominado MetaStore que se encarga de mantener todos los metadatos acerca de la estructura de las tablas y de la localización de los datos. En el momento de crear una tabla utilizando HiveQL existe la posibilidad de crear una tabla externa, esto es, se indica la localización de los ficheros que contienen los datos originales y se añade información acerca de cómo convertir los datos originales en una tabla estructurada. Podemos pensar en una línea de log en la que los campos se separan por "|", podríamos crear una tabla en Hive indicando este separador y se realizaría un mapeo directo por orden de declaración de los campos en la sentencia de creación y de aparición en la línea entre los diferentes campos de cada línea, que forman las filas de la tabla.

Un aspecto a tener en cuenta es que las consultas HiveQL se convierten en tareas MapReduce, lo que hace que la utilización de Hive sea inviable para sistemas de procesamiento en tiempo real debido a la poca idoneidad de MapReduce para este tipo de sistemas. Esta limitación de MapReduce se debe al overhead asociado a la ejecución de cada una de las tareas. Como ya hemos comentado, su ejecución supone una tarea de partición de los datos, el envío de cada uno de estos datos al nodo correspondiente (se puede conseguir una planificación en la que no sea necesario mover datos planificando exactamente cada tarea en el nodo en el que se encuentran los datos) y la ejecución de una nueva máquina virtual Java en cada uno de ellos, finalmente es necesario reordenar los datos y realizar la etapa de *reduce*. Este overhead no es un gran problema cuando se procesan grandes cantidades de datos como miles o millones de registros puesto que no supone una parte significativa sobre el tiempo total. Sin embargo, en un sistema en tiempo real en el que es necesario procesar pequeñas cantidades de datos muchas veces, a medida que se reciben, si supone un problema ya que es necesario ejecutar todas estas tareas continuamente para la pequeña cantidad de registros que se recibe en un momento dado. Existe un proyecto en marcha que busca mayor flexibilidad en Hive permitiéndole ejecutarse sobre Spark [24]. Como veremos, Spark se adapta mejor a las necesidades de un sistema de procesamiento en tiempo real con lo que, en un futuro cercano, Hive sobre Spark se puede convertir en una alternativa interesante para este tipo de sistemas. Dado que este proyecto todavía se encuentra en una fase muy temprana no se ha considerado para la realización de este Trabajo Fin de Grado.

En la primera versión del sistema hemos decidido utilizar Hive con el fin de ofrecer una herramienta de acceso a datos estructurados de cara a las aplicaciones cliente. No obstante, nos hemos encontrado con limitaciones técnicas a la hora de acceder a los datos desde la aplicación de procesamiento en tiempo real que han obligado a rediseñar todo el sistema de almacenamiento. Comentaremos estas dificultades junto a las soluciones propuestas en mayor detalle en el capítulo de diseño.

4.2.4.2. Cloudera Impala

Se trata de un motor distribuido que permite realizar consultas en tiempo real de datos almacenados en HDFS o HBase [25]. Utiliza los mismos metadatos, sintaxis SQL (HiveQL), driver ODBC e interface de usuarios que apache Hive. Las consultas con Impala son mucho más rápidas que en el caso de Hive a costa de un mayor consumo de memoria y CPU.

Cloudera Impala implementa el concepto de Massively Parallel Processing (MPP) ya comentado. El abandono de MapReduce hace que Impala sea mucho más adecuado para sistemas

con necesidades en tiempo real puesto que se eliminan las restricciones que hemos comentado en el caso de Hive y que eran debidas a la utilización de este modelo de computación.

Dado que en nuestro caso no hemos optado por la utilización de **Cloudera Distribution of Hadoop (CDH)** hemos descartado esta alternativa a la hora de seleccionar una herramienta de acceso a datos con lo que no entraremos a describirla en mayor detalle.

4.2.4.3. Apache Phoenix

Apache Phoenix [26] nace como una iniciativa que busca añadir una nueva capa sobre HBase que permita acceder a los datos simulando el comportamiento de una base de datos relacional, de este modo, se incluye un conector JDBC que permite la realización de consultas SQL sobre una base de datos HBase.

Se pueden distinguir dos formas de trabajar con Phoenix:

- **Crear tablas nuevas:** en este caso las tablas se crean en Phoenix mediante sentencias SQL convencionales y el sistema crea la tabla en HBase correspondiente de forma totalmente transparente para el usuario.
- **Mapear tablas ya existentes en HBase:** esta aproximación está pensada para aquellos casos en los que ya tenemos una tabla en HBase y queremos realizar consultas SQL sobre ella, se puede realizar un mapeo en modo *solo lectura* de tal forma que no se modifican las tablas originales presentes en HBase, o por el contrario, en modo *lectura escritura* en el que se pueden modificar las tablas originales, en concreto, se añaden todas las columnas que no estaban presentes en HBase pero que sí están en la definición de la tabla en Phoenix.

Apache Phoenix incorpora un soporte para transacciones reducido, únicamente se incluye el nivel de aislamiento *TRANSACTION_READ_COMMITTED* con lo que un dato únicamente estará disponible para su lectura cuando sea confirmado, de este modo se evitan lo que se conoce como *lecturas sucias*. Además, tampoco se ofrece soporte para sentencias relacionales de SQL que involucren varias tablas como UNION o INTERSECT.

Como ya hemos comentado, utilizaremos HBase como sistema de almacenamiento. En el requisito RDI_003 se especifica que se debe proporcionar un mecanismo de acceso al conjunto de datos originales a través de SQL. Además, en el requisito RAP_003 se recoge la necesidad de configurar *Pentaho Report Designer* de tal forma que se puedan generar nuevos informes personalizados con esta herramienta, para ello, también necesitamos proporcionar una mecanismo de acceso a datos con soporte para SQL. Utilizaremos Phoenix para cubrir estas funcionalidades.

4.2.4.4. Apache Spark

Apache Spark [27] es un sistema de procesamiento distribuido de propósito general. Incluye una serie de herramientas de alto nivel que dan soporte a procesamiento de datos estructurados (**Spark SQL**), aprendizaje automático (**MLlib**), procesamiento de grafos (**GraphX**) y procesamiento en tiempo real (**Spark Streaming**). Spark ofrece un entorno unificado sobre el que se ejecutan todas estas herramientas, y proporciona soporte para su ejecución en un clúster gestionado con YARN o Mesos o incluso en modo *Standalone* en el que utiliza su propio gestor (figura 4.20).

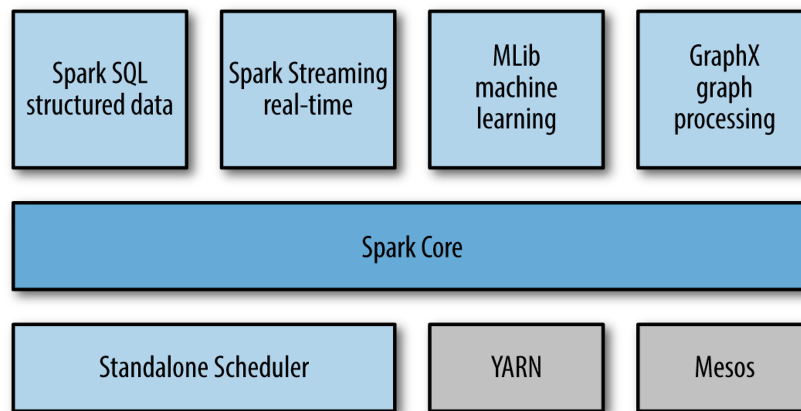


Figura 4.20: Arquitectura Spark

El concepto central dentro de Spark se denomina *Resilient Distributed Data-set* (RDD). Un RDD consiste en un conjunto inmutable de datos que se encuentra dividido en particiones que pueden ser procesados en paralelo en los diferentes nodos del clúster. Una vez se crea un RDD, bien a partir de un conjunto de datos existente (como un fichero) bien a partir de una colección de objetos dentro del lenguaje que se está utilizando, existen dos posibilidades a la hora de trabajar con el mismo [28]:

- **Transformaciones:** consisten en operaciones que a partir de uno o más RDDs de entrada devuelven un RDD de salida como resultado de realizar alguna operación sobre los datos que lo forman. Como hemos comentado, un RDD consiste en un conjunto de datos inmutable con lo que todas las transformaciones devuelven un RDD nuevo. Un ejemplo típico de transformación consiste en *map* que aplica una determinada modificación a cada uno de los elementos del RDD de entrada y crea un nuevo RDD con los resultados. También existen transformaciones que actúan sobre varios RDDs como puede ser *union* que a partir de varios RDDs de entrada devuelve un RDD de salida que contiene todos los datos de todos los RDD de entrada.
- **Acciones:** consisten en operaciones que obtienen un resultado final a partir del RDD o realizan acciones de escritura del conjunto de datos. Entre las acciones disponibles podemos encontrar *count* que devuelve el número de elementos del RDD o *reduce* que permite combinar los elementos del RDD en paralelo. Esto unido a la transformación *map* que ya hemos comentado nos permite ejecutar algoritmos *MapReduce* sobre Spark.

Como hemos visto las únicas operaciones que extraen algún resultado de los RDD son las acciones mientras que las transformaciones únicamente nos permiten generar nuevos RDDs. Spark implementa un mecanismo de *evaluación perezosa*. Esto significa que las transformaciones no se ejecutan hasta que es necesario, esto es, cuando se ejecuta una acción que recibe como entrada un RDD que se debe calcular como una sucesión de transformaciones. Esto evita que se realicen transformaciones innecesarias de las que no se extrae ningún resultado además de dar más flexibilidad a Spark a la hora de poder planificar la ejecución de las transformaciones [28].

Spark proporciona una serie de operaciones para manejar *Pair RDDs* que consisten en RDDs formados por pares clave-valor. Así, por ejemplo contamos con la transformación *mapToPair* que permite convertir un RDD en un Pair RDD especificando la función que transforma cada uno de los elementos del RDD original en un par clave-valor. Además también contamos con acciones como *reduceByKey* que permite combinar todos los elementos asociados a una clave determinada en lugar de realizar la reducción sobre todos los elementos del RDD.

En el contexto de este proyecto nos centraremos en Spark SQL y en Spark Streaming como alternativas a la hora de seleccionar una herramienta de acceso a datos y una herramienta de procesamiento en tiempo real respectivamente.

Spark SQL

Spark SQL ofrece un interface para trabajar con datos estructurados, esto son, datos que poseen un *schema*, es decir, cada uno de los registros cuenta con una serie de campos conocidos. Para manejar este tipo de datos se define un nuevo tipo de RDD denominado *SchemaRDD*, estos se pueden crear a partir de un RDD tradicional indicando el *schema* o directamente de una fuente de datos estructurada como pueden ser ficheros con datos en JSON, Parquet o HIve. Así, por ejemplo, podemos conectar Spark SQL con una instalación de Hive previamente existente pudiendo manipular los datos sin la necesidad de ejecutar consultas sobre Hive, lo que nos permite realizar un acceso a datos eficiente y adecuado para sistemas con necesidades de procesamiento en tiempo real.

Como ya hemos comentado, en un primer planteamiento, habíamos seleccionado Hive como sistema de almacenamiento. Dado que, al ejecutarse sobre MapReduce, Hive no ofrece unos tiempos de acceso asumibles para un sistema de procesamiento en tiempo real, optamos por el acceso a los datos directamente a través de Spark SQL. Como veremos en el siguiente capítulo nos hemos visto obligados a desechar este primer diseño en favor de una implementación del sistema de almacenamiento basado en HBase.

Spark Streaming

Al igual que ocurría en el caso de Spark SQL, en Spark Streaming también se define un nuevo concepto basado en la noción de RDD denominado *DStream* o *discretized streams*. Estos están pensados para el procesamiento de datos a medida que se reciben ofreciendo conectores con múltiples sistemas como Flume o Kafka [29]. Un *DStream* consiste en una secuencia de RDD cada uno de los cuales contiene los datos recibidos en una porción de tiempo determinada. Estos RDDs se procesan en intervalos de tiempo configurables por el usuario, es por esto que recibe el nombre de *discretizados.* Realmente, Spark Streaming implementa el concepto de *micro-batching*, que consiste en el procesamiento *batch* de pequeñas cantidades de datos a medida que se reciben. Como se puede ver en la figura 4.21 el flujo de datos de entrada se discretiza y se envía al motor de ejecución de Spark como si de un RDD tradicional se tratara.

Figura 4.21: Spark Streaming

Spark Streaming incorpora un nuevo conjunto de transformaciones que se pueden aplicar sobre los *DStreams*. Estas transformaciones son sensiblemente diferentes a las que hemos comentado para el caso de los RDD básicos puesto que se debe especificar la cantidad de datos del *stream* de entrada que se tienen en cuenta para cada operación. De este modo se pueden clasificar en transformaciones sin estado (***stateless***) y transformaciones con estado (***stateful***).

- **Transformaciones sin estado (stateless):** se trata de transformaciones que no tienen en cuenta los RDDs previos a la hora de procesar el actual. Esto significa que cada *batch* se procesa de forma independiente al igual que sucedía con los RDDs básicos, por lo tando, el *DStream* resultado estará formado por todos los RDD resultado de realizar la transformación sobre los RDD del *DStream* de entrada.
- **Transformaciones con estado (stateful):** al contrario que en el caso anterior, en este tipo de transformaciones se tienen en cuenta los RDDs previos. Este tipo de transformaciones necesitan la definición de *checkpoints* que permitan almacenar el estado actual del sistema con el fin de ofrecer resistencia a fallos. Dentro de este tipo de transformaciones existen dos formas de trabajar, la primera únicamente tienen en cuenta los datos recibidos en una ventana de tiempo que puede ser definida por el usuario. En la figura 4.22 se puede ver una configuración en la que se utiliza una ventana de 3 RDDs (equivalente a 3*tiempo asociado a cada *batch*) que se actualiza cada 2 RDDs, de este modo no se tiene en cuenta la totalidad de los datos recibidos sino que únicamente se hace referencia a los más recientes.

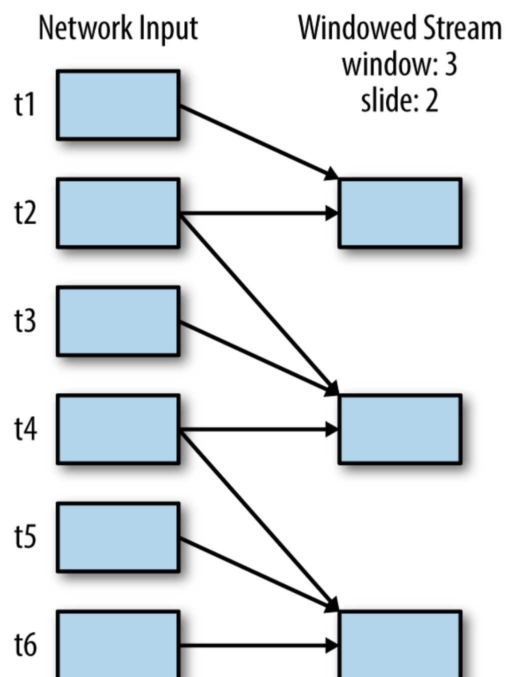


Figura 4.22: Transformaciones stateful con ventana

Por otro lado, también existe la posibilidad de mantener información acerca de la totalidad de los datos recibidos. Para ello se utilizan pares clave-valor de tal modo que el valor asociado a cada clave se actualiza a medida que se reciben nuevos datos.

Dado que Spark Streaming está pensado para el procesamiento de datos a medida que se reciben, cobran especial importancia cuestiones como la tolerancia a fallos y alta disponibilidad del sistema. En este sentido ya hemos comentado la posibilidad de crear *checkpoints* que almacenen el *stream* de datos actual con el fin de poder retomar la ejecución del programa sin pérdida de datos. Además, también se tienen en cuenta errores en los nodos del clúster que realizan el procesamiento (workers). Los datos que se reciben se almacenan con redundancia en los mismos junto a las transformaciones que se deben realizar para realizar los cálculos. Se trata de una consecuencia directa del concepto de transformación de los RDD que ya hemos comentado. De este modo ante un fallo de uno de los **workers** es posible recalcular todos los datos aplicando las transformaciones que están

almacenadas de forma redundante a los datos correspondientes. De este modo no es necesario almacenar todos los resultados intermedios de forma segura ya que no son necesarios a la hora de recuperar el sistema de un fallo.

En la implementación final utilizaremos Spark Streaming como *framework* de procesamiento en tiempo real. Hemos seleccionado esta alternativa en lugar de Apache Storm, que describiremos a continuación, pensando en la integración con Spark SQL. Además, en las ampliaciones del proyecto, también expondremos la posibilidad de implementar un sistema de clasificación automática de la actividad basado en MLlib. Por otro lado, la tolerancia a fallos que ofrece Spark y la compatibilidad con sistemas de gestión de recursos distribuidos como YARN, nos permitirá implementar un sistema robusto y de alta disponibilidad.

4.2.4.5. Apache Storm

Apache Storm [30] consiste en un famework de procesamiento en tiempo real que supone una alternativa a Spark Streaming. Storm se ejecuta en un clúster en el que se distinguen dos tipos de nodos. Por un lado tenemos un nodo maestro que ejecuta el demonio *"Nimbus"* y que se encarga de distribuir las tareas entre los nodos del clúster y realizar la gestión de los fallos. Por otro lado tenemos los *workers* que ejecutan el demonio *"Superviser"* y que se encarga de iniciar y parar las tareas que *Nimbus* asigna a cada uno de los nodos del clúster. Toda la coordinación entre *Nimbus* y todos los procesos *Superviser* se realiza a través de Zookeeper.

La abstracción principal dentro de Storm se denomina *"stream"* y consiste en un flujo de tuplas, se proporcionan dos primitivas para trabajar con *streams*: "*spout"* y *"bolt"*.

- **Spout:** representa el origen de cada *stream*, un ejemplo concreto de un *spout* puede ser un componente que se conecte al API de Twitter y emite un *stream* de tweets.
- **Bolt:** consumen un número de determinado de *streams* y pueden emitir nuevos *streams* de tal forma que el procesamiento se realice en varias fases. Estos componentes pueden realizar cualquier tipo de modificación sobre los datos de entrada desde ejecutar una determinada función sobre ellos hasta conectarse con una base de datos.

Tanto los *bolts* como los *spouts* se agrupan formando redes que se denominan "*topology"* y que consisten en un grafo en el que se definen todas las transformaciones que reciben los *streams* dentro de Storm. En la figura 4.23 se puede ver un ejemplo en el que tenemos dos *spouts* y cuatro *bolts*:

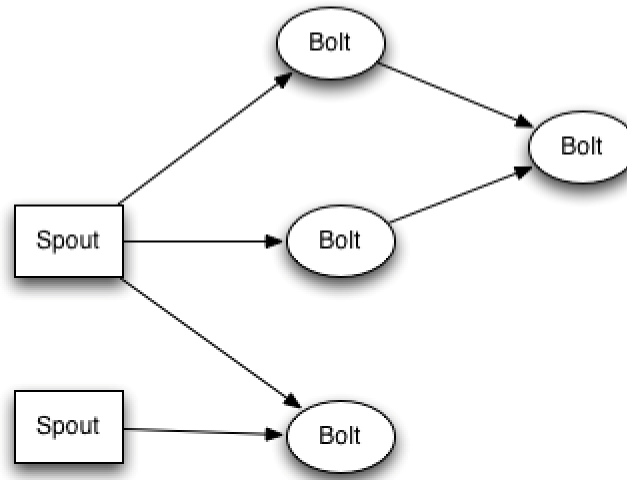


Figura 4.23: Topology Storm

Storm se encarga de distribuir los datos de los *streams* de entrada de tal forma que se consiga el mayor grado de paralelismo a la hora de ejecutar la *topology* en el clúster.

4.2.4.6. Pentaho Reporting

Se trata de una herramienta gratuita que forma parte de la edición *community* de Pentaho BI Suite, un conjunto de utilidades dedicadas al mundo del *Business Intelligent*. Pentaho Reporting [31] está compuesto por tres elementos principales:

- **Report Designer:** se trata de una herramienta gráfica que nos permite crear definiciones de informes realizados de forma totalmente personalizada por el usuario con posibilidad de acceso a una gran variedad de fuentes de datos.
- **Reporting Engine:** se trata del núcleo del sistema, es la parte encarga de ejecutar las definiciones creadas con la herramienta anterior y crear los informes a partir de los datos correspondientes.
- **Reporting SDK:** nos permite embeber los informes en nuestras aplicaciones, de este modo podemos cargar una definición hecha desde la herramienta Report Designer e incluir el informe generado en la vista de nuestra aplicación. Utilizaremos este flujo de trabajo para la creación y publicación de informes en nuestra aplicación web.

Además de crear la definición de lo informes que estarán disponibles en la aplicación web, proporcionaremos un mecanismo de generación de informes personalizados cumpliendo con el requisito RAP_003.

4.3. Sistema de Dispensación Electrónica

Nuestro proyecto se enfoca en el desarrollo de una herramienta que pueda analizar los log de actividad de un Sistema de Receta Electrónica en tiempo real, en concreto, nos centramos en el sistema de Dispensación Electrónica (DISEL) que se encuentra actualmente implantado en Galicia. Por lo tanto, entender el funcionamiento de este sistema es clave a la hora de poder desarrollar nuestro proyecto.

4.3.1. Arquitectura

El sistema de Dispensación Electrónica está desplegado en cuatro nodos, en cada uno de estos nodos se ejecutan dos instancias, cada una en un servidor de aplicaciones diferente. De este modo tendremos cuatro nodos (ocho procesos) enviando datos de forma simultánea a nuestro clúster para ser analizados, esto condicionará el diseño de nuestro sistema de recepción de datos tal y como veremos más adelante.

En todo caso, los datos se verán por parte del analizador como un único conjunto sin entrar a valorar de que nodo provienen o de que proceso dentro de ese nodo.

4.3.2. Formato de los logs

En general, por cada operación se generan tres líneas de log, una en el momento en el que se recibe la petición, otra con datos que puedan ser relevantes acerca de la operación y otra en el momento en el que se emite la respuesta. Cabe destacar que, dado que se atienden múltiples peticiones de forma simultánea, estas tres líneas asociadas a una operación no tienen por qué generarse de forma consecutiva sino que se pueden intercalar con otras líneas asociadas a otras operaciones. Por lo tanto, con el fin de permitir identificar a que operación hace referencia cada una de las líneas se incorpora un identificador único por cada operación como uno de los campos que aparece en todas las líneas, independientemente del tipo que sea.

A continuación definiremos el patrón que sigue cada uno de los tipos de líneas de log generados, las palabras que aparecen en *cursiva* se refieren a cadenas que aparecen directamente en las líneas de log originales mientras las demás se utilizan simplemente para poder hacer referencia al campo en las descripciones.

4.3.2.1. Peticiones

Fecha | Nivel | *plataforma* | id | *Petición* | *Servicio*: servicio | *Operación*: operación | *Cliente*: cliente

- **Fecha:** contiene la fecha de generación de la línea por parte del sistema de Dispensación Electrónica en el formato "AAAA-MM-dd HH:mm:ss,SSS", por ejemplo: "*2015-04-29 22:05:09,214*".
- **Nivel:** el nivel de prioridad con el que se genera la línea y que, para los tres tipos de línea que estamos manejando se utiliza *INFO*. En los datos que manejamos para la realización de este Trabajo Fin de Grado no contamos con líneas con un nivel diferente como puede ser un error.
- ***plataforma:*** este campo con el valor *"plataforma"* indica que se trata de peticiones recibidas por el Sistema de Dispensación Electrónica. También puede tener el valor *"recursos"* en aquellas líneas que se registran como consecuencia de una petición realizada por el Sistema de Dispensación Electrónica a otros sistemas. En los datos que manejamos en este Trabajo Fin de Grado únicamente manejamos líneas con el valor "*plataforma".*
- **Id:** como ya hemos comentado necesitamos un identificador único para cada operación para poder agrupar todas las líneas que hacen referencia a dicha operación.
- ***Petición:*** este campo tiene siempre la cadena de caracteres "Petición" y nos permite identificar que la línea se ha generado porque se ha recibido una petición.

- ***Servicio:*** **servicio**: este campo recoge información acerca del servicio que está atendiendo la petición dentro del sistema de Dispensación Electrónica. Un posible valor para este campo es: "*Servicio: ServicioDispensaciones*"
- ***Operación:*** **operación:** permite identificar el tipo de operación que se está ejecutando, este campo se relaciona en cierto modo con el anterior puesto que cada servicio atiende un determinado conjunto de tipos de operación. Un posible valor sería: "*Operación: registroDispensacion_eReceita*".
- ***Cliente:*** **cliente:** este campo nos permite identificar el sistema que está realizando la petición puesto que el sistema de Dispensación Electrónica se interconecta con múltiples sistema en una arquitectura de nivel superior compleja y que no discutiremos en este documento puesto que se sale de los intereses de nuestro proyecto. Un posible valor sería: "*Cliente: NCD*".

4.3.2.2. Datos

Fecha | Nivel | *plataforma* | id | <datos><farmacia> </farmacia><farmaceutico> </farmaceutico><tipodocumento>CIP</tipodocumento><documento> </documento><fuente> </fuente></datos>

Los cuatro primeros campos contienen la misma información que el caso anterior. En este tipo de línea, a estos primeros campos comunes, se le añade un campo que contiene información extra acerca de los datos asociados a la operación en formato XML. Cabe destacar que no todos los campos de este XML tienen porque tener datos para todas las operaciones sino que pueden aparecer campos vacíos.

- **farmacia:** identificador de la farmacia que origina la ejecución de la operación.
- **farmacéutico:** identificador del profesional que ejecuta la operación.
- **tipoDocumento:** tipo de documento por el que se identifica al paciente y cuyo valor se recoge en el campo *documento*.
- **documento:** identificador del paciente, en el formato adecuado para el documento especificado en el campo anterior.
- **fuente:** el origen del que se han obtenido los datos de identificación del paciente (por ejemplo, a través de la banda magnética de la tarjeta sanitaria o de forma manual).

4.3.2.3. Respuestas

Fecha | Nivel | *plataforma* | id | *Respuesta* | *Servicio*: servicio | *Operación*: operación | *Cliente*: cliente | Tamaño | Tiempo

La información recogida en este tipo de línea es idéntica a la recogida en las líneas de petición a excepción de los dos últimos campos y del quinto campo que recoge el literal *Respuesta*.

- **Tamaño:** este campo recoge el tamaño de la respuesta que se ha generado en bytes.
- **Tiempo:** indica el tiempo que se ha tardado en realizar la operación.

Es habitual que únicamente aparezcan las líneas de petición y respuesta y que no se incluya la línea de datos. A continuación podemos ver un ejemplo de una operación que incluye las tres:

2015-04-29 22:05:09,214 | INFO | plataforma | [18c82e855a11c46e7d71b425f6abeab84c753bf7] | Petición | Servicio: ServicioDispensaciones | Operación: registroDispensacion_eReceita | Cliente: NCD

2015-04-29 22:05:10,792 | INFO | plataforma | [18c82e855a11c46e7d71b425f6abeab84c753bf7] | <datos><farmacia>FC00000</farmacia><farmaceutico>FT00000</farmaceutico><tipodocumento>CIP</tipodocumento><documento>DOC00000</documento><fuente>MANUAL</fuente></datos>

2015-04-29 22:05:21,072 | INFO | plataforma | [18c82e855a11c46e7d71b425f6abeab84c753bf7] | Respuesta | Servicio: ServicioDispensaciones | Operación: registroDispensacion_eReceita | Cliente: NCD | 5253 | 11859

Para estimar el volumen de datos, hemos analizado los logs de Dispensación Electrónica de un día y nos encontramos con **501.649 líneas con datos**, **731.845 líneas asociadas a Peticiones** y **716.899 líneas asociadas a Respuestas** lo que supone un total de **1.950.393**. Podemos asumir un volumen total cercano a los **2.000.000 líneas/día**, únicamente para estos tres tipos de líneas que son las únicas que manejaremos en este Trabajo Fin de Grado. Si nos centramos en los datos, para el mismo día, obtenemos un total de **125.547 documentos de paciente diferentes**, **2.897 farmacéuticos diferentes** y **1.337 farmacias diferentes.** Estos datos pueden dar una idea de la magnitud de los ficheros de log generados y de la necesidad de una herramienta que permita automatizar, en la medida de lo posible, el análisis de estos datos ante la imposibilidad de abarcarlos por completo de forma manual.

4.4. Virtualización

En lo que se refiere a la tecnología de virtualización no se ha llevado a cabo una selección tecnológica como tal puesto que en el requisito RAP_004 se establece como sistema de virtualización Docker con el fin de explorar las ventajas y desventajas que puede ofrecer. A continuación realizaremos una descripción detallada de esta tecnología.

4.4.1. Docker

Docker [32] ofrece un sistema de virtualización basado en contenedores que utiliza una serie de herramientas propias del kernel de Linux para crear entornos de ejecución independientes con un overhead muy reducido en comparación con otros sistemas de virtualización.

La idea que está detrás de la mayor parte de las tecnologías de virtualización consiste en crear una capa de abstracción sobre la que se ejecuta una nueva máquina de forma aislada. Por el contrario, los contenedores consisten en crear entornos de ejecución aislados sobre el mismo sistema operativo, no se virtualiza una máquina completa. En la figura 4.24 se puede ver esto gráficamente:

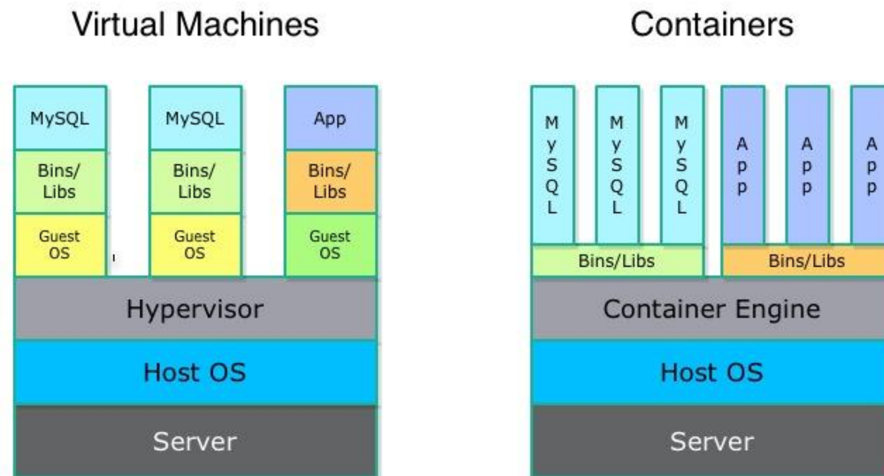


Figura 4.24: Contenedores frente a máquinas virtuales

Como se puede apreciar los contenedores comparten todos aquellos recursos que son comunes incluyendo el kernel del sistema operativo y las dependencias de las aplicaciones que se ejecutan en ellos. En el caso de las máquinas virtuales tradicionales, se opta por crear un nuevo sistema operativo que se ejecuta sobre la capa de virtualización y en el que se instalan todas las dependencias necesarias. Esto convierte a los contenedores en una alternativa más eficiente en lo que al uso de recursos se refiere.

No obstante, los contenedores se convierten en una opción menos flexible puesto que el sistema que se ejecuta dentro de cada uno de ellos debe ser el mismo, o del mismo tipo, que el sistema anfitrión. De este modo podemos ejecutar una distribución Linux dentro de un contenedor que se despliega sobre una máquina Linux, pero no podemos ejecutarlo, por ejemplo, sobre un sistema Windows.

Como ya hemos comentado Docker utiliza una serie de características del kernel de Linux que le permiten crear contendores aislados, cada uno de los cuales cuenta con su propio entorno de red, gestor de recursos y almacenamiento. Esto permite la ejecución de múltiples contenedores en un mismo equipo sin que puedan llegar a interferir entre ellos. Cabe destacar que Docker también está disponible para otros sistemas además de Linux tales como Windows o OS X. En estos casos, los contenedores se despliegan dentro de una máquina virtual ligera que ofrece todas las funcionalidades necesarias del kernel de Linux. Esta aproximación es más ineficiente puesto que supone la inclusión de una nueva capa de virtualización que es justo lo que queremos evitar con el uso de contenedores. En todo caso, únicamente se crea una máquina virtual en cada host sobre la que se ejecutan todos los contenedores. Además se incluye un cliente que permite manejar estos contenedores desde la línea de comandos del sistema anfitrión de forma transparente tal y como se muestra en la figura 4.25.

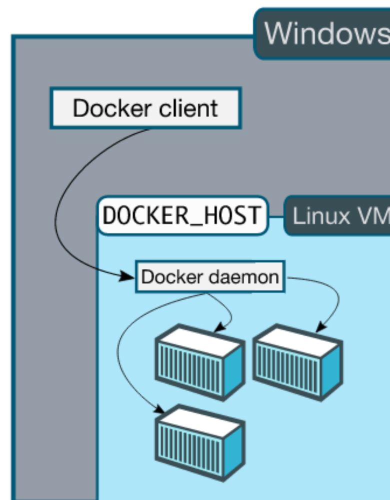


Figura 4.25: Docker en Windows

En sistemas Linux no es necesaria la presencia de esta máquina virtual, pero también se sigue una arquitectura cliente servidor en la que el cliente habitualmente se instala en el mismo host que el demonio encargado de gestionar los contenedores, no obstante, se puede instalar el cliente en un host remoto.

Existen tres conceptos fundamentales dentro de Docker, las imágenes, los registros y los contenedores [33].

4.4.1.1. Imágenes

Se trata de los elementos a partir de los que se crean los contenedores. Consisten en ficheros de definición en los que se especifican las acciones que se deben llevar a cabo para realizar la configuración base a todos los contenedores. Entre estas acciones se encuentran añadir ficheros, ejecutar un comando, abrir puertos o definir variables de entorno. Además, se permite definir una imagen partiendo de otra ya existente. De este modo para la creación de la imagen resultante se ejecutarán todas las acciones de las imágenes que forman la cadena. A continuación se puede ver un ejemplo sencillo:

```
FROM fedora
RUN yum install -y gcc
ADD ./myprogramfiles.tar /tmp
```

Código 4.1

En este caso se crea una imagen a partir de la imagen denominada “*fedora*”. La palabra clave RUN indica que se debe ejecutar un comando, en este caso se instala gcc. Por último se añade una archivo, que se encuentra en el directorio en el que se encuentra el fichero de definición y que en los contenedores que se creen a partir de esta imagen se encontrará en “/tmp”.

Una vez se crea la imagen a partir de estos ficheros, se mantiene almacenada en cache de tal forma que cuando se inicia un contenedor se coge dicha imagen como la configuración base.

4.4.1.2. Registros

Las imágenes creadas se organizan en registros que pueden ser públicos o privados. La comunidad de Docker mantiene un registro público denominado Docker Hub en el que están disponibles un gran número de imágenes que se puede utilizar como base para nuevas imágenes o pueden ser utilizadas directamente para crear nuevos contenedores.

4.4.1.3. Contenedores

Se trata del elemento central, representa un espacio de ejecución independiente en el que se pueden instalar las aplicaciones o servicios que deseamos ejecutar de forma aislada. En cada uno de estos contenedores ejecutaremos uno de los nodos del clúster Hadoop. Además de los nodos del clúster, también crearemos contenedores para desplegar tanto la aplicación que simula el funcionamiento del sistema de Dispensación Electrónica como la aplicación web de generación informes.

Los entornos de ejecución además de estar aislados entre sí también están totalmente aislados del sistema anfitrión. Además, el contenido de estos contenedores no condiciona de ninguna manera el funcionamiento de Docker.

5. Diseño

En este capítulo describiremos la arquitectura general del sistema, pasando a continuación a proponer un diseño más concreto acerca del sistema de procesamiento en tiempo real. En este apartado indicaremos las herramientas seleccionadas para la implementación del sistema final. Por último, introduciremos el diseño de la aplicación Web que pone los informes a disposición de los usuarios y de la aplicación que simula el funcionamiento del sistema de Dispensación Electrónica a partir de los ficheros de log de los que disponemos.

5.1. Arquitectura general del sistema

A continuación describiremos la arquitectura general del sistema en la que se puede ver cómo interactúan los diferentes componentes a un alto nivel de detalle. En la figura 5.1 vemos como la comunicación entre el sistema de Dispensación Electrónica y el sistema de procesamiento de log se realiza a través de log4j tal y como se especifica en el requisito RAP_001. El sistema de procesamiento de log consistirá en un clúster Hadoop (requisito RAP_002). Describiremos este sistema en detalle más adelante. El requisito RDI_001 establece que se debe implementar una aplicación web que contenga una serie de informes predefinidos, esta aplicación obtendrá los datos del clúster Hadoop a través de una conexión JDBC. Por último implementaremos un sistema de monitorización del sistema de procesamiento en tiempo real que nos permita cumplir con lo establecido en los requisitos RDI_007 y RDI_008 que indican que se deben generar informes acerca del estado del sistema con el fin de validar su correcto funcionamiento a los efectos de la fase de pruebas del proyecto. Este sistema de monitorización realizará peticiones HTTP tal y como comentaremos en una sección posterior dedicada exclusivamente a este componente.

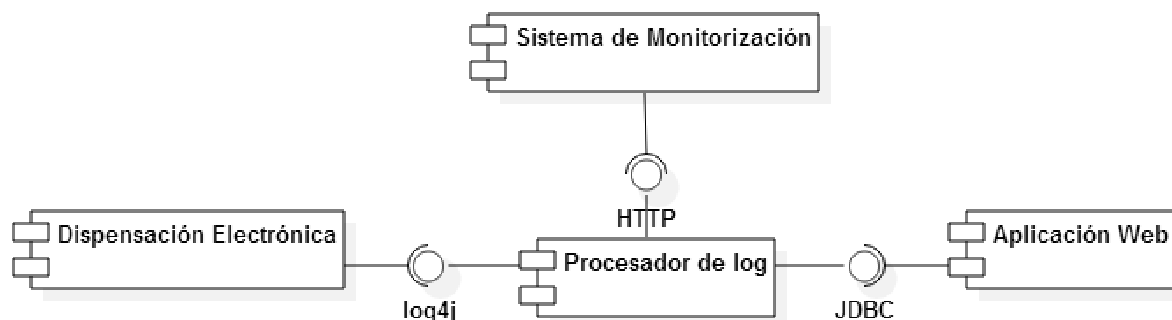


Figura 5.1: Arquitectura general del sistema

5.2. Arquitectura del sistema de procesamiento de log

En esta sección presentaremos la arquitectura general del sistema de procesamiento de log. En primer lugar, discutiremos la arquitectura lambda [34], un modelo de desarrollo de aplicaciones Big Data que combina procesamiento en tiempo real [35] con procesamiento *batch* para ofrecer un sistema robusto y de baja latencia. La arquitectura Lambda ofrece una solución muy general con aplicación potencial en un gran número de escenarios. Propondremos una solución alternativa, mucho más específica, pensando en las características que definen nuestro sistema.

5.2.1. Arquitectura Lambda

La arquitectura Lambda fue diseñada por Nathan Marz, un experimentado desarrollador de aplicaciones Big Data con experiencia en empresas como *Backtype* o *Twitter*. Se trata de una solución

de propósito general, con aplicación en una gran variedad de escenarios ofreciendo una solución robusta, escalable y de baja latencia.

Esta arquitectura se utiliza para implementar sistemas en los que necesitamos aplicar una serie de funciones sobre un gran conjunto de datos en tiempo real. Idealmente, podríamos modelar el problema con una aproximación del tipo: *consulta = función(todos los datos)*. En la práctica, esto es inabordable por la potencia de cómputo necesaria para poder procesar enormes cantidades de datos en cada consulta. Nathan Marz propone mantener una serie de vistas en las que se mantienen los resultados preprocesados, de este modo, las consultas se pueden realizar directamente sobre estas vistas eliminando la necesidad de realizar todo el procesamiento en cada consulta. Para lograr esto, se divide el sistema en tres capas: capa de procesamiento *batch*, capa de velocidad y la capa de servicio.

5.2.1.1. Capa de procesamiento *batch*

Esta capa se encarga de mantener el conjunto de datos maestro, una estructura en la que se mantienen todos los datos inmutables y a la que únicamente se añaden nuevos registros. Además, en esta capa, se procesan todos estos datos de forma periódica con el fin de ofrecer un conjunto de vistas que contenga los resultados preprocesados. Es importante tener acceso a la totalidad de los datos originales para poder generar cualquier vista arbitraria. De este modo, se consigue un sistema extensible en el que añadir una nueva vista no supone más que añadir la nueva funcionalidad al sistema de procesamiento, obteniéndose en la siguiente ejecución una nueva vista con los nuevos resultados.

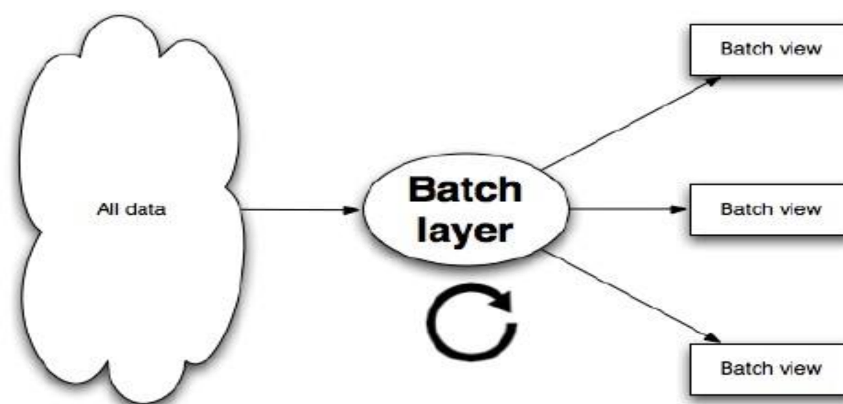


Figura 5.2: Capa de procesamiento batch

5.2.1.2. Capa de servicio

Se encarga de indexar las vistas producidas por la capa de procesamiento *batch*. Básicamente, consiste en una base de datos sobre la que los clientes pueden realizar consultas y obtener los resultados preprocesados de forma eficiente. La frecuencia con la que se actualizan las vistas de la capa de servicio depende de la frecuencia con la que la capa *batch* procesa los datos originales. Estos periodos de actualización suelen ser de varias horas o incluso días dependiendo de cada caso. No es

necesaria una base de datos que ofrezca una latencia reducida en escrituras aleatorias lo que puede simplificar el funcionamiento del sistema y su implementación.

Un ejemplo de implementación para esta capa pueden ser tablas externas en Hive que hacen referencia a los ficheros que contienen los datos procesados que ha producido la capa *batch.* En esta capa se simplifica el acceso a los datos por parte de los clientes y se consigue aislarlos de la implementación de la capa *batch*.

5.2.1.3. Capa de velocidad

Con las dos capas anteriores, tenemos un sistema en el que los resultados no están disponibles hasta que termina la siguiente ejecución. Dado que este periodo suele ser de varias horas, los resultados no tienen en cuenta los datos más recientes. En esta capa se producen las vistas con los resultados relativos a los últimos datos recibidos, que no se han tenido en cuenta en la capa de procesamiento *batch*.

La principal diferencia entre esta capa y la capa de procesamiento *batch* radica en que, en este caso, las vistas se actualizan a medida que se reciben nuevos datos, mientras que en el caso de procesamiento *batch*, los resultados se obtienen tras procesar el total de los datos. Esto supone que, en este caso, necesitamos un medio de almacenamiento que ofrezca baja latencia, tanto en escrituras, como en lecturas aleatorias, ya que se están realizando continuas actualizaciones sobre los resultados finales.

Por último, necesitamos un componente capaz de agregar los resultados obtenidos de las vistas producidas en tiempo real y de las vistas producidas en *batch* (figura 5.3).

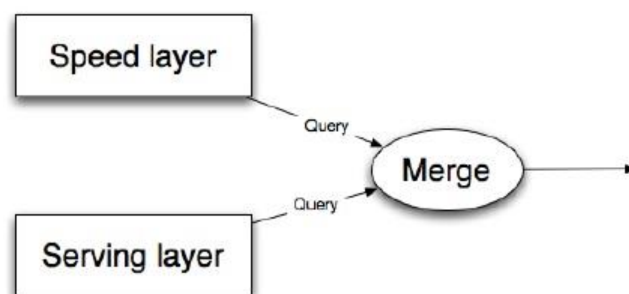


Figura 5.3: Capa de velocidad

En la figura 5.4 se puede ver un diagrama en el que se muestran los diferentes componentes que dan lugar a un sistema implementado siguiendo las directrices de la arquitectura Lambda.

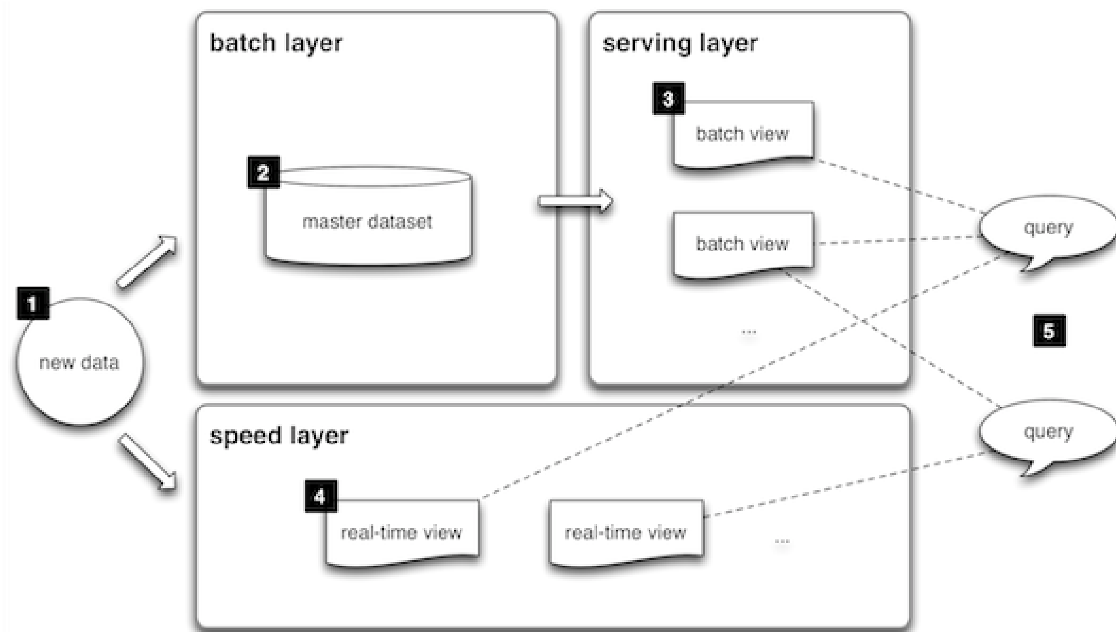


Figura 5.4: arquitectura lambda

1- Los datos se entregan a medida que se reciben tanto a la capa velocidad como a la capa de procesamiento *batch*.
2- La capa de procesamiento *batch* mantiene el conjunto de datos maestro y lo procesa periódicamente para obtener las vistas.
3- Las vistas producidas por la capa de procesamiento *batch* se ponen a disposición de los clientes a través de la capa de servicio.
4- La capa de velocidad produce vistas actualizadas con los datos más recientes.
5- Las consultas pueden ser respondidas mediante los datos preprocesados que se mantiene en ambos tipos de vista.

5.2.1.4. Ejemplo de implementación

A continuación mostraremos un ejemplo práctico en el que se proponen una serie de tecnologías para dar soporte a las diferentes capas de la arquitectura lambda. Además, discutiremos la idoneidad de esta arquitectura para la implementación de nuestro sistema.

Como se puede ver en la figura 5.5, se ha seleccionado como software de procesamiento en tiempo real Apache Storm, se trata de una elección habitual a la hora de implementar un sistema siguiendo la arquitectura Lambda. Storm fue creado por el propio Nathan Marz y es la tecnología recomendada para implementar la capa de velocidad. En la capa *batch* se ha seleccionado Hadoop, HDFS como sistema de almacenamiento y MapReduce como framework de procesamiento. En este caso, se utilizan como sistemas de bases de datos HBase para almacenar las vistas en tiempo real e Impala para mostrar las vistas procesadas en *batch* y los datos agregados.

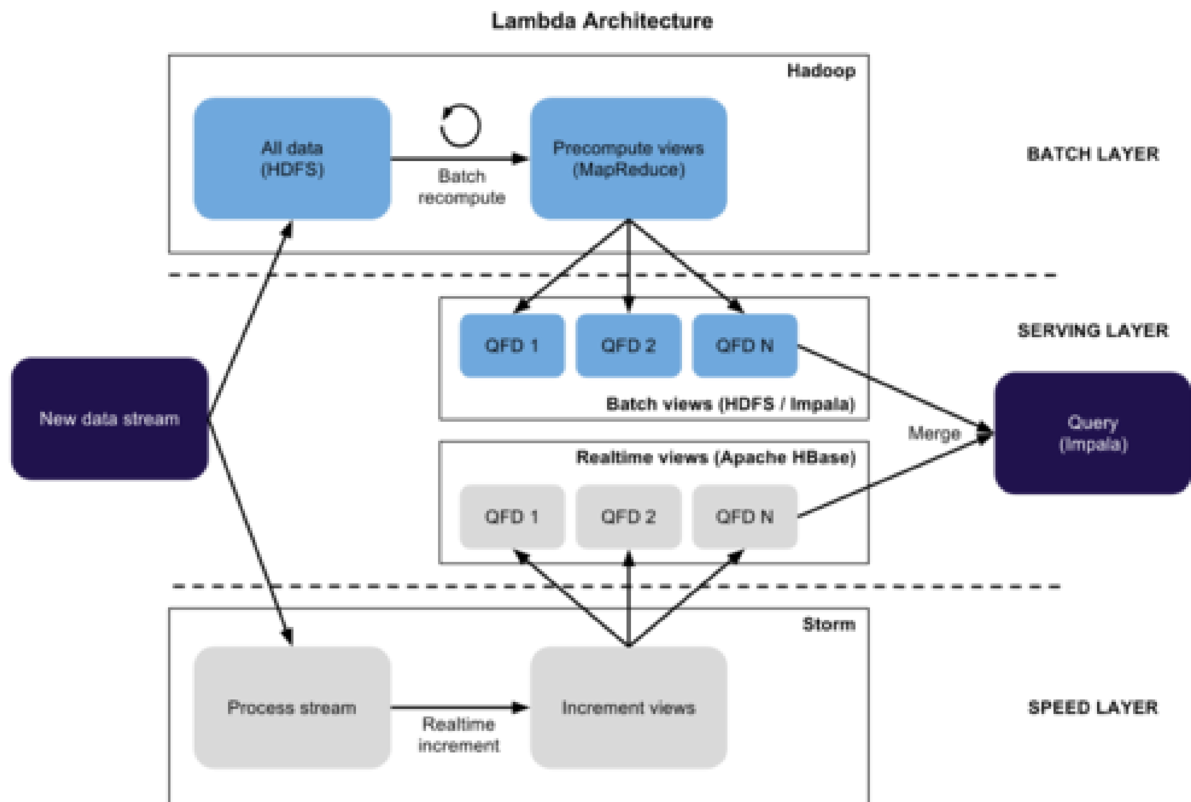


Figura 5.5: Implementación arquitectura lambda

Se puede considerar esta implementación como un estándar a la hora de implementar una arquitectura Lambda. Los puntos en los que nos podemos encontrar con alguna diferencia pueden ser los sistemas de base de datos seleccionados y, en los sistemas más recientes, la inclusión de Spark como framework de procesamiento, sobre todo, en la capa de velocidad con Spark Streaming.

Uno de los principios básicos de esta arquitectura radica en el nivel de compromiso entre velocidad de procesamiento y tolerancia a fallos. Por un lado, el uso de HDFS para almacenar el conjunto completo de los datos aporta robustez a costa de perder velocidad y supone la utilización de computación *batch* en lugar de en tiempo real. Por otro lado, tenemos la capa de velocidad que complementa los resultados de la capa *batch* con procesamiento en tiempo real, este punto del sistema se presupone más propenso a la pérdida de datos. No obstante, esto implica tener la implementación de la lógica de procesamiento replicada, una implementación en MapReduce (o cualquier otro framework de procesamiento *batch*) y otra implementación en Storm (o cualquier otro framework de procesamiento en tiempo real) lo que puede suponer un aumento considerable en los costes, tanto de desarrollo como de mantenimiento del sistema.

Han surgido soluciones que pretenden eliminar el problema de la duplicidad de código en la arquitectura lambda, una de ellas es Summingbird [36]. Se trata de un proyecto creado por Twitter que añade una nueva capa de abstracción permitiendo ejecutar los programas que utilicen este framework en un gran número de plataformas de procesamiento *batch,* como MapReduce, y de procesamiento en tiempo real, como Apache Storm.

Como consecuencia de tener estas dos capas de procesamiento aisladas, debemos considerar la complejidad asociada a mantener ambos sistemas coordinados. En la mayor parte de los casos, nuestro modelo de procesamiento será similar al que se muestra en la figura 5.6:

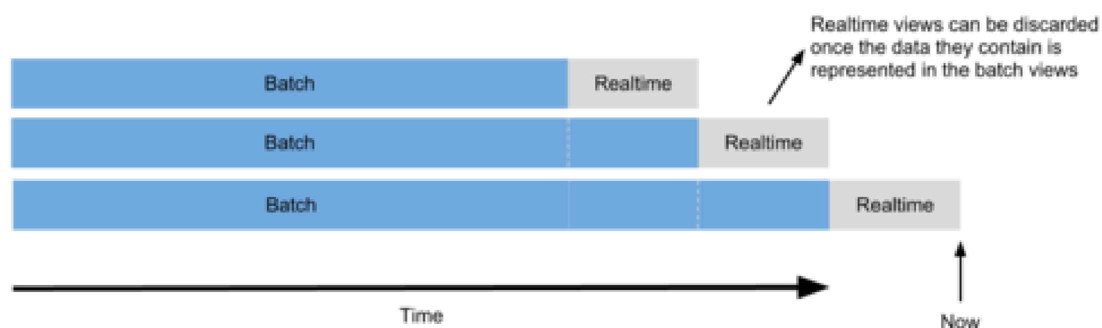


Figura 5.6: Coordinación entre capas de procesamiento

En las vistas en tiempo real únicamente se deben mantener los resultados asociados a los datos más recientes y que no se han tenido en cuenta en la iteración anterior en la capa *batch*. Una vez se realiza un nuevo procesamiento, las vistas en tiempo real deben ser descartadas, pues los resultados ya están presentes en la capa de servicio. La presencia de datos en ambas vistas puede suponer la entrega de resultados inconsistentes al cliente.

Actualmente, existen sistemas de procesamiento en tiempo real, como Spark Streaming, lo suficientemente maduros como para poder prescindir de la capa de procesamiento *batch* en la mayor parte de los casos. Spark en general, y Spark Streaming en particular, incorporan mecanismo de tolerancia a fallos como ya hemos comentado.

La división en dos capas de procesamiento propuesta por la arquitectura Lambda, puede tener sentido en algunos sistemas complejos en los que el procesamiento en tiempo real se realiza siguiendo estrategias diferentes a las utilizadas en la capa *batch*. Esto se puede dar en algunos escenarios en los que, por ejemplo, para obtener los resultados en la capa de velocidad se descartan algunos datos complejos que ralentizarían el procesamiento sin aportar un gran valor a los resultados recientes. Sin embargo, estos datos complejos pueden tener un gran valor para obtener resultados históricos con lo que si se incluyen en la capa de procesamiento *batch*. Además del valor que puedan tener los datos en unas vistas u otras, la capa *batch* no tiene unas restricciones temporales tan exigentes como la capa de velocidad con lo que no se vería afectada por sistemas de procesamiento muy complejos y pesados.

Otro principio presente en esta arquitectura radica en prestar especial atención al reprocesado de los datos. En cada procesamiento realizado en la capa de *batch* se tienen en cuenta la totalidad de los datos. El reprocesado cobra especial importancia a la hora de hacer modificaciones en el software, bien por la detección de un error en el sistema, bien por un cambio en los requisitos. Si no se tiene esto en cuenta, se puede dar el caso en el que los datos que se están sirviendo a los clientes no sean coherentes con la lógica de procesamiento, ya que han sido calculados con una versión anterior del sistema, lo que es especialmente importante si la versión anterior tenía algún error. No obstante, realizar un reprocesado en cada iteración de la capa *batch* puede no ser necesario, en el caso de la arquitectura Lambda se incluye, entre otras cosas, pensando en descartar los datos de la capa de velocidad por considerar que se trata de un componente poco robusto. [37]

5.2.2. Arquitectura alternativa

Para la implementación de nuestro sistema definiremos una arquitectura en la que unificaremos las dos capas de procesamiento en una única capa en la que utilizaremos Spark Streaming. Definiremos una capa de almacenamiento que sustituirá a la capa de servicio y que se encargará de almacenar los resultados y el conjunto de datos maestro, que antes teníamos en la capa

batch. Añadiremos una tercera capa de recolección en la que se incluyen sistemas de prevención contra la pérdida de datos y en la que se agregan todos los datos provenientes de múltiples clientes.

En la figura 5.7 se puede ver un esquema general de la arquitectura así como las tecnologías propuestas para su implementación:

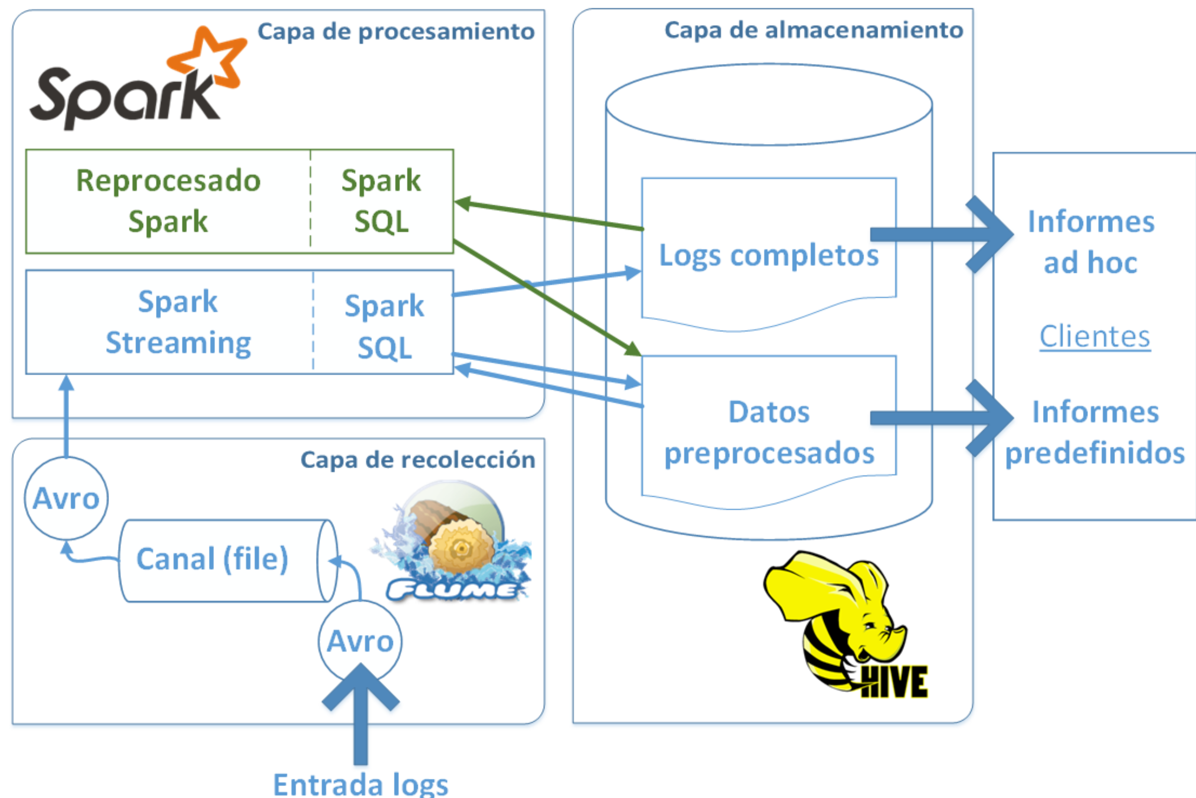


Figura 5.7: Arquitectura alternativa inicial

5.2.2.1. Capa de recolección

Esta capa funciona como único punto de entrada de datos al sistema. En esta arquitectura esta capa cobra especial relevancia con respecto a la arquitectura Lambda en la que únicamente se definía un proceso encargado de copiar datos a la capa *batch* y a la capa de velocidad. En este caso, hemos optado por un sistema en el que únicamente se realiza procesamiento en tiempo real con lo que necesitaremos una capa capaz de almacenar datos en caso de que se produzca un error en la capa de procesamiento o existan picos en los que el ratio de líneas de log procesadas sea inferior al de líneas recibidas. Por lo tanto, esta capa juega un papel muy importante a la hora de unificar las capas de procesamiento y así poder reducir los problemas, tanto de costes de implementación y mantenimiento como de coordinación de los dos sistemas, que ya hemos comentado en la arquitectura Lambda.

Para la implementación de esta capa hemos seleccionado Apache Flume, Desplegaremos un agente en cada uno de los nodos en los que está desplegado el sistema de Dispensación Electrónica y otro en nuestro clúster que agregará todos los logs recibidos en un único flujo de datos. en la figura 5.8 se puede ver el funcionamiento de este diseño multicapa de agentes Flume para dos nodos:

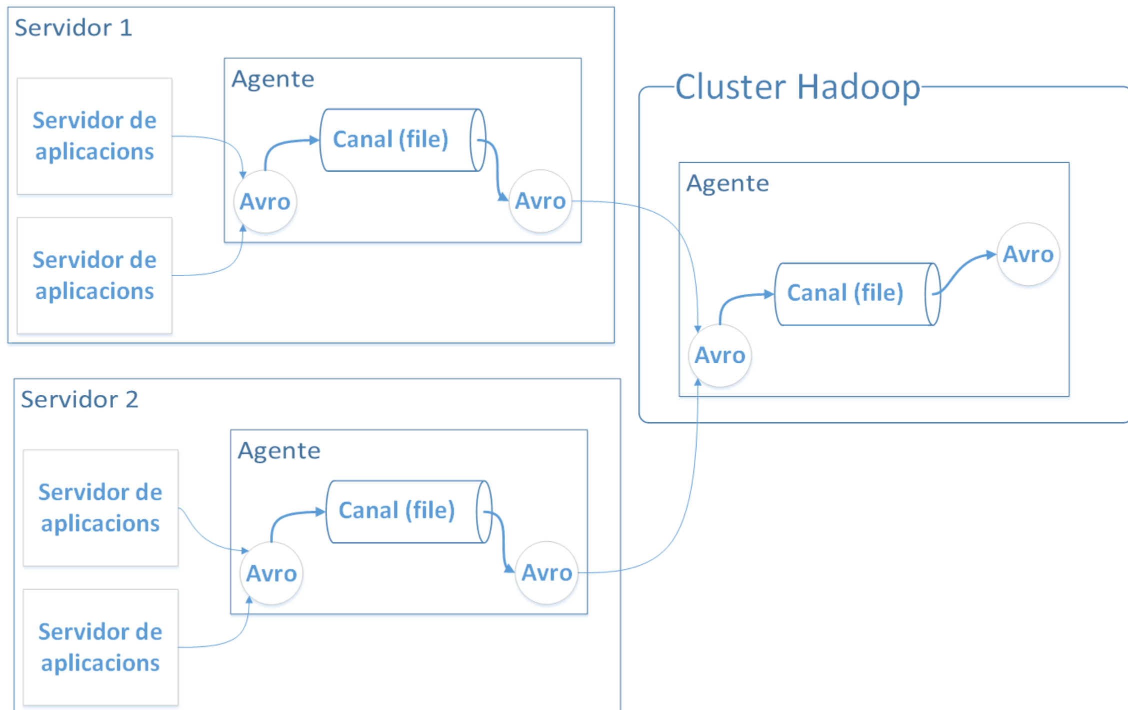


Figura 5.8: Capa de recolección

Este diseño multicapa de agentes Flume proporciona tolerancia a fallos y nos permite desacoplar el sistema de procesamiento de log del funcionamiento del sistema de Dispensación Electrónica. En caso de fallo en el clúster Hadoop no se perderán datos ya que, en cada nodo del sistema de Dispensación Electrónica, los datos permanecerán en el canal hasta que puedan ser reenviados. Al atribuir esta responsabilidad a la capa de recolección, necesitamos un canal de gran capacidad que permita almacenar datos durante un fallo del sistema, por lo tanto elegiremos un canal que almacena los datos en fichero. Este tipo de canal también nos permite la utilización de copias de seguridad de tal modo que, por ejemplo, podemos realizar una copia del contenido del canal en un disco duro diferente del nodo. Además, la utilización de este tipo de canal frente a un canal en memoria nos permite nos permite reenviar los datos una vez se ha recuperado un nodo después de un error, mientras que con un canal que almacena los datos en memoria perderíamos todos los datos que se encuentran en el canal esperando a ser enviados.

Tal y como ya hemos comentado, utilizaremos log4j para registrar las líneas de log desde el sistema de Dispensación Electrónica. Los agentes Flume recibirán los datos serializados en Avro y los enviarán a la siguiente capa de agentes. Finalmente, los datos se entregan a la capa de procesamiento aún serializados con Avro.

5.2.2.2. Capa de almacenamiento

Esta capa se encarga de almacenar los resultados preprocesados por la capa de procesamiento y de atender las consultas de los clientes. Además, almacenaremos el conjunto completo de los logs para permitir que los clientes pueda realizar consultas *ad hoc*, más allá de las que hemos predefinido. Dado que, en nuestro caso, únicamente manejamos datos estructurados, podemos utilizar una base de datos SQL para almacenar tanto los resultados como los datos completos. De esta forma, obtenemos un sistema de baja latencia al incorporar la idea presente en la arquitectura lambda de mantener un conjunto de resultados preprocesados, y además, añadimos mayor flexibilidad al permitir nuevas consultas.

A continuación se puede ver la estructura de la tabla que almacenará los resultados. Mantendremos el número de peticiones de cada operación y el tiempo de procesamiento total dedicado a esa operación por minuto. A medida que se reciben nuevos datos se deben actualizar las filas correspondientes de esta tabla con el fin de reflejar la situación actual.

Resultados
fecha {PK} operacion {PK} numPeticiones tiempoProcesamiento

Figura 5.9: Estructura tabla de resultados

De este modo, una posible fila podría ser:

Fecha	Operación	numPeticiones	tiempoProcesamiento
2015-04-07 10:16:00	identificacionPacienteAC	25	30

Figura 5.10: Ejemplo de resultado preprocesado

Esta línea se interpreta de la siguiente forma: *entre las 10:16 y las 10:17 del 7 de mayo de 2015 se han procesado 25 operaciones de tipo identificaciónPacienteAC y se han invertido un total de 30 segundos en procesar las 25 peticiones.* Si se recibe una nueva línea de log asociada a una petición de tipo *identificaciónPacienteAC* entre las 10:16 y las 10:17 deberemos actualizar esta línea incrementando el valor de numPeticiones en uno y el de tiempoProcesamiento con el valor que corresponda.

Mantener los datos completos accesibles por parte de los clientes puede ser de especial utilidad en ciertos casos excepcionales que no han sido previstos en el momento de definir las vistas del sistema de almacenamiento. Podemos pensar en un fallo del sistema, en un escenario tradicional, el responsable debería acudir a los ficheros de log para poder investigar que ha llevado al sistema al estado de error entre un gran número de datos que pueden ser irrelevantes. Con esta aproximación ponemos a disposición de los usuarios una plataforma mucho más eficaz sobre el que se pueden realizar consultas complejas permitiendo el acceso a través de SQL. A continuación se pueden ver el modelo de datos que utilizaremos para describir las diferentes líneas de log:

Peticion	Respuesta	Datos
id {PK} fecha plataforma nivelLog servicio operacion cliente	id {PK} fecha plataforma nivelLog servicio operacion cliente tamano tiempoProcesamiento	id {PK} fecha plataforma nivelLog farmacia farmaceutico tipoDocumento documento fuente

Figura 5.11: Modelo de datos

En todo caso, este modelo de capa de almacenamiento únicamente es válido cuando tenemos datos estructurados que podemos almacenar directamente en nuestra base de datos sin necesidad de un procesamiento previo. Si es necesario realizar un procesamiento complejo sobre los datos originales para poder extraer información estructurada podríamos estar perdiendo parte de los datos originales.

Para la implementación de esta capa hemos seleccionado Hive, dado que se trata de un estándar *de facto* en lo que se refiere a almacenamiento estructurado en el ecosistema Hadoop. La mayor parte de las herramientas de *reporting* son totalmente compatibles con este sistema de almacenamiento. Además, se trata de un sistema que proporciona drivers JDBC/ODBC con lo que las posibilidades que tenemos para implementar los clientes son muy amplias.

5.2.2.3. Capa de procesamiento

En esta capa se realiza el procesamiento en tiempo real de todos los datos utilizando únicamente Spark Streaming. Necesitamos un sistema eficiente para el acceso a datos que permita cumplir las restricciones temporales impuestas por un sistema de procesamiento en tiempo real. Dado que Hive se ejecuta sobre MapReduce, no se adecua a nuestro sistema por lo que utilizaremos Spark SQL para acceder directamente a los metadatos de Hive. De este modo, obtenemos un sistema de almacenamiento totalmente estándar desde el punto de vista de los clientes y un sistema de baja latencia desde el punto de vista de la capa de procesamiento.

En esta capa se procesan los datos procedentes de la capa de recolección y se actualizan los resultados preprocesados disponibles en la capa de almacenamiento. Además, se clasifican las líneas de log en peticiones, respuestas y líneas que contienen datos acerca de las operaciones y se insertan en la tabla correspondiente, se trata del único procesamiento que recibe el conjunto de datos maestro. Este procesamiento es necesario ya que tenemos un conjunto de datos heterogéneo con lo que no es posible almacenarlos en un única tabla. En todo caso, se trata de un procesamiento muy simple que no supondrá pérdida de información.

Como se puede ver en la descripción general del sistema que hemos mostrado anteriormente, hemos incluido un módulo que hace referencia al reprocesado. Como ya mencionamos en la definición de la arquitectura Lambda, el reprocesado se debe tener en cuenta siempre que se realice un cambio en la lógica de procesamiento de los datos. En nuestro caso, únicamente ejecutaremos un programa Spark, cada vez que despleguemos un nuevo sistema de procesamiento Spark Streaming, que recalculará todos los resultados a partir de los datos originales presentes en la capa de almacenamiento. Como veremos en la sección de implementación, ejecutar un código Spark Streaming como un programa *batch* no supone grandes modificaciones, únicamente deberemos leer los datos desde la capa de almacenamiento en lugar de desde la capa de recolección.

5.2.2.4. Problemas encontrados y versión final

A la hora de poner en práctica esta arquitectura nos hemos encontrado con algunos problemas que han obligado a evolucionar nuestro diseño a una nueva versión en la que se modifica alguna tecnología y se añade algún componente extra.

Como ya hemos comentado, el acceso a datos desde la capa de procesamiento se hacía a través de Spark SQL, en la versión de Spark que manejamos, la 1.2.1, no es posible realizar actualizaciones sobre una base de datos. Esto es una consecuencia directa del concepto de RDD que

ya hemos descrito. Se entiende un RDD como un conjunto de datos inmutable a partir del cual se puede obtener un nuevo RDD realizando alguna transformación. Spark SQL maneja una abstracción similar al concepto de RDD llamado Schema RDD y que, por lo tanto, se trata de conjuntos de datos inmutables con lo que únicamente permite añadir nuevos registros a la base de datos. Esta aproximación sería válida, por ejemplo, como método de acceso a datos para la capa de procesamiento *batch* de la arquitectura Lambda original ya que, como vimos en su descripción, vuelve a generar todos los resultados a partir del conjunto de datos maestro y no de forma incremental a partir de lo que ya tenemos computado en la base de datos.

La utilización de Hive directamente no era viable debido a las restricciones de rendimiento que impone un sistema de procesamiento en tiempo real. Por lo tanto, hemos decidido utilizar HBase como sistema de almacenamiento dado que ofrece un gran rendimiento en lecturas y escrituras aleatorias. Esto tiene consecuencias directas, tanto para los programas cliente que generan informes, como para los usuarios que acceden directamente a la base de datos y es que HBase no es una base de datos SQL. Con el fin de ofrecer una interfaz SQL introduciremos un nuevo componente en la capa de almacenamiento. Crearemos una serie de vistas en Apache Phoenix sobre los datos presentes en HBase de tal forma que los clientes ejecutarán sus consultas a través de este nuevo componente y no directamente sobre la base de datos. A continuación se puede ver un esquema de la arquitectura final:

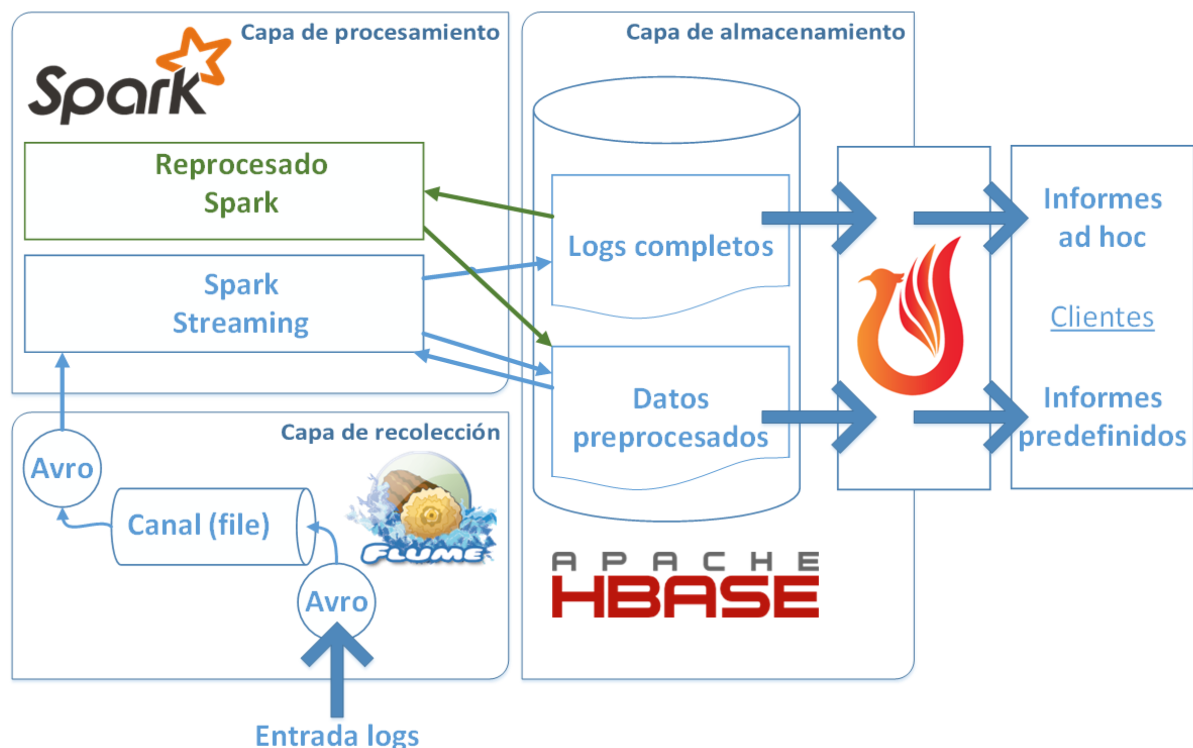


Figura 5.12: Arquitectura alternativa final

Conseguimos de este modo todas las ventajas que nos proporciona el lenguaje SQL a la hora de realizar análisis sobre los datos y de obtener interoperabilidad con herramientas de consulta y generación de informes y, además, tenemos un mecanismo de acceso a datos eficiente desde el punto de vista de la capa de procesamiento.

Como hemos visto en esta nueva versión la capa de almacenamiento se ha rediseñado completamente. Contaremos con dos tablas en HBase, una dedicada a almacenar el conjunto completo de datos (todas las líneas de log que se reciben) y otra dedicada a mantener los resultados preprocesados.

La primera de las tablas la llamaremos "log" y contará con una única *column family* que llamaremos "*campos*", las columnas que la forman dependerán del tipo de línea que se esté almacenando. Se contemplan cuatro tipos de líneas diferentes: "Petición", "Respuesta", "Datos" y "Desconocida". En esta clasificación se contemplan todas las líneas descritas anteriormente además de un nuevo tipo que utilizaremos para aquellas líneas que no cumplan con el patrón de los otros tres tipos, de este modo, cumplimos con el objetivo de mantener el conjunto de datos completo sin pérdida de datos aunque recibamos líneas con formados no esperados. Utilizaremos una clave compuesta que consistirá en la fecha en la que se ha generado la línea convertida a bytes concatenado con la cadena de caracteres que representa el tipo de línea "Petición", "Respuesta", "Datos" o "Desconocida" utilizando como separador un byte a cero.

Por otro lado, tenemos la tabla que almacena los resultados, "result". Esta tabla tiene una estructura similar a la que comentábamos en la primera versión, estando compuesta por una *column family* con dos columnas para todas las líneas:

- **tproc:** representa el tiempo de procesamiento.
- **npet:** representa el número de peticiones.

Insertaremos una línea por cada minuto y tipo de operación al igual que hemos comentado anteriormente con lo que utilizaremos la misma estructura de clave que en la tabla "log", en este caso la cadena de caracteres representa el tipo de operación en lugar del tipo de línea y la fecha representa el minuto para el que se está realizando la agregación.

Crearemos una vista en Phoenix que mapeará la tabla "result" de HBase, de este modo ofreceremos exactamente la misma interfaz SQL que en el caso anterior en el que utilizábamos Hive.

5.2.3. Software de procesamiento

En esta sección expondremos el funcionamiento de la capa de procesamiento a un mayor nivel de detalle. En la figura 5.13 se pueden ver lar transformaciones y las acciones que se llevan a cabo para obtener los resultados a partir del flujo de datos de entrada.

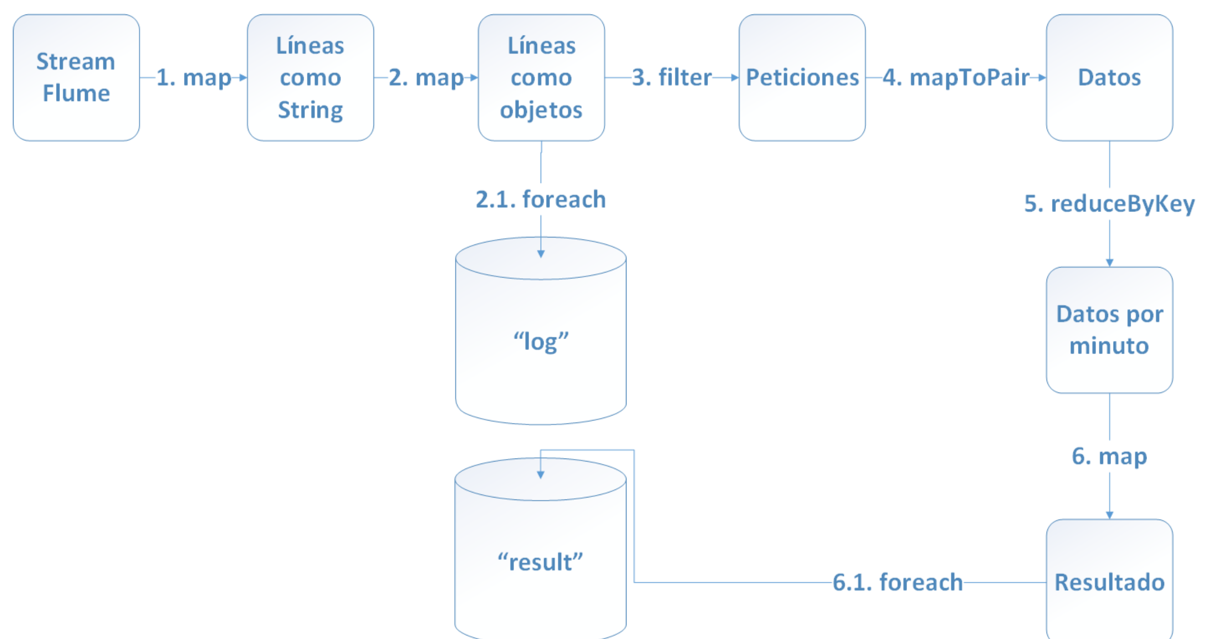


Figura 5.13: Transformaciones Spark Streaming

1. **map:** transforma los datos que se reciben de Flume serializados con Avro en un conjunto de cadenas de caracteres que representan las líneas sin serializar.
2. **map:** a partir de cada una de las líneas se crea un objeto dotando de estructura al RDD.
   2.1. **foreach:** se almacenan los datos en el conjunto de datos maestro.
3. **filter:** se filtran los datos con el fin de mantener únicamente las líneas asociadas a las respuestas que son las líneas que contienen información acerca del tiempo de procesamiento invertido en cada operación.
4. **mapToPair:** por cada línea se crea un par clave-valor, la clave se forma con la fecha truncada a los minutos y el tipo de operación, por otro lado, el valor contiene el tiempo de procesamiento para la línea que se está procesando y se inicializa el número de operaciones a 1.
5. **reduceByKey:** para todas las claves iguales (mismo tipo de operación y mismo minuto) se suman los tiempos de procesamiento y el número de peticiones.
6. **map:** por último, se convierten los pares clave valor a un único tipo de datos con la misma estructura que la tabla en la que se almacenaran en HBase, un objeto con cuatro atributos: tipo de operación, fecha, tiempo de procesamiento y número de peticiones.
   6.1. **foreach:** se almacenan los resultados en la tabla “result”.

5.3. Nodos Sistema de Dispensación Electrónica

En el requisito RCA_003 se recoge que se debe desarrollar una aplicación que simule el funcionamiento del sistema de Dispensación Electrónica puesto que no tendremos acceso al sistema real. Además, en el requisito RAP_001 se especifica que el registro de las líneas de log se debe realizar a través de la herramienta log4j. El sistema de Dispensación Electrónica interactuará con el analizador de logs a través de la capa de recolección con lo que únicamente necesitamos poder enviar las líneas de log a través de Flume. Tal y como veremos en capítulo de implementación, es posible integrar log4j con Flume simplemente modificando los ficheros de configuración de log4.

Para realizar la simulación del sistema de Dispensación Electrónica contamos con ficheros de log generados por el sistema real durante unas catorce horas. Estos ficheros de log los tenemos almacenados con la estructura que se muestra en la figura 5.14:

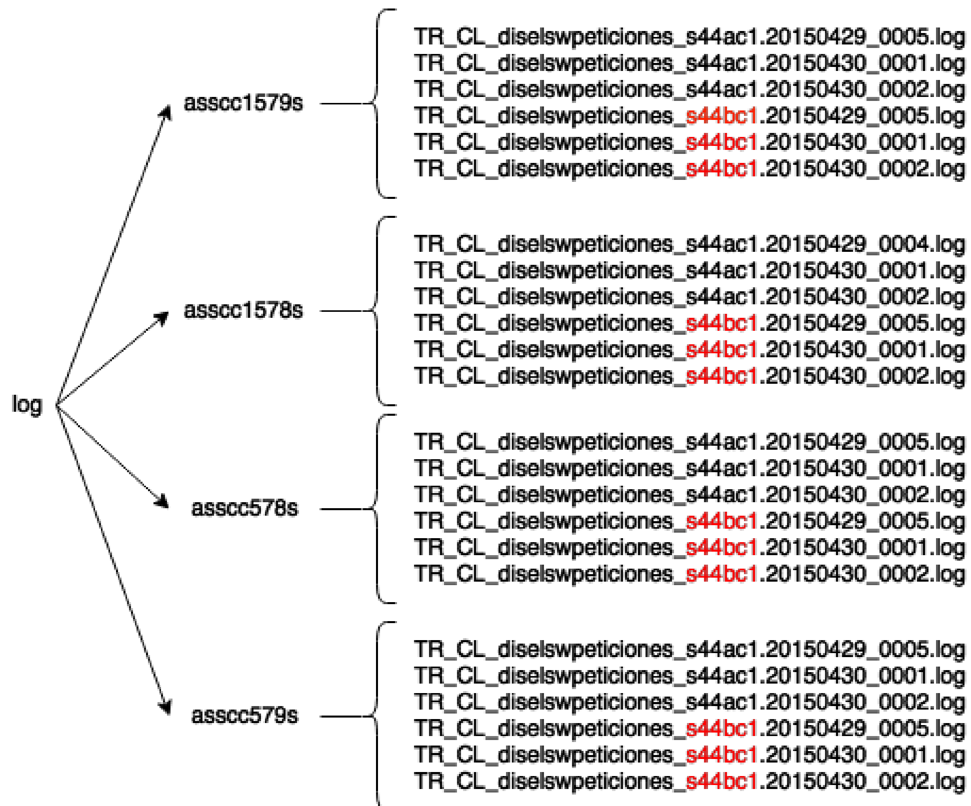


Figura 5.14: Estructura de carpetas de logs

Por cada uno de los cuatro nodos que componen el sistema de Dispensación Electrónica tenemos una carpeta que contiene los ficheros de log generados por ese nodo. Dentro de esta carpeta tenemos los log generados por las dos instancias que se están ejecutando en cada nodo. Como se puede ver en la imagen los nombres de los ficheros contienen el código *"s44ac1"* o el código *"s44bc1"*, que nos permite diferenciar entre los ficheros generados por cada una de las instancias. Además, en el nombre se incluye la fecha en que han sido generados y el número de secuencia del fichero dentro de esa fecha ya que se sigue una política de rotación por la que cada uno de los ficheros ocupa 10 MB, una vez se ha alcanzado este límite se almacenan las nuevas líneas en un nuevo fichero.

Crearemos cuatro contenedores Docker que simularan el funcionamiento de cada uno de los nodos. En cada contenedor se ejecutará una aplicación que leerá los datos de una carpeta y enviará las líneas a la hora indicada en el primer campo de cada una. De este modo, los datos se generarán a la misma velocidad que el sistema de Dispensación Electrónica original. En la figura 5.15 se puede ver un diagrama de flujo en el que se describe este proceso.

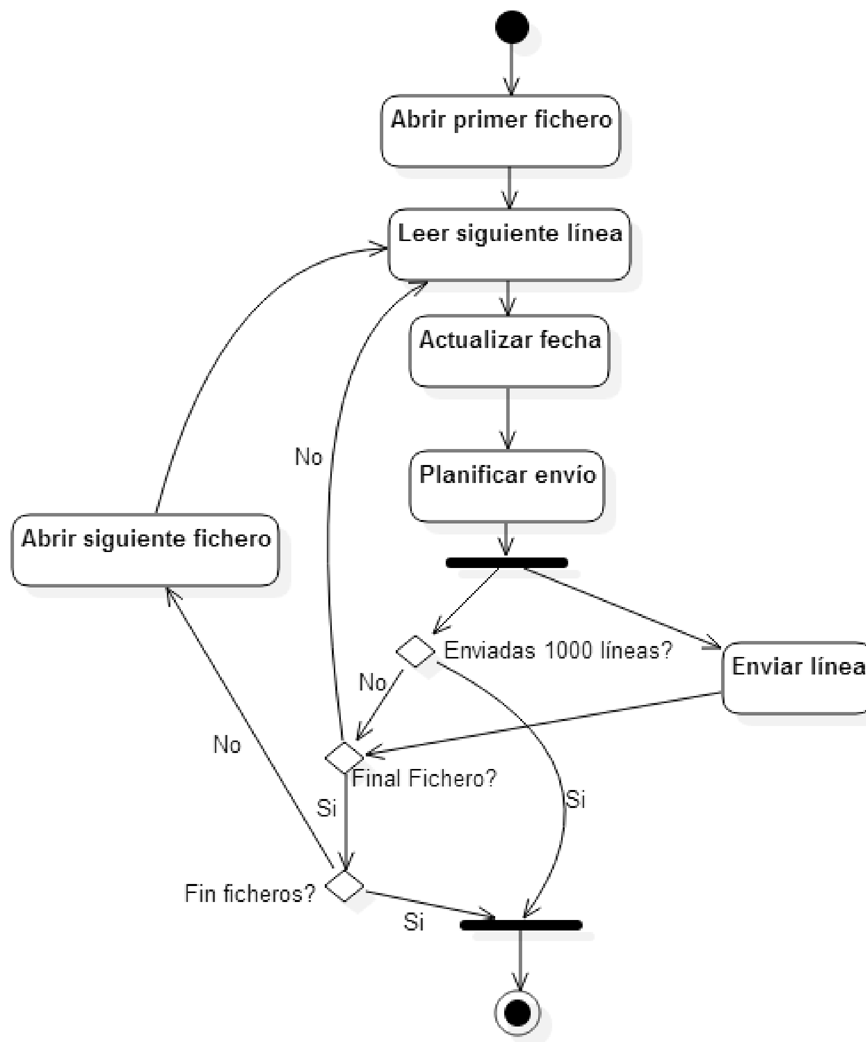


Figura 5.15: Diagrama de flujo de envío de líneas de log

En primer lugar abrimos el fichero y leemos una línea, a continuación se lee la fecha y se actualiza el día, mes y año al actual, por último se planifica el envío para esa fecha. Este proceso se ejecuta hasta que se han programado para su envío las 1000 primeras líneas. En caso de que se alcance el final de fichero se debe abrir el siguiente por orden según la política de rotación que ya hemos comentado. Cuando se alcance el final del último fichero el algoritmo termina. Hasta este punto tenemos programadas las 1000 primeras líneas, cuando una línea sea enviada, se programa una nueva línea siguiendo el mismo proceso. De este modo se busca tener siempre en torno a 1000 líneas programadas (en el momento en que una se envía y hasta que se lea la siguiente no hay exactamente 1000 líneas programadas) que actúan a modo de buffer para que no haya problemas en el caso en que se deban enviar las líneas a un ritmo mayor al de lectura.

Este algoritmo se debe ejecutar dos veces en cada nodo, una para cada una de las instancias del sistema de Dispensación Electrónica.

5.4. Nodos del clúster Hadoop

Como distribución de Hadoop seleccionaremos **Hortonworks Data Platform (HDP)**, concretamente la versión 2.2, la última disponible en el momento de realizar este proyecto. Esta elección la hemos realizado, principalmente, pensando en el despliegue del clúster en contenedores Docker [38]. Existe una *startup* denominada SequenceIQ que ofrece una serie de productos que

ayudan a desplegar un clúster Hadoop sobre contenedores Docker en la nube. La distribución con la que trabajan es precisamente con la producida por Hortonworks y, de hecho, durante la realización de este proyecto SequenceIQ ha sido adquirida por parte de la propia Hortonworks [39], lo que demuestra el interés de esta distribución en facilitar el despliegue de Hadoop utilizando contenedores como tecnología de virtualización. Además HP mantiene una relación muy estrecha con Hortonworks siendo uno de los principales miembros de su programa de *partners*.

5.5. Sistema de monitorización

Como ya hemos comentado, desarrollaremos un sistema de monitorización del sistema con el fin de poder comprobar que todo funciona según lo esperado. Para poder medir si el ritmo de procesamiento es lo suficientemente rápido como para poder soportar la carga a la que se está sometiendo el sistema, inspeccionaremos el nivel de ocupación de los canales en los agentes Flume. Un nivel alto de utilización indicará que el ritmo de procesamiento es muy inferior al de generación. Además, con el fin de comprobar que no se ha producido pérdida de datos desde su generación hasta su almacenamiento en HBase comprobaremos el número de eventos enviados y recibidos con éxito de cada agente, esto nos permitirá averiguar en qué punto se están perdiendo dichos datos.

Para la implementación de este sistema utilizaremos las funcionalidades que ofrece Flume, en concreto, la posibilidad de levantar un servidor HTTP que devuelva el estado actual del agente. Implementaremos una aplicación Java que consulte este estado en periodos de tiempo regulares y vuelque la información en un fichero. Posteriormente podremos analizar estos datos y concluir si las pruebas se han realizado o no con éxito. Para analizar los datos utilizaremos un script Python que genere gráficos acerca del tamaño de los canales (requisito RDI_007) y otro acerca de los eventos enviados y recibidos (requisito RDI_007).

5.6. Aplicación web

En el requisito RDI_001 se establece la necesidad de crear una serie de informes predefinidos que se pongan a disposición de los usuarios a través de una aplicación web. En esta sección nos centraremos en el diseño de esta aplicación. Implementaremos un API de servicios REST que permita el acceso a estos informes en formato HTML. Los informes se generarán a partir de una serie de ficheros de definición creados con la herramienta Report Designer. Además los usuarios podrán definir nuevos informes a través de esta herramienta utilizando la configuración que explicaremos en la sección de implementación, con esto cumplimos con el requisito RAP_003. Además, la implementación de un API como método de publicación de informes facilitará la integración en un sistema mayor en el que se pueda mostrar un cuadro de mando con información acerca de múltiples sistemas y no solo del sistema de Dispensación Electrónica.

Implementaremos un patrón modelo-vista-controlador (MVC) en el que el papel de modelo lo jugará una aplicación Java, el controlador será una clase que exponga los métodos de acceso a través de HTTP utilizando el framework de creación de servicios *REST* Jersey y que invocará los métodos del modelo para generar los informes. Por último, implementaremos la vista utilizando una página HTML basada en el framework *Bootstrap* y accederemos a los servicios REST a través de JQuery.

A la hora de implementar el modelo utilizaremos una *fachada* que dará acceso a todas las funcionalidades de generación de informes y que implementará un patrón *singleton* con el fin de que exista, como máximo, una única instancia de esta clase. Utilizaremos la clase Reporte como un VO (*value object*) en el que se encapsula la ruta en la que se encuentra la definición del informe y los parámetros que se deben utilizar a la hora de generar dicho informe. Las demás clases se utilizan para generar el informe utilizando el SDK de Pentaho Reporting [40] y encapsular el resultado en formato

HTML. En la figura 5.16 se muestra el diagrama de clases de aplicación web, las clases sombreadas en un color más oscuro representan clases propias del SDK.

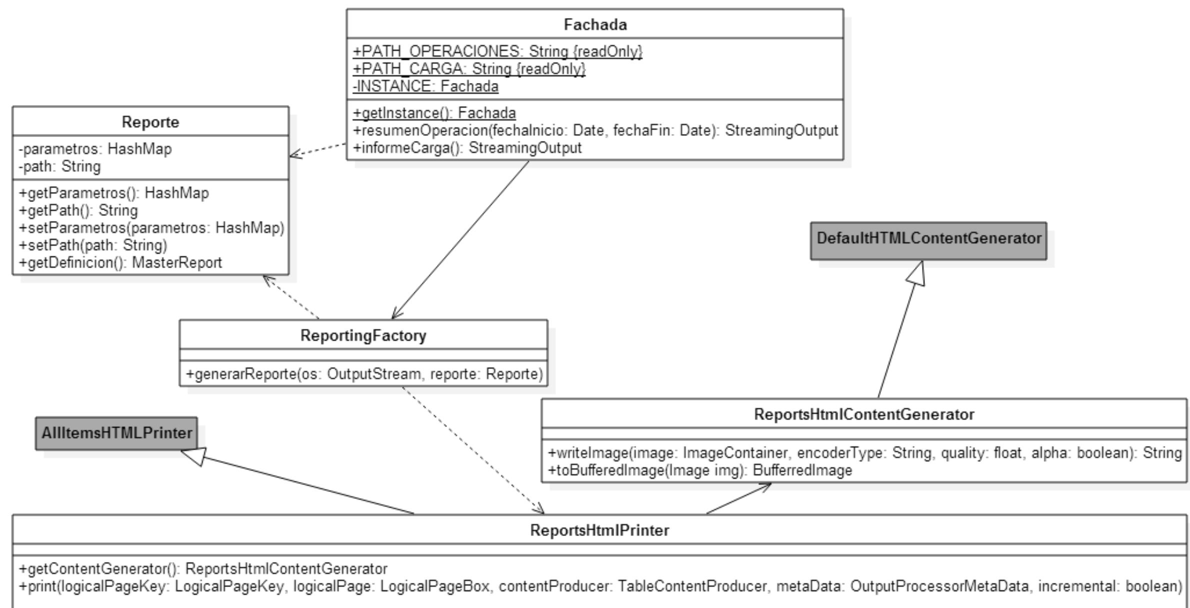


Figura 5.16: Diagrama de clases de aplicación web

En la figura 5.17 se muestra un diagrama de secuencia para la creación del informe acerca del uso del sistema desglosado por operación (explicaremos el contenido de este informe en detalle al final de esta sección). La clase *InformesResource* juega el papel de controlador en nuestra arquitectura MVC, y por lo tanto, es el encargado de atender las peticiones de las vistas. Además, hemos incluido un objeto denominado SDK que representa el SDK de Pentaho con el fin de poder mostrar con mayor claridad el funcionamiento del programa sin la necesidad de representar todas las clases que lo conforman ganando, de este modo, en claridad en el diagrama.

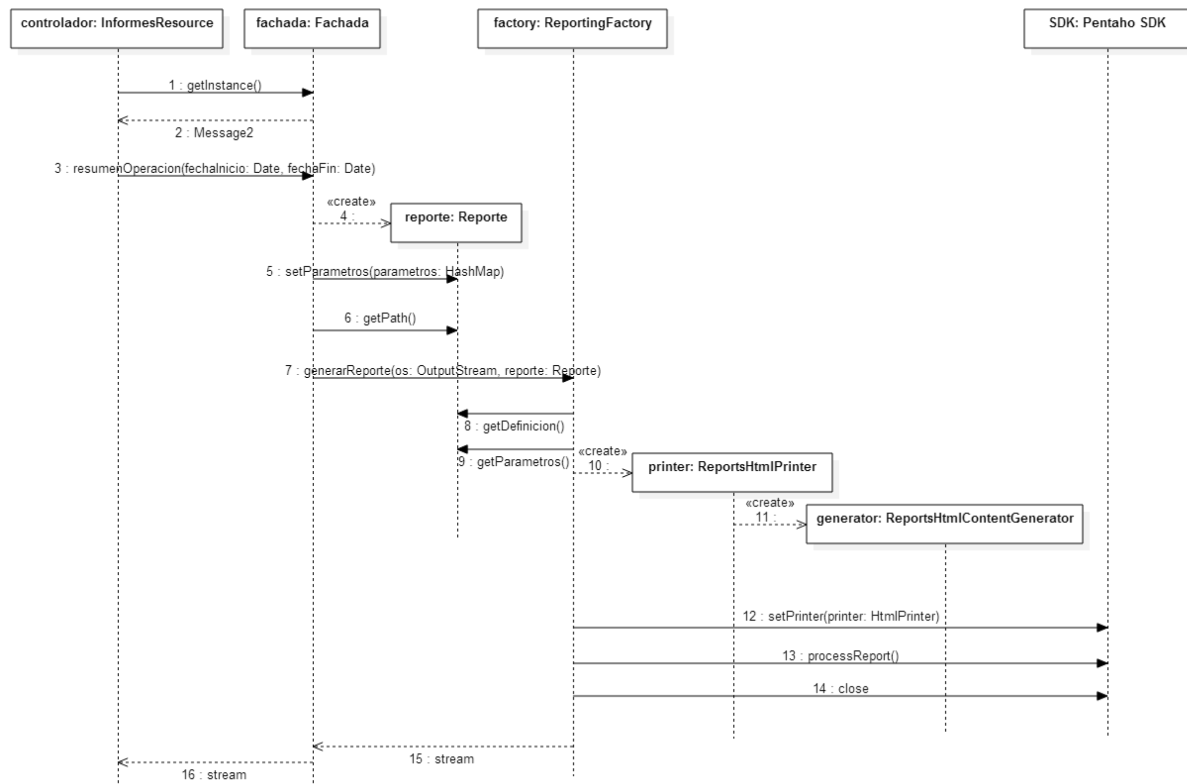


Figura 5.17: Diagrama de secuencia generación de informes

5.6.1. Operaciones por unidad de tiempo

Uniremos los requisitos de difusión de la información RDI_004 y RDI_005 en un único informe que contendrá un gráfico asociado al número de operaciones por franja horaria durante día actual y otro gráfico que contendrá el número de operaciones por minuto durante la hora actual. Este informe estará disponible a través de la siguiente URL (suponiendo que la aplicación se encuentra desplegada en *localhost* con puerto *8080*):

http://localhost:8080/Reporting/api-informes/1.0/informes/carga

- Devuelve el informe comentado anteriormente a partir de los datos actuales en formato HTML.
- No acepta ningún parámetro adicional de entrada.

5.6.2. Uso del sistema por operación

En el requisito RDI_006 se expone la necesidad de un informe que recoja la actividad del sistema desglosado por cada tipo de operación. Por ello, diseñaremos un informe que contiene una tabla en la que para cada uno de los tipos de operación (filas) incluiremos el tiempo de procesamiento, el número de peticiones y el tiempo de procesamiento medio para cada operación. Este informe será parametrizado de tal forma que aceptará el intervalo de tiempo en el que se tendrán en cuenta los datos. En este caso la URL será la siguiente:

http://localhost:8080/Reporting/api-informes/1.0/informes/operaciones

- Devuelve el informe comentado anteriormente a partir de los datos actuales en formato HTML.
- Parámetros:
    - **inicio:** fecha de inicio desde la que se tendrán en cuenta los datos. Debe estar en formato “*dd-MM-yyyy hh:mm*”.
    - **fin:** fechas de fin hasta la que se tendrán en cuenta datos. Debe ser posterior a la fecha de inicio y estar en formato “*dd-MM-yyyy hh:mm*”.

6. Implementación

En este capítulo comentaremos la implementación final del sistema. Para ello nos apoyaremos en los diseños propuestos, así como en el análisis tecnológico realizado anteriormente.

6.1. Despliegue y configuración del clúster

Como ya hemos comentado, desplegaremos un clúster Hadoop utilizando contenedores Docker. Como punto de partida, tomaremos unos scripts desarrollados por la empresa SequenceIQ [38], los modificaremos y los adaptaremos a nuestras necesidades. Junto a estos scripts, también se proporciona una imagen Docker que utilizaremos como base para el desarrollo de nuestra propia imagen.

A continuación describiremos las modificaciones que hemos realizado sobre los *scripts* originales.

6.1.1. Permitir reanudar el clúster

En el script original se ofrece una función denominada *amb-start-cluster* que crea un clúster Hadoop con un número de nodos arbitrario. Para cada nodo se crea un nuevo contenedor Docker, un problema es que, si queremos parar estos contenedores (los contenedores se paran de forma automática en el momento en el que se reinicia el *host* sobre el que se están ejecutando) no podemos volver a iniciarlos puesto que la dirección IP que se asigna a cada uno de ellos es dinámica con lo que, potencialmente, se pueden asignar direcciones diferentes a las originales lo que imposibilitaría la comunicación entre los nodos del clúster.

Establecer direcciones estáticas para cada contenedor no está soportado puesto que sería contrario a la filosofía de Docker de crear entornos de ejecución independientes totalmente portables a otras máquinas. Permitir esta característica podría suponer un conflicto entre diferentes contenedores que tengan asignada la misma dirección que se han estado ejecutando en máquinas independientes y que se migran a un mismo host. Como solución, a la hora de enlazar contenedores de forma independiente a su dirección IP, se ofrece la opción --*link* en el comando de creación de un nuevo contenedor [33]. Esta alternativa es demasiado restrictiva para nuestro caso puesto que si enlazamos todos los contenedores en el momento de su creación, cuando queramos añadir un nuevo nodo deberemos modificar la lista de contenedores enlazados de todos los nodos del clúster Hadoop, lo cual no es posible puesto que no se permite modificar este parámetro en tiempo de ejecución. En su lugar, nos hemos beneficiado del hecho de que los contenedores originales utilizaban como software de gestión de errores *Serf* [41], sistema que permite gestionar clúster genéricos a los que un nodo se puede unir simplemente contactando con uno de los nodos del clúster original. Además, incluye mecanismos de gestión de eventos como que un nodo abandone el clúster, siendo la utilidad que se le daba en la implementación inicial puesto que cuando un nodo falla se dispara este evento.

Para conseguir esto, hemos creado una función denominada *hdp-create-cluster* (nótese que se ha sustituido el prefijo *amb* por *hdp* en todas las funciones del nuevo script) que crea un clúster de tamaño arbitrario (por defecto de 3 nodos) y asigna el nombre *amb*X a cada contenedor, siendo X un número consecutivo entre 0 y el número máximo de nodos del clúster menos uno. En el contenedor *amb0* se encuentra el servidor Ambari que gestiona todo el clúster. De este modo los siguientes contenedores creados cuentan con una opción del tipo --*link amb0:servidor* que crea una entrada en el fichero */etc/hosts* de cada uno de los contenedores con la IP del contenedor *amb0* y, en este caso, el nombre de host *servidor*. Cada vez que el contenedor *amb0* cambia de dirección IP, Docker se encargará de modificar en todos los contenedores este fichero para asegurar que la IP es correcta. A

continuación se puede ver el comando que se ejecuta internamente para crear este primer contenedor:

```
$ docker run -v /usr/hdp:/compartida -d $DOCKER_OPTS --name
$AMBARI_SERVER_NAME -h $AMBARI_SERVER_NAME.$MYDOMAIN $BASE_IMAGE --tag
ambari-server=true
```

Comando 6.1

- La opción "-v" nos permite montar la carpeta */usr/hdp* del host en la ruta */compartida* dentro del contenedor. Todos los contenedores creados incluirán esta opción lo que nos permitirá intercambiar ficheros de forma sencilla entre todos los contenedores y el host principal.
- La opción "-d" hará que el contenedor se ejecute en segundo plano, es decir, no interactúa directamente con el usuario (modo *demonio*).
- En la variable *"DOCKER_OPTS"* podemos establecer nuevas opciones. Por defecto solo contiene el servidor de DNS que nos permitirá poder tener acceso a internet dentro del contenedor y el *"entrypoint"*, esto es el primer comando que se ejecutará al iniciar el contenedor y que en este caso será: */usr/local/serf/bin/start-serf-agent.sh*. Esto hace que al iniciar el contenedor se inicie también un *Agente Serf*. Cada nodo que pertenece a un clúster gestionado por Serf debe ejecutar un agente que se encarga de realizar toda la gestión de eventos (incorporaciones de nuevos nodos, abandono de nodos del clúster, etc.).
- Con la opción "--name" se establece el nombre del contenedor, en este caso, el valor de la variable *AMBARI_SERVER_NAME*, que como ya hemos comentado, por defecto será *amb0.*
- Con "-h" establecemos el nombre del host dentro de la red (que puede ser diferente al nombre del contenedor).
- La variable *BASE_IMAGE* contiene el nombre de la imagen en la que se basa el contenedor, en nuestro caso utilizaremos una imagen propia denominada "user/hdp:1.0".
- Por último, etiquetamos el contenedor como un servidor Ambari.

En el caso de los demás nodos el comando de creación es el siguiente:

```
$ docker run -v /usr/hdp:/compartida --link
$AMBARI_SERVER_NAME:ambari_server -d -e SERF_JOIN_IP=$AMBARI_SERVER_IP
$DOCKER_OPTS --name ${NODE_PREFIX}$NUMBER -h
${NODE_PREFIX}${NUMBER}.$MYDOMAIN $BASE_IMAGE
```

Comando 6.2

La principal diferencia radica en la inclusión de "--link" que crea una entrada en /etc/hosts para ambari_server con la IP del contenedor amb0. Además, con "-e" se añade la variable de entorno SERF_JOIN_IP que coge como valor la IP del contenedor amb0, esta variable se utiliza para inicializar el agente Serf.

Por otro lado hemos creado una función denominada *hdp-stop-cluster* que, básicamente, ejecuta *docker stop* sobre todos los contenedores del clúster Hadoop. Por último, tenemos la función que buscábamos con todas estas modificaciones y que llamaremos *hdp-start-cluster*, esta función reanuda la ejecución del clúster volviendo a iniciar los contenedores que previamente hemos parado.

```
hdp-start-node() {
  NUMBER=$1
  if [ $# -eq 1 ] ;then
```

```
        docker start ${NODE_PREFIX}$NUMBER
        if [ $NUMBER -ge 1 ] ;then
          docker exec -d ${NODE_PREFIX}$NUMBER /compartida/join-
cluster.sh
        fi
      fi
    }
```

Código 6.1

Con este código iteramos sobre todos los contenedores y los iniciamos con *docker start*. El comando *docker exec* nos permite ejecutar un comando en un contenedor que ya se encuentra en ejecución, en este caso se ejecuta */compartida/join-cluster.sh*, un script que hemos desarrollado para reconfigurar el agente Serf y que se una al clúster a través de la nueva dirección IP de *amb0* que tenemos actualizada en */etc/hosts*.

6.1.2. Soporte para realizar imágenes sobre el estado actual del clúster

Como ya hemos comentado en la descripción de Docker, la definición de los contenedores se realiza mediante el concepto de imagen, de este modo, se puede entender un contenedor como una instancia de una imagen. Docker también nos proporciona mecanismos para seguir el proceso inverso, es decir, crear una imagen a partir de un contenedor. Esto nos permite obtener una instantánea del estado actual de nuestro contenedor a partir de la cual podremos crear nuevos contenedores. Además, las imágenes se encuentran bajo un sistema de control de versiones con lo que podemos contar con varias imágenes sobre un mismo contenedor relacionadas con instantes de tiempo diferentes.

De este modo, hemos creado una función que nos permite crear una imagen de todos los contenedores del clúster a partir de las cuales podremos volver a ejecutar los contenedores retomando el estado en ese punto. Para crear estas imágenes únicamente necesitamos iterar por todos los contenedores y ejecutar *docker commit* pasando el nombre de la imagen que queremos crear. A continuación se muestra el comando que hemos utilizado:

```
$ docker commit ${NODE_PREFIX}$NUMBER ${IMAGE_PREFIX}${NUMBER}:$TAG
```

Comando 6.3

El nombre de la imagen se crea de la misma forma que el nombre de los contenedores, con un prefijo y un número incremental, por último, utilizaremos una etiqueta que permite identificar la versión de la imagen.

6.1.3. Automatización de la instalación de los servicios en el clúster

Una de las principales aportaciones de SequenceIQ al ecosistema Hadoop ha sido el desarrollo de Ambari Shell, una interfaz de línea de comando que permite interactuar con Ambari, lo que facilita la automatización de los procesos de instalación y gestión del clúster. En nuestro caso hemos evolucionado la implementación inicial, que requería la ejecución de los comandos necesarios por parte del usuario para la instalación de todos los componentes necesarios en el clúster, hacia un modelo más automatizado en el que simplemente es necesario ejecutar un comando que inicia todo el proceso de instalación y despliegue.

Uno de los conceptos principales en Ambari es el de *blueprint* [42]. Se trata de ficheros en los que se definen los componentes que se deben instalar en cada uno de los nodos del clúster. Los nodos se agrupan en categorías de tal forma que únicamente se definen una vez los componentes que se instalarán en todos los nodos de una misma categoría (figura 6.1). En nuestro caso tendremos dos

categorías, el nodo principal que contiene el servidor Ambari y en el que se instalarán todos los componentes *maestro* de los diferentes servicios del ecosistema Hadoop y una segunda categoría a la que pertenecen todos los demás nodos y que jugarán el papel de nodos esclavo. Esta clasificación que hemos seleccionado provoca que tanto el NameNode como el SecondaryNameNode se ejecuten en el mismo nodo, a pesar de que, como ya hemos comentado, no es recomendable. En todo, caso los recursos computacionales de los que disponemos no nos permiten desplegar un clúster con un mayor número de nodos con una mayor distribución de las funciones necesarias.

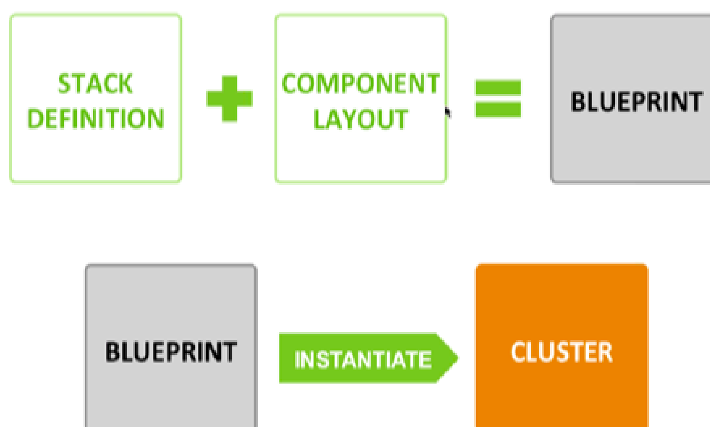


Figura 6.1: Ambari blueprint

Con todas las funciones definidas anteriormente, para la creación del clúster únicamente necesitamos ejecutar los siguientes comandos:

```
$ . hdp-cluster.sh
$ hdp-create-cluster
$ hdp-provisioning
```

Comando 6.4

Con el primer comando cargamos las funciones en el entorno actual ejecutando el script *hdp-cluster.sh*, a continuación, creamos el clúster con *hdp-create-cluster* y, por último, con *hdp-provisioning* instalamos todos los componentes indicados en el fichero */compartida/hdp-blueprint*. Es importante tener en cuenta que los contenedores mapean la carpeta */usr/hdp* a la ruta */compartida* y utilizan una serie de recursos que deben estar en la carpeta */usr/hdp*. Por lo tanto, para que todo funcione correctamente debemos copiar todas las dependencias a esta carpeta, para lo que hemos desarrollado la función *hdp-crear-carpeta* que crea la nueva carpeta y copia todos los ficheros necesarios (que se encuentran en el material adjunto a esta memoria) en dicha carpeta.

6.1.4. Instalación y configuración de servicios adicionales

Desafortunadamente, en el momento de realizar este proyecto Ambari no ofrece soporte para Spark y Phoenix, por lo tanto, deberemos instalar y configurar estos servicios de forma manual. Como ya hemos comentado en la comparativa de las diferentes distribuciones de Hadoop, Ambari todavía no ha alcanzado el nivel de madurez de otras herramientas como es el caso de Cloudera Manager, no obstante, está en constante evolución y ya existen líneas de trabajo para incluir soporte a estos servicios.

6.1.4.1. Instalación y configuración de Spark

Se trata de un proceso muy simple puesto que únicamente deberemos ejecutar el siguiente comando en uno de los contenedores:

```
$ /compartida/spark-install.sh install
```

Comando 6.5

Este script también lo proporciona SequenceIQ [43] y nos permite instalar Spark en nuestro clúster Hadoop únicamente ejecutándolo en un nodo puesto que se encarga de almacenar en HDFS las dependencias necesarias. La instalación incluye la herramienta *spark-submit* que permite enviar un programa al clúster para su ejecución. Si únicamente ejecutamos este comando en un nodo, sólo podremos lanzar programas desde dicho nodo. Si necesitamos poder ejecutar programas Spark desde diferentes nodos deberemos instalar esta utilidad en todos ellos.

Además, hemos configurado el sistema de generación de log de Spark para tener más información a la hora de evaluar si el sistema se ha ejecutado correctamente. Spark utiliza la herramienta log4j para la definición de los ficheros de log que genera. Podremos configurar este comportamiento a través del fichero */usr/local/spark/conf/log4j.properties*. En nuestro caso hemos escogido la siguiente configuración:

```
log4j.rootCategory=INFO, console, file
log4j.appender.console=org.apache.log4j.ConsoleAppender
log4j.appender.console.target=System.err
log4j.appender.console.layout=org.apache.log4j.PatternLayout
log4j.appender.console.layout.ConversionPattern=%d{yy/MM/dd
HH:mm:ss} %p %c{1}: %m%n

log4j.appender.file=org.apache.log4j.RollingFileAppender
log4j.appender.file.File=/var/log/spark-streaming.log
log4j.appender.file.MaxFileSize=10MB
log4j.appender.file.MaxBackupIndex=10
log4j.appender.file.layout=org.apache.log4j.PatternLayout
log4j.appender.file.layout.ConversionPattern=%d{yy/MM/dd
HH:mm:ss} %p %c{1}: %m%n
log4j.appender.file.Threshold = WARN
```

Código 6.2

De este modo enviamos toda la salida de log a la consola y únicamente enviamos a un fichero las líneas de log de nivel WARN o superior. En la implementación Java que hemos realizado registramos varias líneas en caso de que se produzca una excepción informando del error.

6.1.4.2. Instalación y configuración de Phoenix

En el caso de Phoenix necesitamos instalarlo en todos los nodos del clúster. En nuestro caso tenemos un tamaño muy pequeño y manejable con lo que no supone un problema, en todo caso, pensando en un clúster formado por una gran cantidad de nodos podría ser interesante automatizar el proceso, por ejemplo, realizando un script que itere sobre todos los contenedores y ejecute el comando de instalación en cada uno de los nodos a través de *docker exec*. Nosotros ejecutaremos el siguiente comando en los tres nodos [44]:

```
$ yum install Phoenix
```

Comando 6.6

A continuación deberemos reiniciar tanto HBase como ZooKeeper para que Phoenix funcione correctamente. Por último debemos crear las tablas que utilizaremos a la hora de realizar las consultas SQL para generar los informes. Es importante haber creado previamente las correspondientes tablas en HBase, de lo contrario, el proceso de creación de las vistas fallará informando de que no se ha encontrado la tabla correspondiente en HBase. Para interactuar con Phoenix utilizaremos el script *sqlline.py* que se encuentra en */usr/hdp/2.2.0.0-2041/phoenix/bin*, para ello ejecutaremos el siguiente comando:

```
$ ./sqlline.py localhost:2181/hbase
```

Comando 6.7

La siguiente sentencia crea una vista sobre la tabla *“result”* que se encuentra en HBase y coge todas las columnas de la familia *“result”*, que, como hemos definido en el diseño, es la única que tenemos en esta tabla. Además, dota de tipo a cada una de las columnas. Recordar que en HBase todo se almacena como un *array* de *bytes*. De este modo permitimos realizar consultas SQL sobre los datos de la tabla *“result”* de HBase.

```
CREATE VIEW "result" (
    "fecha" UNSIGNED_TIMESTAMP NOT NULL,
    "operacion" VARCHAR NOT NULL,
    "tproc" UNSIGNED_LONG,
    "npet" UNSIGNED_LONG,
    CONSTRAINT pk PRIMARY KEY("fecha", "operacion")
)default_column_family='result';
```

Código 6.3

6.1.4.3. Instalación y configuración de Flume

A continuación describiremos el proceso de configuración de un nuevo agente Flume desde la herramienta web de Ambari. En la siguiente figura se puede ver la página principal de esta interfaz en la que tenemos información acerca del estado actual del clúster.

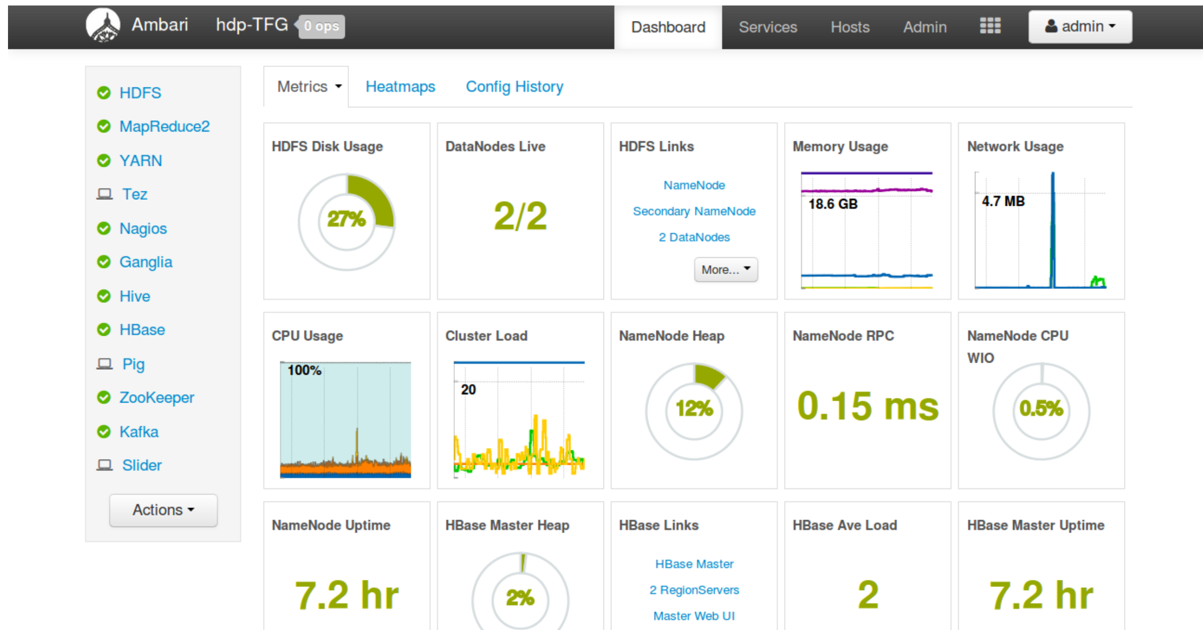


Figura 6.2: Página principal Ambari

Abajo a la izquierda, en *Actions* tenemos un desplegable que nos permite instalar nuevos servicios en el clúster a través de la opción *Add Service*. A continuación nos saldrá una ventana emergente en la que podremos seleccionar el servicio que deseamos instalar, en este caso seleccionamos Flume.

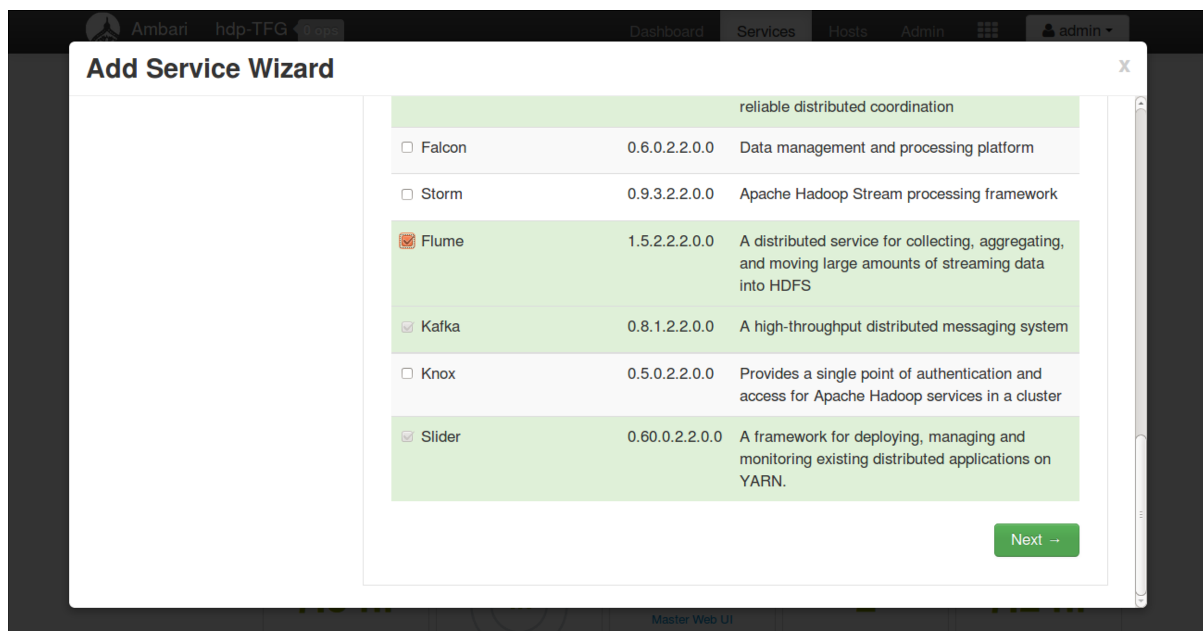


Figura 6.3: Instalar nuevos servicios

Dejaremos todas las configuraciones con su valor por defecto a excepción de la configuración de los componentes del agente Flume tal y como se puede ver en la siguiente figura. Este agente será el encargado de agregar en un único flujo los datos los cuatro agentes Flume de los nodos del sistema de Dispensación Electrónica y utilizará tanto un sumidero como una fuente que manejan eventos serializados en Avro. Como capacidad máxima del canal hemos seleccionado 2.000.000 eventos, esto, tal y como veremos a continuación en la descripción de los agentes del sistema de Dispensación Electrónica nos permitiría almacenar todos los eventos generados en un día. Se trata de la misma

capacidad con la que cuentan, en conjunto, los cuatro agentes de la capa anterior, de este modo, tras un periodo de un día de inactividad del clúster (periodo máximo para el que se garantiza que no ocurre pérdida de datos) todos los datos podrían pasar de los agentes de la primera capa a este agente dejando a los demás totalmente liberados para almacenar nuevos eventos que se generen.

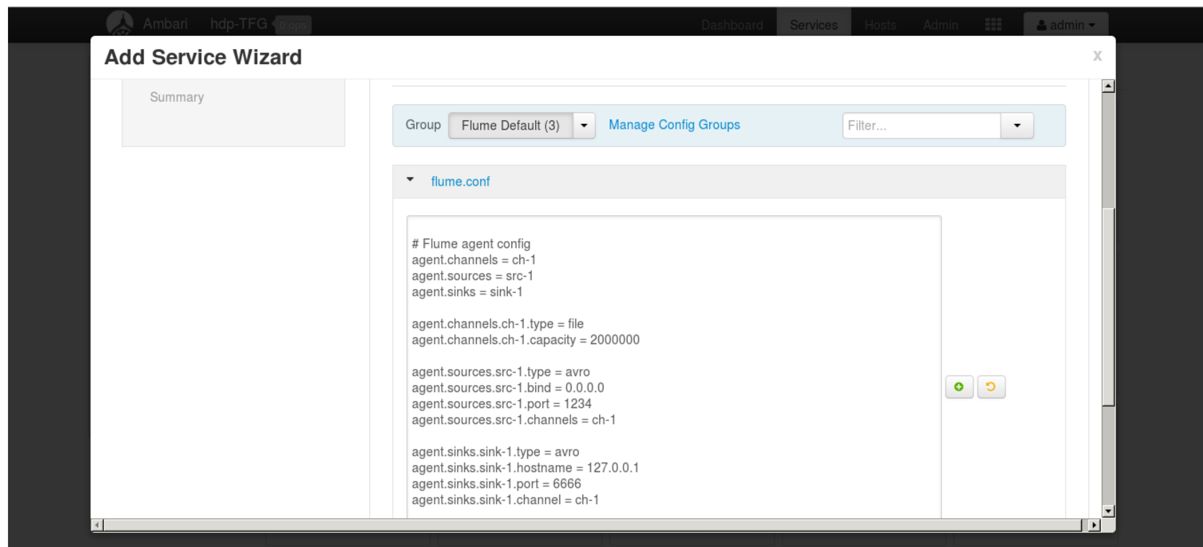


Figura 6.4: Configuración Flume

Por último, el asistente de configuración mostrará un recordatorio informando de que ciertos servicios se deben reiniciar tras la instalación de Flume.

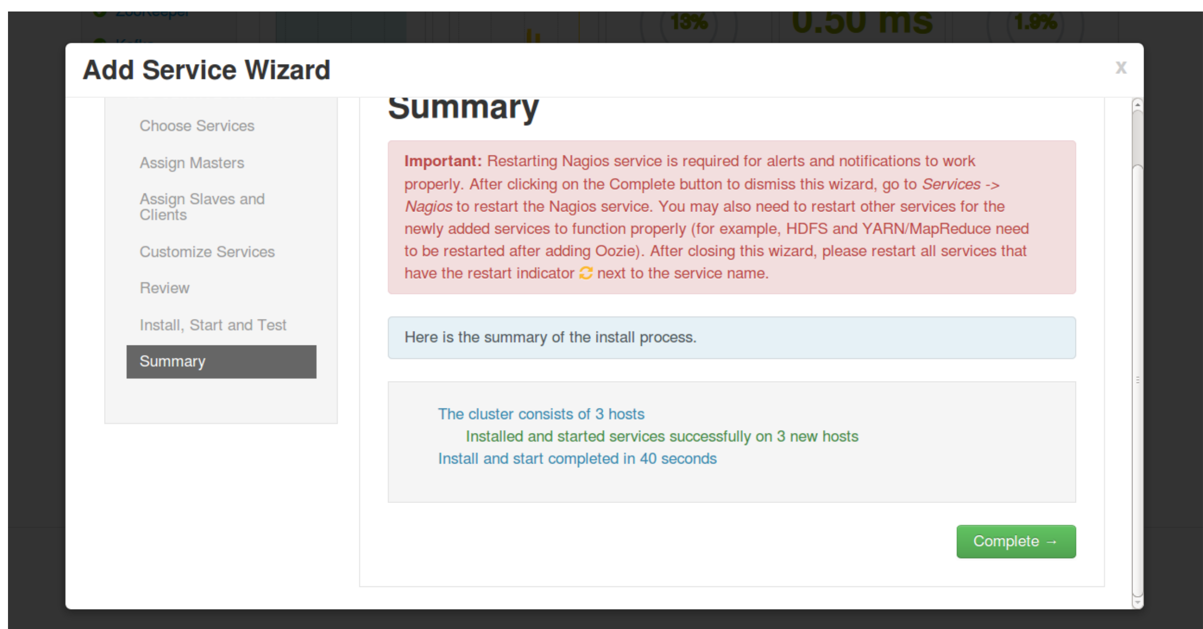


Figura 6.5: Instalación finalizada

6.2. Sistema de Dispensación Electrónica

A la hora de crear los contenedores que simularán el comportamiento del sistema de Dispensación Electrónica utilizaremos el siguiente fichero de definición para la creación de la imagen Docker.

```
#Nodos de DISEL
FROM ubuntu:14.04
RUN \
    apt-get update && \
    apt-get install -y openjdk-7-jre

ADD apache-flume-1.6.0-bin /usr/local/bin/apache-flume-1.6.0-bin
ADD disel.jar /usr/local/bin/
ADD flume.sh /usr/local/bin/apache-flume-1.6.0-bin/
ADD flumeDisel.conf /usr/local/bin/apache-flume-1.6.0-bin/

ENV JAVA_HOME /usr/lib/jvm/java-7-openjdk-amd64
```

Código 6.4

Esta imagen se basa en la imagen oficial de Ubuntu, concretamente la versión 14.04 y se instala el OpenJDK para poder ejecutar tanto Flume como la aplicación Java que registrará las nuevas líneas de log. A continuación se añaden los ficheros necesarios (que deben estar presentes en el directorio en el que se encuentra el fichero de definición de la imagen), describiremos estos ficheros a continuación:

- **apache-flume-1.6.0-bin:** se trata de la distribución de Flume utilizada.
- **disel.jar:** consiste en la aplicación que hemos desarrollado para simular el funcionamiento del sistema de Dispensación Electrónica tal y como hemos descrito en la sección de diseño. Como ya hemos comentado, los log se deben enviar a través de la herramienta log4j, para ello utilizaremos el siguiente fichero de configuración:

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="warn" name="DISEL" packages="">
  <Appenders>
    <Flume name="eventLogger" compress="false">
      <Agent host="127.0.0.1" port="8585"/>
      <PatternLayout pattern="%m%n"/>
    </Flume>
  </Appenders>
  <Loggers>
    <Root level="info">
      <AppenderRef ref="eventLogger"/>
    </Root>
  </Loggers>
</Configuration>
```

Código 6.5

De este modo se configura log4j de tal modo que las líneas de log se envían al agente Flume desplegado en *localhost* al puerto 8585.

- **flume.sh:** se trata de un script en el que se encuentra el siguiente comando:

```
/usr/local/bin/apache-flume-1.6.0-bin/bin/flume-ng agent
        --conf /usr/local/bin/apache-flume-1.6.0-bin/conf
        --conf-file flumeDisel.conf --name agent
```

```
-Dflume.root.logger=DEBUG,console
-Dflume.monitoring.type=http
-Dflume.monitoring.port=34545
```

Código 6.6

En este comando se utiliza el fichero de configuración que describiremos en el siguiente punto para desplegar el agente Flume. Además, con las últimas opciones se despliega de forma automática un servidor HTTP que atiende peticiones en el puerto 34545 y que devuelve información acerca de las estadísticas de ejecución del agente. Utilizaremos esta información en la sección de pruebas con el fin de validar el correcto funcionamiento del sistema.

- **flumeDisel.conf:** mediante este fichero se definen los componentes del agente Flume que se desplegará:

```
agent.channels = ch-1
agent.sources = src-1
agent.sinks = sink-1

agent.channels.ch-1.type = file
agent.channels.ch-1.capacity = 500000

agent.sources.src-1.type = avro
agent.sources.src-1.bind = 0.0.0.0
agent.sources.src-1.port = 8585
agent.sources.src-1.channels = ch-1

agent.sinks.sink-1.type = avro
agent.sinks.sink-1.hostname = amb0
agent.sinks.sink-1.port = 1234
agent.sinks.sink-1.channel = ch-1
```

Código 6.7

Definimos un canal que almacena los eventos en un fichero tal y como hemos especificado en el diseño, cumpliendo con el requisito RCA_002 que establece un periodo de un día de inactividad del clúster durante el que no se deben producir pérdidas de datos. En el apartado de análisis, a partir de los datos de log de los que disponemos, estimamos un volumen cercano a los 2.000.000 de líneas por día, por lo tanto, estableceremos un tamaño de canal que nos permita almacenar todos estos eventos, en concreto, dividiremos este volumen total entre los cuatro agentes con lo que tenemos un total de 500.000 eventos. Utilizaremos tanto una fuente como un sumidero que utilizan Avro para la serialización de los eventos recibidos de log4j y de los eventos enviados a la siguiente capa.

6.3. Sistema de procesamiento

El sistema de procesamiento representa el núcleo del proyecto que estamos desarrollando. Se trata de una aplicación Java que se ejecuta sobre Spark de forma distribuida en el clúster. En esta sección comentaremos alguno de los puntos más relevantes de la implementación de este sistema.

Con el siguiente código se crea el *DStream* con los datos de entrada provenientes de Flume y, a continuación, se ejecuta un *map* sobre el conjunto de datos de entrada que extrae las líneas de log de los eventos Flume que, como ya hemos comentado, se encuentran serializados mediante Avro.

```
        JavaReceiverInputDStream<SparkFlumeEvent> flumeStream = FlumeUtils
                .createStream(ssc, host, puerto);

        JavaDStream<String> lineasString = flumeStream
                .map(new Function<SparkFlumeEvent, String>() {
                    public String call(SparkFlumeEvent flumeEvent){
                        AvroFlumeEvent avroEvent = flumeEvent.event();
                        ByteBuffer bb = (ByteBuffer)
avroEvent.get(avroEvent
                                .getSchema().getField("body").pos());

                        String cadena = new String(bb.array());
                        return cadena;
                    }
                });
```

Código 6.8

Modificando este código podemos cambiar el origen de los datos y establecer, por ejemplo, HBase. De este modo podríamos implementar un componente encargado del reprocesado de los datos tal y como mencionábamos en el diseño. Cuando se libere una nueva versión del analizador, basta con desplegar el mismo programa con el método de entrada modificado para leer de HBase y poder tener los resultados que calcula la nueva versión a partir de todo el histórico que tenemos almacenado.

Realizando sucesivas transformaciones, tal y como se muestra en los diagramas propuestos en la sección de diseño, obtenemos el conjunto de datos resultado que debemos almacenar en HBase, para ello utilizamos el siguiente código:

```
        resultados.foreach(new Function<JavaRDD<Resultado>, Void>(){
                    public Void call(JavaRDD<Resultado> rdd) throws
        IOException{
                        DaoResultados dao = new DaoResultados();
                        for(Resultado r : rdd.collect())
                            dao.insertar(r);
                        return null;
                    }
                });
```

Código 6.9

En este código se itera sobre todos los RDD que forman el *DStream* y se almacena el contenido en HBase a través de la clase DaoResultados. Cabe destacar que este objeto de acceso a datos se instancia directamente en lugar de utilizar el patrón DAO en conjunto con *AbstractFactory* tal y como haríamos en una aplicación convencional. Como ya hemos comentado, Spark Streaming se basa en el concepto de *micro batch* con lo que, esta porción de código se ejecuta constantemente para pequeñas cantidades de datos. Con lo que se necesita que el proceso de instanciación de los recursos suponga el menor overhead posible. Una alternativa en una aplicación convencional sería instanciar el objeto fuera de esta sección de código y pasar directamente el objeto ya instanciado todas las veces que sea necesario. En Spark no podemos enviar este tipo de parámetros a las funciones que se ejecutan en las transformaciones o en las acciones (como es este caso) o utilizar objetos que se instancian fuera de la definición de dichas funciones. Esto es razonable puesto que tendremos una gran cantidad de nodos ejecutando estas funciones de forma transparente con lo que se podría producir condiciones de carrera que obligarían a adoptar medidas extra de gestión de concurrencia, como semáforos o mutex, que restarían flexibilidad a la hora de planificar las tareas dentro del clúster.

Esta aplicación la hemos desarrollado de tal forma que se puede configurar a través del fichero */compartida/resources/config.properties*. Este fichero tiene el siguiente formato:

```
#Conexión
host=”Dirección IP del Host en el que se ejecuta el agente Flume”
puerto=”Puerto que hemos establecido en la configuración del
sumidero (sink)”

#Duración
duracion=”Duración de cada microbatch en milisegundos”
```

Código 6.10

dónde *host* y *puerto* establecen los parámetros de conexión con el agente Flume y *duracion* establece la duración de cada *batch*, esto dependerá de las capacidades de cómputo disponibles y de la complejidad asociada al procesamiento. En nuestro caso estableceremos este valor a 5000, es decir 5 segundos.

Como ya hemos comentado, Spark ofrece la posibilidad de ejecutarse sobre un clúster gestionado con YARN como es el nuestro. Con el siguiente comando podemos ejecutar nuestra aplicación utilizando YARN como gestor de recursos:

```
spark-submit
      --class com.hp.ereceta.analizador.procesamientoLog.ProcesamientoLog
      --master yarn-cluster
      /compartida/analizador_1.0.jar
```

Comando 6.8

6.4. Definición de informes

A continuación definiremos el proceso de generación de la definición de informes con la herramienta Pentaho Report Designer, posteriormente cargaremos estas definiciones desde la aplicación web para poder generar los informes finales que se presentarán al usuario.

Desde la ventana principal seleccionamos *File* > *New* y se abrirá una pestaña con una nueva definición de un informe en blanco tal y como se muestra en la siguiente captura de pantalla:

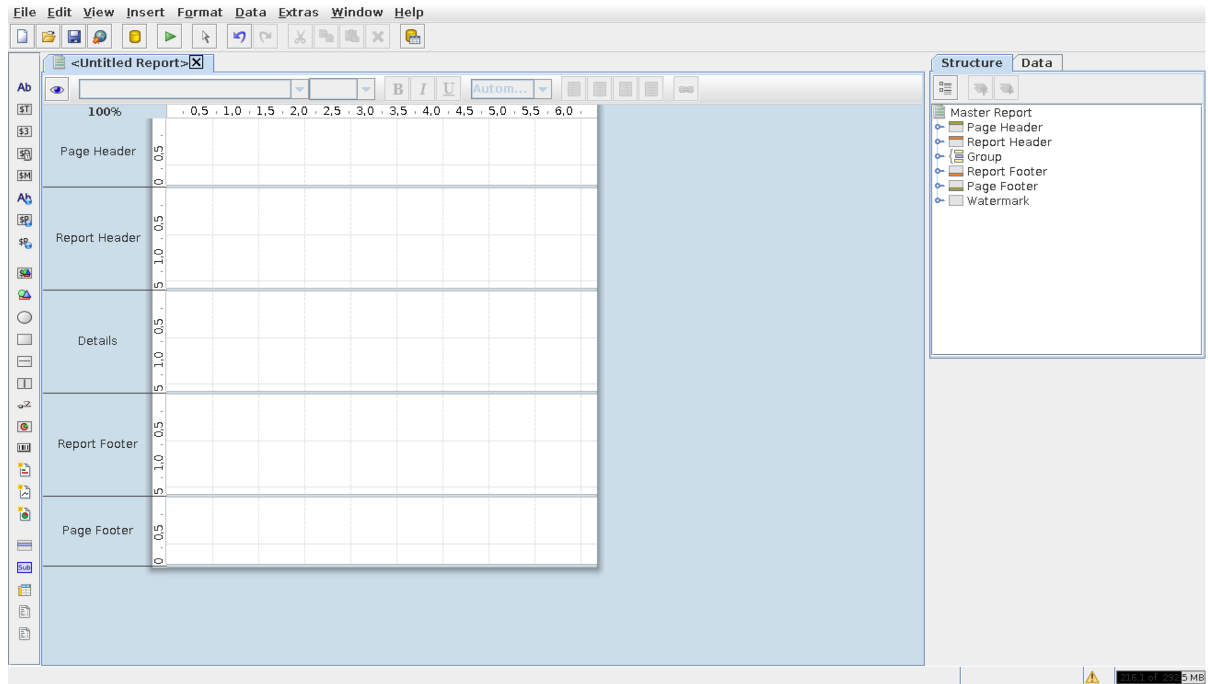
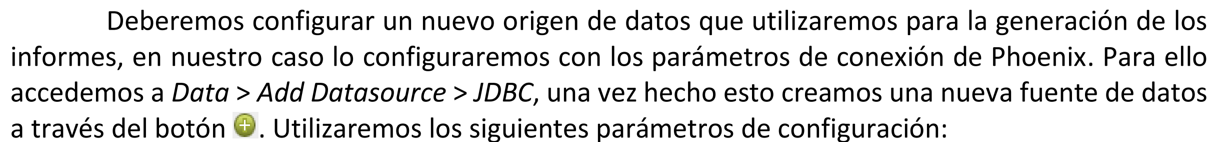


Figura 6.6: Ventana principal Pentaho Report Designer

Deberemos configurar un nuevo origen de datos que utilizaremos para la generación de los informes, en nuestro caso lo configuraremos con los parámetros de conexión de Phoenix. Para ello accedemos a *Data* > *Add Datasource* > *JDBC*, una vez hecho esto creamos una nueva fuente de datos a través del botón . Utilizaremos los siguientes parámetros de configuración:

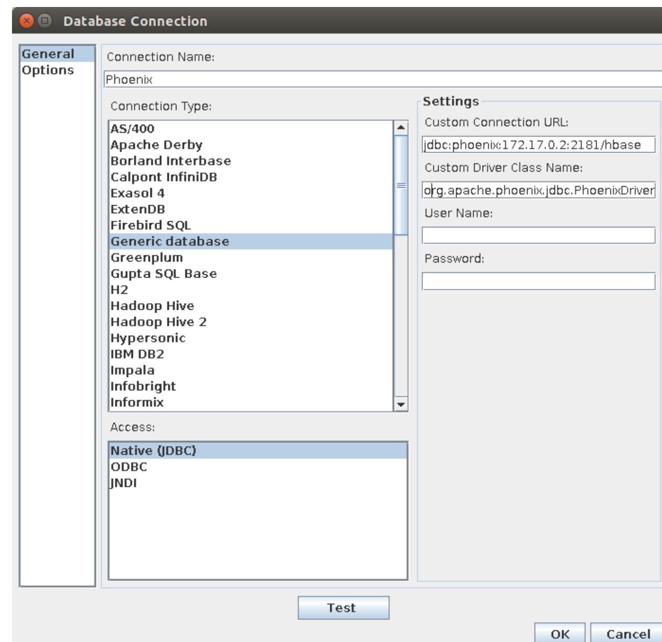


Figura 6.7: Configuración Datasource

Por último, necesitamos añadir el *driver* de Phoenix para que Pentaho Report Designer pueda establecer la conexión y encontrar la clase *org.apache.phoenix.jdbc.PhoenixDriver* que hemos establecido en la ventana anterior. Para ello, copiamos el fichero *phoenix-client.jar* que podemos encontrar en la carpeta de instalación de Phoenix dentro de alguno de los nodos de clúster Hadoop y lo copiamos en la carpeta de instalación de Pentaho Report Dsigner en la ruta *lib/jdbc/*. De este modo

ya tendremos acceso a los datos que se encuentran almacenados en el clúster Hadoop, únicamente necesitaremos definir las consultas SQL a través de las cuales obtendremos los datos para la generación de nuestros informes. En la ventana de creación de un nuevo *Datasource* que vemos a continuación también tenemos disponible un editor SQL que nos permite asociar consultas a cada *Datasource*.

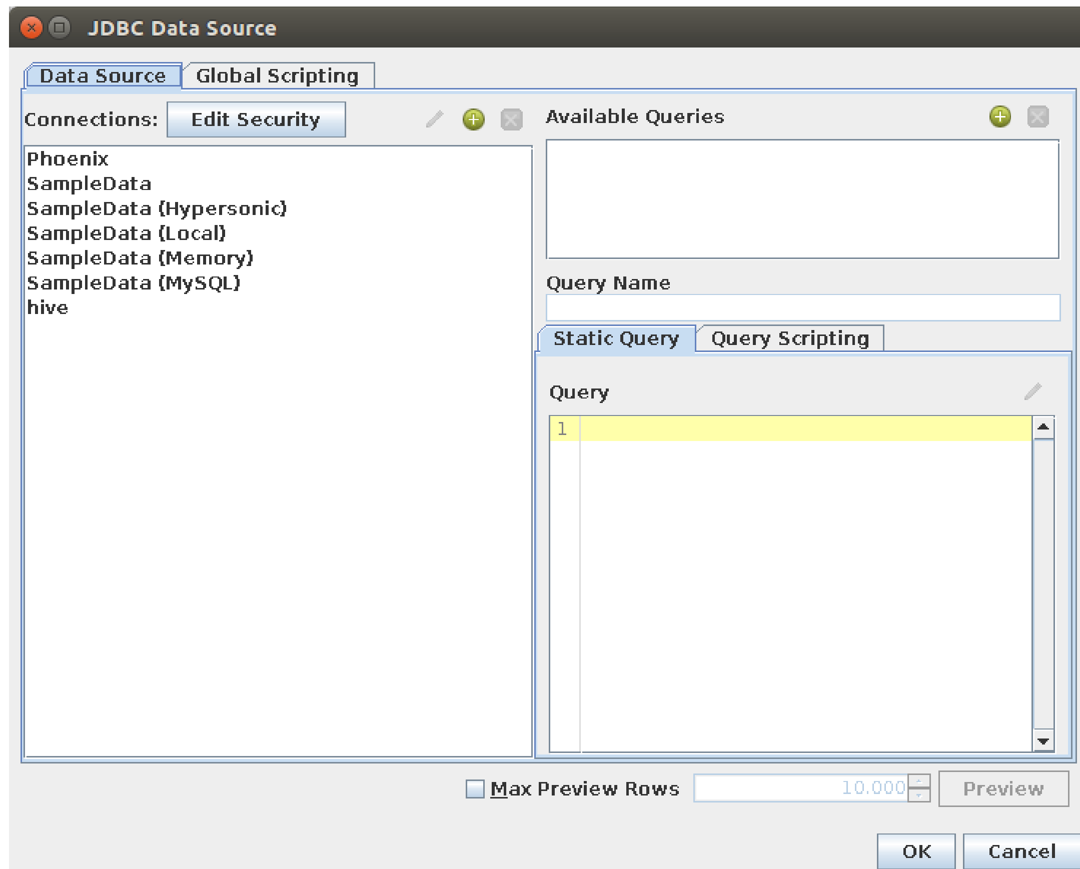


Figura 6.8: Editor SQL

Para el primer informe que recoge el número de peticiones por unidad de tiempo definiremos dos consultas, una para la gráfica de actividad para el día completo y otra únicamente para la hora con el resumen de peticiones por minuto. En el primer caso la consulta es la siguiente:

```
SELECT "hora", COALESCE(SUM(R."npet"),0) AS "peticiones"
FROM (SELECT "fecha", "npet"
      FROM "result"
      WHERE "fecha" > trunc(CURRENT_TIME(), 'DAY')) AS R
      RIGHT JOIN "todasHoras"
      ON "hora" = CAST(to_number(to_char(R."fecha", 'HH')) AS
INTEGER)%24
GROUP BY "hora"
```

Código 6.11

Esta consulta utiliza una tabla auxiliar denominada *todasHoras* que mantiene una fila por cada una de las horas del día, esto nos permite establecer a 0 aquellas horas para las que no hay peticiones registradas y no provocar discontinuidades en la gráfica del informe. Las limitaciones del lenguaje SQL soportado por Phoenix, actualmente, hace que sea necesaria la utilización de este tipo de tablas auxiliares para algunos tipos de consultas.

En el caso de la segunda consultas se aplica un razonamiento similar utilizando una tabla denominada *todosMinutos*.

```
SELECT "minuto" AS "minuto", COALESCE(SUM(R."npet"),0) AS
"peticiones"
FROM (SELECT "fecha", "npet"
      FROM "result"
      WHERE "fecha" > trunc(CURRENT_TIME(), 'HOUR')) AS R
      RIGHT JOIN "todosMinutos" ON "minuto" =
to_number(to_char(R."fecha", 'mm'))
GROUP BY "minuto"
```

Código 6.12

Para el segundo informe que recoge el resumen de número de peticiones por operación y el tiempo de procesamiento de dicado a cada una de ellas utilizaremos la siguiente consulta:

```
SELECT "operacion", SUM("npet") AS "peticiones", SUM("tproc") AS
"tiempo",
      AVG("tproc"/"npet") AS tmedio
FROM "result"
WHERE "fecha" BETWEEN to_date(${fechaInicio}, 'yyyy-MM-dd HH:mm:ss')
              AND to_date(${fechaFin}, 'yyyy-MM-dd HH:mm:ss')
GROUP BY "operacion"
ORDER BY "peticiones" DESC
```

Código 6.13

Esta consulta se encuentra parametrizada por los argumento *fechaInicio* y *fechaFin*. De este modo podremos pasar estos parámetros desde la aplicación web de tal forma que el usuario podrá definir el periodo de tiempo sobre el que se realizará el resumen.

Por último, únicamente deberemos añadir los componentes gráficos deseados que tenemos disponibles en la paleta situada en la parte izquierda de la ventana principal. Almacenaremos estas definiciones en ficheros *.prpt* que posteriormente cargaremos desde la aplicación web con las utilidades de Pentaho Reporting Engine SDK [40].

En todo caso, siguiendo el mismo procedimiento que hemos definido para la generación de las definiciones de estos informes podemos generar informes *ad hoc* totalmente personalizados, pudiendo aprovechar las posibilidades de exportación a diversos formatos como PDF o HTML que ofrece Pentaho Report Designer cumpliendo con lo establecido en el requisito RAP_003. Estos informes se podrán crear a partir de la tabla con los resultados preprocesados o a partir de los datos completos que tenemos almacenados en HBase, en este caso sería necesario crear nuevas vistas en Phoenix que den acceso a estos datos a través de SQL.

La aplicación web se desplegará en un contenedor Docker basado en la imagen definida por el siguiente fichero de definición:

```
FROM tomcat:8.0
ADD Reporting.war /usr/local/tomcat/webapps/
```

Código 6.14

Como se puede ver, se trata de una imagen muy simple basada en la imagen oficial de Tomcat 8.0. Únicamente añadiremos la aplicación web en la ruta correspondiente para que se despliegue en

el contenedor de aplicaciones. En primer lugar debemos crear la imagen con el siguiente comando en la carpeta en la que se encuentra el fichero anterior:

```
$ docker build -t reporting:1.0 .
```

Comando 6.9

Para ejecutarlos únicamente necesitamos el siguiente comando y el servidor de aplicaciones se ejecutará en el contenedor:

```
$ docker run -it reporting:1.0
```

Comando 6.10

7. Pruebas

En este capítulo mostraremos los resultados obtenidos tras la ejecución del sistema simulando un entorno real, empezaremos definiendo el escenario de pruebas con el fin de permitir que las pruebas sean repetibles y se puedan aplicar sobre nuevas versiones. Por último, comentaremos los resultados obtenidos.

7.1. Definición del escenario de pruebas

A continuación describiremos el escenario sobre el que se ejecutarán las pruebas.

7.1.1. Equipo de pruebas

Para la realización de las pruebas utilizaremos un equipo con las siguientes características:

- **Procesador:** Intel Core i7-2630QM
- **Memoria RAM:** 8GB
- **Disco duro:** 128GB SSD
- **Sistema Operativo:** Ubuntu 13.10

En este sentido nos ajustamos al requisito RAP_005 que impone una restricción en la cantidad de memoria disponible de 8GB. En lo que se refiere a capacidad de disco no tendremos ningún problema puesto que únicamente manejamos datos generados durante unas horas, en un entono real, probablemente necesitaríamos varios terabytes para almacenar datos históricos de varios años.

7.1.2. Datos de prueba

Como ya hemos comentado, contamos con un conjunto de log reales (anonimizados) del sistema de Dispensación Electrónica que utilizaremos para simular su funcionamiento y poder probar nuestro sistema de procesamiento de log en una situación real. Contamos con un conjunto de log reducido, concretamente tenemos logs desde las 22:04 hasta las 13:47 del día siguiente, esta restricción se debe a que se ha producido un error en el sistema en el momento en el que se han generado los logs que debían ser utilizados en este proyecto. No obstante, esto supone ciertas ventajas como la posibilidad de probar nuestro sistema en una situación de fallo del sistema con lo que podemos ver si efectivamente se refleja este hecho en los resultados obtenidos.

En total, contamos con 547.903 líneas. Cabe destacar que este conjunto de datos contiene una serie de líneas de log que no cumplen con ninguno de los formatos que hemos definido, de este modo también comprobaremos el comportamiento del sistema ante líneas que no cumplen con un formato conocido. Esta circunstancia ya ha sido prevista en la etapa de diseño y se ha propuesto un tratamiento especial para estos casos. Estas nuevas líneas tienen el formato siguiente:

Fecha | *INFO* | *plataforma* | id | *Peticion Echo DISEL NCD*

Como podemos ver se trata de una línea en la que, en los últimos campos, no se incluye el separador "|". Esto no se adapta al formato de *Petición* que hemos definido en el análisis del sistema de Dispensación Electrónica.

7.2. Capa de recolección

Las estadísticas de ejecución que nos ofrecen los agentes Flume son de vital importancia a la hora de validar el funcionamiento del sistema y ver si existe un cuello de botella o se están perdiendo datos en algún punto. Para ello analizaremos el número de eventos que se almacenan en el canal a lo largo de toda la ejecución de las pruebas. Un número alto de eventos en el canal nos indica que se están recibiendo un mayor número de eventos de los que se pueden enviar al siguiente componente. Además, también analizaremos el número de eventos enviados correctamente con el fin de poder ver si en algún punto no se están retransmitiendo la totalidad de los datos. Para obtener estos datos utilizaremos dos aproximaciones diferentes. Por un lado tenemos el agente que se ejecuta en el clúster Hadoop. Para obtener los datos de ejecución de este agente utilizaremos los datos obtenidos por el sistema de monitorización Ganglia [45] que tenemos instalado junto a Ambari. Por otro lado, tenemos los agentes que se despliegan en los nodos del sistema de Dispensación Electrónica. En este caso no contamos con Ganglia, en su lugar emplearemos las utilidades de *monitoring* que ofrece Flume a través de HTTP. Hemos desarrollado una aplicación Java que realiza, de forma periódica, estas peticiones HTTP y escribe los datos que devuelve Flume en un fichero. Posteriormente, utilizaremos un script escrito en Python para generar las gráficas que mostraremos en este documento.

En la siguiente figura vemos la evolución del número de eventos que se mantienen en el canal de los agentes Flume que se despliegan en los cuatro nodos del sistema de Dispensación Electrónica. Recordar que estos agentes reciben los datos directamente del sistema de Dispensación Electrónica y los envían a otro agente Flume que se encuentra en el clúster Hadoop. Como se puede ver, se alcanza un máximo de 12 eventos, muy lejos de los 500.000 que hemos establecido como capacidad máxima. Esto nos indica que la tasa de envío es lo suficientemente alta en relación a la tasa de recepción. Vemos como alcanzamos los niveles máximos entorno a las 10:00, hora en la que, como veremos, la carga del sistema aumenta considerablemente con respecto a la soportada durante la noche.

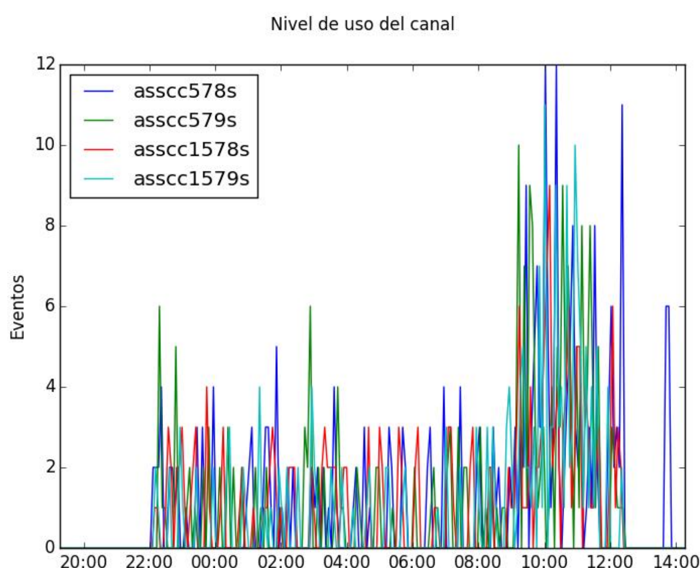


Figura 7.1: Gráfica uso del canal

En la siguiente gráfica vemos el acumulado de eventos enviados para cada uno de los cuatro agentes anteriores. En esta gráfica podemos confirmar que el mayor nivel de uso de los canales se corresponde con periodos en los que se reciben un mayor número de líneas de log. A partir de las 10:00 tenemos un considerable incremento del número de líneas enviadas que se prolonga hasta las

12:00, dónde podemos ver como alguno de los nodos deja de emitir nuevas líneas mientras que otros siguen enviando líneas hasta cerca de las 14:00. Hasta las 12:00 tenemos un balanceo de carga muy ajustado entre los diferentes nodos, a partir de esta hora se puede apreciar una mayor variabilidad provocada por la situación anómala que comentábamos en la descripción de los datos de prueba. Esto también se refleja en la gráfica anterior, puesto que el agente desplegado en el nodo *asscc578s* sigue teniendo eventos en el canal dos horas después de que los demás hayan dejado de utilizarlo.

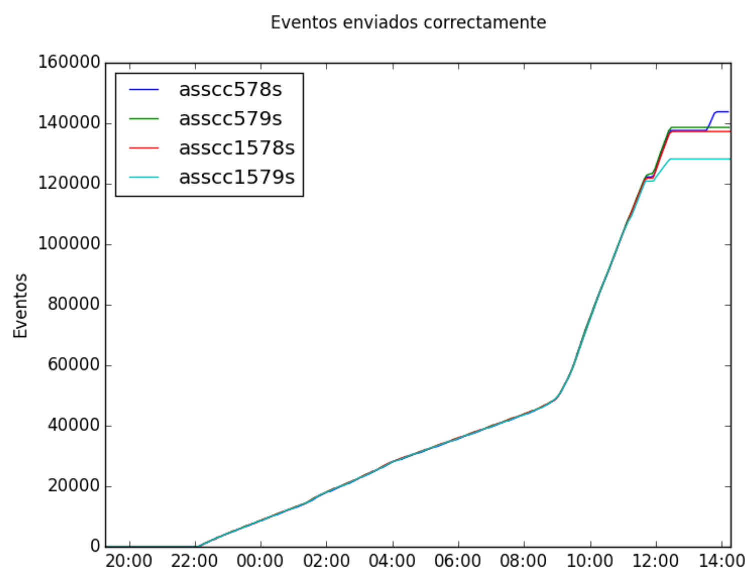


Figura 7.2: Gráfica de eventos enviados

Sumando los valores finales de los cuatro nodos obtenemos el total de líneas enviadas correctamente que es 547.903 y que coincide exactamente con el volumen de datos de entrada con lo que podemos comprobar que no se han perdido datos en este punto. Además, también nos indica que la aplicación que hemos desarrollado para simular el funcionamiento del sistema de Dispensación Electrónica está funcionando correctamente.

A continuación analizaremos las estadísticas del agente que se ejecuta en el clúster Hadoop. En la figura 7.3 tenemos el nivel de uso del canal. Como podemos ver, la gráfica sigue una forma similar a la de los agentes desplegados en los nodos del sistema de Dispensación Electrónica alcanzando valores más altos. Esto se debe a que por este agente se retransmite la totalidad de los datos mientras que los anteriores únicamente soportaban aproximadamente un cuarto de la carga total. En todo caso, seguimos teniendo unos valore máximos muy reducidos y totalmente asumibles, que indican que los eventos se están entregando a la capa de procesamiento prácticamente al mismo tiempo que se reciben.

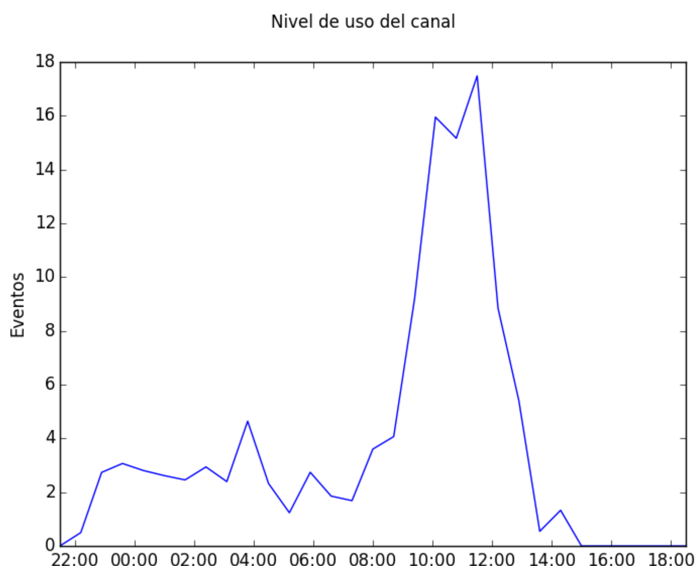


Figura 7.3: Uso canal Flume en Hadoop

A continuación volvemos a tener el valor acumulado de eventos enviados correctamente, en este caso, al tratarse del agente encargado de agregar todos los datos en un único flujo, consiste en el total de líneas generadas. Esta gráfica alcanza su máximo en 547.903, número total de líneas de las que disponíamos con lo que podemos ver que se han entregado todas las líneas a la capa de procesamiento.

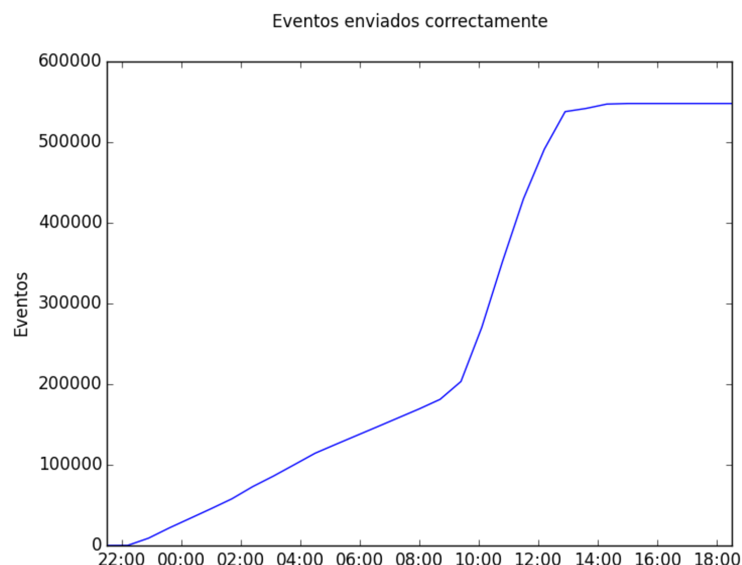


Figura 7.4: Eventos enviados Flume en Hadoop

Con los datos aportados podemos ver como la capa de recolección de nuestra arquitectura no solo es robusta, puesto que ante un fallo puede almacenar en los respectivos canales los eventos, sino que también ofrece un rendimiento aceptable. Este punto era clave puesto que la utilización de un canal que mantiene los eventos en almacenamiento persistente en lugar de en memoria principal podría hacer el sistema demasiado lento a la hora de entregar de los eventos.

7.3. Capa de procesamiento

Inspeccionaremos los ficheros de log producidos por Spark Streaming durante la ejecución de las pruebas. Revisando los datos únicamente encontramos mensajes de advertencia acerca de líneas desconocidas, es decir que no cumplen con ninguno de los formatos preestablecidos, concretamente se recogen advertencias sobre 7.790 líneas que coinciden exactamente con el número de líneas que siguen el formato de petición diferente comentado en la definición de los datos de prueba.

En el código, en el tratamiento de las excepciones, se registran líneas de error que no aparecen en el fichero, esto nos indica que la única situación anómala prevista que ha sucedido es la aparición de líneas no esperadas. En todo caso, como ya hemos comentado, estas líneas se almacenan en la base de datos indicando *"Desconocida"* como su tipo.

7.4. Resultados

A continuación mostraremos dos capturas de pantalla de la aplicación web, una por cada uno de los informes que hemos definido. Esto nos valdrá para validar el correcto funcionamiento tanto de la capa de acceso a datos como de la aplicación web.

En primer lugar, tenemos el informe de carga el sistema, como se puede ver, hemos incluido las dos gráficas que se especifica en el diseño. Por un lado, tenemos el número de peticiones por franja horaria. Por otro lado, tenemos el número de peticiones por minuto en la última hora, las capturas las hemos obtenido a las 14:00, justo a continuación de los últimos datos que tenemos disponibles.

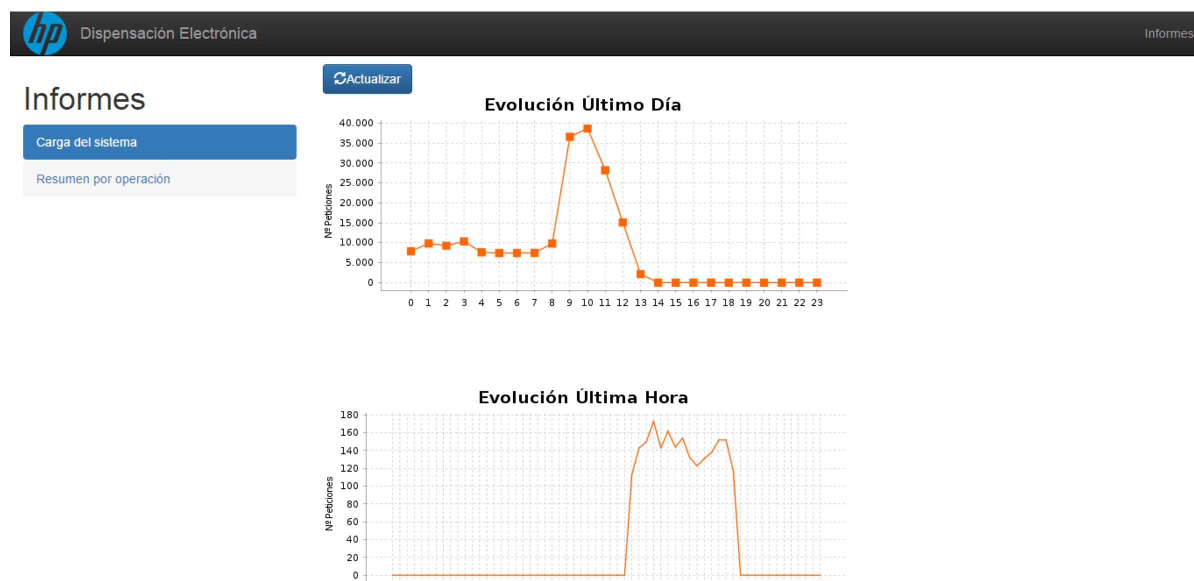
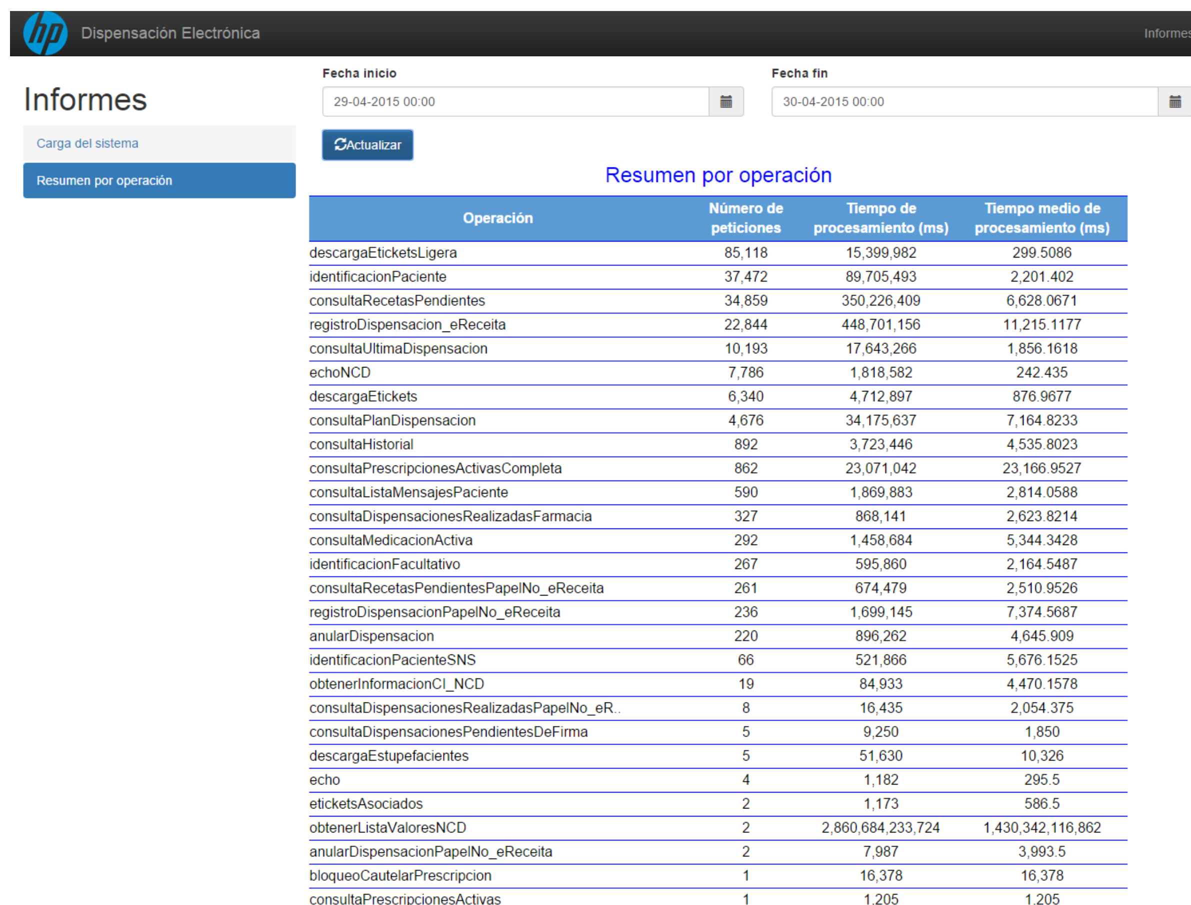


Figura 7.5: Informes de número de peticiones por unidad de tiempo

Como ya anticipábamos en el análisis de la ejecución de los agentes Flume entre las 9:00 y 10:00 se produce un importante pico en el número de peticiones procesadas por el sistema de Dispensación Electrónica pasando de una **10.000 peticiones/hora** durante la noche a **unas 40.000 peticiones/hora**. A continuación, vemos como el número de peticiones se desploma, esto se debe a la situación anómala que venimos comentado. En la gráfica de uso durante la última hora podemos ver como apenas existe actividad con un pico de alrededor de **180 peticiones/minuto**, si cogemos las 40.000 peticiones/hora de que teníamos antes de que se desplomará la actividad obtenemos un total de más de **600 peticiones/minuto** en promedio. Esto nos da una idea de la baja actividad a la que se

estaba sometiendo el sistema en esta franja horaria en la que, como hemos visto, únicamente estaba emitiendo nuevas líneas uno de los cuatro nodos del sistema de Dispensación Electrónica.

A continuación tenemos el informe en el que se muestra el número de peticiones recibidos por cada tipo de operación, el tiempo empleado en procesarlas y el tiempo promedio en milisegundos. Hemos seleccionado un intervalo de tiempo entre las 00:00 del 29/04/2015 y las 14:00 del 30/04/2015.



Operación	Número de peticiones	Tiempo de procesamiento (ms)	Tiempo medio de procesamiento (ms)
descargaEticketsLigera	85,118	15,399,982	299.5086
identificacionPaciente	37,472	89,705,493	2,201.402
consultaRecetasPendientes	34,859	350,226,409	6,628.0671
registroDispensacion_eReceita	22,844	448,701,156	11,215.1177
consultaUltimaDispensacion	10,193	17,643,266	1,856.1618
echoNCD	7,786	1,818,582	242.435
descargaEtickets	6,340	4,712,897	876.9677
consultaPlanDispensacion	4,676	34,175,637	7,164.8233
consultaHistorial	892	3,723,446	4,535.8023
consultaPrescripcionesActivasCompleta	862	23,071,042	23,166.9527
consultaListaMensajesPaciente	590	1,869,883	2,814.0588
consultaDispensacionesRealizadasFarmacia	327	868,141	2,623.8214
consultaMedicacionActiva	292	1,458,684	5,344.3428
identificacionFacultativo	267	595,860	2,164.5487
consultaRecetasPendientesPapelNo_eReceita	261	674,479	2,510.9526
registroDispensacionPapelNo_eReceita	236	1,699,145	7,374.5687
anularDispensacion	220	896,262	4,645.909
identificacionPacienteSNS	66	521,866	5,676.1525
obtenerInformacionCI_NCD	19	84,933	4,470.1578
consultaDispensacionesRealizadasPapelNo_eR..	8	16,435	2,054.375
consultaDispensacionesPendientesDeFirma	5	9,250	1,850
descargaEstupefacientes	5	51,630	10,326
echo	4	1,182	295.5
eticketsAsociados	2	1,173	586.5
obtenerListaValoresNCD	2	2,860,684,233,724	1,430,342,116,862
anularDispensacionPapelNo_eReceita	2	7,987	3,993.5
bloqueoCautelarPrescripcion	1	16,378	16,378
consultaPrescripcionesActivas	1	1,205	1,205

Figura 7.6: Resumen por tipo de operación

En estos datos es especialmente relevante el caso de la operación *obtenerListaValoresNCD*. Podemos ver un tiempo medio de procesamiento por petición extremadamente alto que representa un total de 45 años. Sin duda se trata de un dato atípico probablemente provocado por un error el sistema de Dispensación Electrónica a la hora de registrar los datos de estas operaciones y que puede estar relacionado con toda la actividad atípica que venimos comentando. Obviando este caso, tenemos que la operación de la que más peticiones se reciben es *descargaEticketsLigera* mientras que la operación que abarca un mayor tiempo de procesamiento total es *registroDispensacion_eReceta.* También podemos ver como *consultaPrescripcionesActivasCompleta* es la operación que más tarda en ser procesada en promedio.

Los dos informes que hemos comentando evidencian claramente una situación irregular en la que el sistema de Dispensación Electrónica ha registrado una actividad que se aleja de una ejecución habitual. En todo caso, no disponemos de estadísticas de ejecución durante periodos de tiempo en los que no se ha producido ningún error que nos permitan obtener un análisis más profundo permitiendo comparar estos informes con alguno cuya actividad no se salga de lo normal.

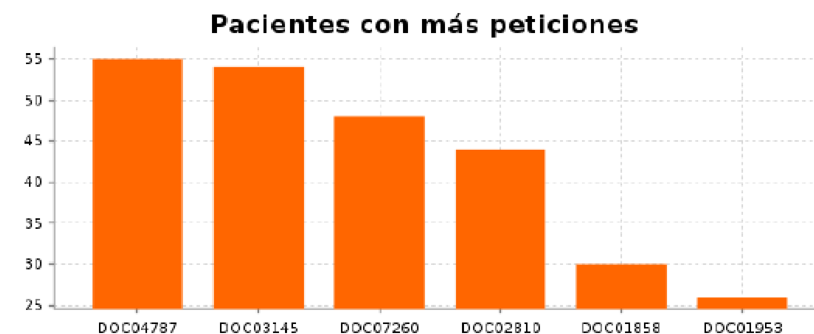
Con el fin de comprobar que efectivamente se cumple con lo especificado en el requisito RDI_002 crearemos un nuevo informe totalmente diferente de los que hemos comentado. Para ello definiremos una nueva vista en Phoenix que nos permita acceder a las líneas que tienen datos asociados acerca de la farmacia y los pacientes con los que se relaciona la operación. Crearemos esta vista con la siguiente sentencia SQL:

```
CREATE VIEW "log" (
    "fecha" UNSIGNED_TIMESTAMP NOT NULL,
    "operacion" VARCHAR NOT NULL,
    "fmcia" VARCHAR,
    "fmtico" VARCHAR,
    "tdoc" VARCHAR,
    "doc" VARCHAR,
    CONSTRAINT pk PRIMARY KEY("fecha", "operacion")
)default_column_family='campos';
```

Código 7.1

A partir de estos datos generaremos el informe que se muestra en la figura 7.7 y que recoge un análisis de las peticiones relacionadas con cada paciente. En primer lugar, mostramos un gráfico con los 5 pacientes que han originado un mayor número de peticiones, a continuación, se muestra una tabla en la que se pueden ver los pacientes que han generado peticiones desde un mayor número de farmacias diferentes.

Resumen de peticiones por paciente



Número de farmacias diferentes visitadas por paciente

Paciente	Farmacias diferentes
DOC03145	30
DOC02810	23
DOC05935	12
DOC13412	6
DOC11305	5
DOC00021	5
DOC03055	5
DOC21750	4
DOC14194	3
DOC11503	3
DOC10567	3
DOC05190	3

Figura 7.7: Informe ad hoc

Como se puede ver, se trata de un informe radicalmente diferente a los anteriores y que utiliza el conjunto de datos originales en lugar de los datos preprocesados. Además, el interés principal de este informe no es ver si la actividad que está registrando el sistema de Dispensación Electrónica es adecuado sino que se centra más en la faceta de análisis social acerca de cómo los pacientes interactúan con este sistema. Esto muestra el amplio rango de casos de aplicación que tiene este proyecto más allá de comprobar el correcto funcionamiento del sistema de Dispensación Electrónica.

Haciendo un breve análisis de este último informe podemos ver como existe un paciente que ha acudido a 30 farmacias diferentes. Para generar este informe estamos teniendo en cuenta la totalidad de los datos de los que disponemos y que apenas abarcan unas 14 horas. Vemos como se trata de un número de farmacias diferentes bastante elevado en relación al periodo de tiempo que estamos analizando, esto también lo podemos aplicar tanto al segundo paciente de la lista como al tercero. Para poder sacar conclusiones tendríamos que realizar un estudio en mayor profundidad en busca, por ejemplo, de alguna correlación entre las farmacias visitadas o las franjas horarias en que se han dado estas visitas. También podemos ver como el paciente que más peticiones ha originado, DOC04787, no aparece en la lista de pacientes con mayor número de farmacias visitadas, esto nos indica que en muy pocas farmacias (como mucho en 3) se han realizado un gran número de peticiones acerca de este paciente. Al igual que el caso anterior sería necesario un análisis más profundo para

poder extraer alguna conclusión. En todo caso, el objetivo de este informe es demostrar que se puede extraer una gran variedad de información de nuestro sistema más allá de los informes que hemos previsto en este proyecto.

La generación de nuevos informes tiene una penalización importante con respecto a los que ya están definidos puesto que deben acceder a la totalidad de los datos en lugar de a un pequeño subconjunto de resultados preprocesados. En los datos que hemos manejado para la realización de estas pruebas hemos pasado de un total de 547.903 filas (una por cada una de las líneas de log) a 5.841 filas en la tabla de resultados preprocesados. Esto supone que acceder a los datos preprocesados representa alrededor de un 1% con respecto a los datos originales. A la necesidad de acceder a un número mucho mayor de datos, en estos nuevos informes, tenemos que sumar el tiempo empleado en realizar todas las computaciones asociadas a las funciones de agregación que ya tenemos calculadas en la tabla de resultados.

Por la naturaleza de estas pruebas, no se ha probado el sistema durante periodos largos de tiempo como pueden ser varios día o incluso semanas que pueden evidenciar algún tipo de problema no detectado. Además, en un entorno real, la carga que debe soportar será sensiblemente mayor ya que, potencialmente, tendremos un gran número de clientes realizando varias peticiones en paralelo tanto de informes predefinidos como de nuevos informes con el consiguiente incremento de los recursos necesarios. Todo esto hace que sea conveniente la realización de una minuciosa monitorización sometiendo al sistema a una situación de carga similar a la que nos encontraremos en un entorno de producción con el fin de asegurar el correcto funcionamiento.

8. Conclusiones

A pesar de que, como ya hemos comentado, sería necesario un periodo de pruebas mayor y más ambicioso en el que se pudiera someter al sistema a una situación de carga real durante un mayor periodo de tiempo, podemos concluir que se han alcanzado los objetivos originales al desarrollar un sistema de procesamiento de log en tiempo real plenamente funcional. Más allá del software concreto desarrollado, hemos llevado a cabo un análisis tecnológico y una propuesta de diseño de una arquitectura mucho más general aplicable a una gran variedad de sistemas de procesamiento Big Data en tiempo real. Una de las fortalezas del sistema propuesto radica precisamente en su versatilidad y facilidad de adaptación a nuevas características y requisitos sin tener que realizar grandes cambios tal y como comentaremos en la siguiente sección dedicada exclusivamente a las posibles ampliaciones.

La propuesta de una arquitectura alternativa, basada en los mismos principios que la arquitectura lambda, nos ha permitido integrar diversas herramientas del ecosistema Hadoop para formar un sistema robusto y mantenible. Esta labor de diseño e integración representa uno de los principales valores del presente proyecto puesto que es susceptible de ser reutilizado en futuros desarrollos con la consiguiente reducción de los costes asociados.

En lo que se refiere a la capa de almacenamiento, hemos conseguido integrar todas las ventajas asociadas a una base de datos NoSQL, como es HBase, sin dejar de proporcionar una interfaz SQL de acceso a datos de cara a los clientes. Esto nos ha permitido conseguir un sistema de almacenamiento de alto rendimiento sin renunciar a la gran compatibilidad con terceros sistemas y de capacidad de análisis que ofrece el estándar SQL. Conseguimos así utilizar una herramienta propia del *business intelligence* tradicional como es Pentaho Report Designer en un entorno Big Data en el que los datos se encuentran en una base de datos no relacional.

Más allá del procesamiento en tiempo real con Spark Streaming, Spark está ganando en popularidad y se está convirtiendo en una alternativa de futuro a MapReduce como principal sistema de procesamiento *batch* en Hadoop. Un claro ejemplo lo tenemos en los objetivos marcados desde el equipo de desarrollo de Hive que planean utilizar Spark como nuevo motor de ejecución de consultas HiveQL. Esto, unido al gran crecimiento que están experimentando otros componentes como Spark SQL y MLlib hace que las posibilidades de esta plataforma crezcan enormemente a corto y medio plazo. Todas estas circunstancias dan especial valor al análisis realizado sobre Spark puesto que las expectativas actuales indican que el potencial de ser utilizado en nuevos proyectos es muy alto. Precisamente, la capa de procesamiento representa el núcleo del sistema que hemos desarrollado y se beneficia de las funcionalidades que ofrecen todas las demás herramientas que hemos utilizado. En este sentido, la integración de Spark Streaming con Flume y HBase consiste en una de las claves de cualquier sistema de procesamiento en tiempo real en entornos Hadoop.

En lo que se refiere al envío de datos, hemos analizado y comparado las dos principales alternativas que existen actualmente: Flume y Kafka. El análisis de las principales ventajas que aporta cada uno de ellos, así como la experiencia adquirida al utilizar Flume en un sistema concreto, podrá ser de especial relevancia en futuros proyectos a la hora de seleccionar la herramienta más adecuada minimizando los riesgos asociados a la selección tecnológica.

La utilización de un sistema de virtualización ligero como es Docker nos ha permitido desarrollar un sistema con un consumo de recursos muy asumible que puede ser susceptible de ejecutarse directamente en los mismos nodos que el sistema principal. Se trata de un sistema auxiliar cuya misión es la monitorización de un sistema principal, con lo que el consumo de recursos se debe mantener lo más ajustado posible ya que no es aceptable un sistema de monitorización cuyos costes superen a los del propio sistema que se está monitorizando. En este sentido, la posibilidad de que

ambos sistemas se ejecuten en un mismo entorno dependerá en gran mediad del nivel de utilización de los recursos disponibles por parte del sistema principal.

La constante evolución de todo el ecosistema Hadoop, especialmente de alguno de los proyectos que hemos utilizado como puede ser Phoenix que aún no cuentan con un gran nivel de madurez, abren nuevas alternativas de futuro a la hora de aplicar el sistema propuesto a un mayor rango de casos de aplicación. Esto es especialmente aplicable al caso de Docker cuya creciente popularidad puede hacer que se convierta en una de las opciones más relevantes a la hora de desplegar servicios en la nube.

Más allá de todos los conocimientos técnicos adquiridos tras el aprendizaje de un gran número de tecnologías y paradigmas nuevos, desde el punto de vista de la realización de un Trabajo Fin de Grado, es especialmente relevante la posibilidad de aplicar gran parte de los conocimientos y destrezas adquiridos durante el Grado en un proyecto de una envergadura considerable. En este sentido, cobran especial relevancia todas aquellas habilidades transversales a la gran mayoría de proyectos software como pueden ser las propias de la gestión de proyectos y el desarrollo de software de forma sistemática y con garantías. El hecho de tratarse de un proyecto con vocación de ser aplicado en un entorno de producción real, es especialmente enriquecedor para el alumno al permitir enfocar todas estas habilidades en la creación de un producto que pueda cubrir una necesidad real.

8.1. Ampliaciones

Una de las principales ampliaciones serían las que afectan al sistema de procesamiento de log, utilizando la arquitectura propuesta y simplemente añadiendo nuevas transformaciones sobre el flujo de datos de entrada dentro de Spark Streaming podríamos llegar a obtener un conjunto más rico de resultados. Además, dado que todo el procesamiento se realiza utilizando Spark, se puede considerar la utilización de técnicas de *machine learning* soportadas por MLlib que permitan clasificar la actividad del sistema de Dispensación Electrónica de forma automática basándonos en los nuevos conjuntos de resultados, que contarán con mucha más información. Como resultado, podríamos llegar a conseguir una herramienta muy potente capaz de predecir la actividad del sistema permitiendo adelantarse a situaciones indeseadas antes de que se produzcan.

En relación con lo anterior, la aplicación web también es susceptible de grandes mejoras puesto que, si contamos con una herramienta que nos permite clasificar la actividad de forma automática y en tiempo real, podemos ofrecer un servicio de alertas ante determinados patrones de tal forma que se ponga sobre aviso a los usuarios ante la posibilidad de que pueda ocurrir una determinada situación (por ejemplo el colapso del sistema). Además, manejar un conjunto de resultados más rico supone rediseñar los informes generados permitiendo producir cuadros de mando con un mayor número de indicadores. La interfaz gráfica también es susceptible de mejoras. Una de las principales sería una conversión automática de unidades de tal forma que los tiempos no se muestren siempre en milisegundos sino que se muestren en unidades mayores que faciliten su interpretación.

En nuestra implementación se trata el sistema de Dispensación Electrónica como un único componente que emite datos cuando, en realidad, se trata de cuatro nodos independientes. Puede ser interesante, sobre todo pensando en la clasificación automática de la actividad, contar con información acerca de en qué nodo se ha registrado cada una de las líneas y poder ver de este modo el estado de cada uno por separado. Actualmente, en la implementación del sistema de Dispensación Electrónica real esto no se considera puesto que cada nodo genera sus log en una carpeta independiente y, por lo tanto, no es necesario incluir esta información como un campo especial dentro de cada línea. Podríamos modificar el formato de las líneas para que siempre incluyan este nuevo campo. No obstante, si no deseamos que el formato se deba modificar podemos optar por

añadir lo que se conoce en Flume como *Interceptor*. Los *Interceptor* ofrecen la posibilidad de modificar o eliminar eventos a medida que se reciben. Existen implementaciones predefinidas pero también se ofrece la posibilidad de crear nuevos *Interceptor* a partir de clases Java propias que implementen el interfaz *org.apache.flume.interceptor.Interceptor*. En nuestro caso, simplemente podríamos coger uno de los que ya viene implementado por defecto, *Host Interceptor*, cuya funcionalidad es precisamente la de añadir una cabecera con el nombre del host y la IP.

En nuestro análisis de las pruebas realizadas, hemos tenido en cuenta cierta información acerca de la actividad de los nodos del sistema de Dispensación Electrónica por separado. Esto ha sido posible gracias a que contábamos con las estadísticas de uso de los agentes Flume. Esta información nos ha permitido entender mejor que estaba pasando en cada momento, lo que demuestra que esta ampliación en la capa de recolección de datos y, consecuentemente, en el capa de procesamiento, podría aportar gran valor añadido sobre la implementación actual.

En todo caso, se utiliza la misma arquitectura base, únicamente se modifica la configuración de los agentes Flume para mantener en cada evento el nombre y la IP del nodo que lo envía y se rediseña el algoritmo que se ejecuta en la capa de procesamiento para dar soporte a las nuevas funcionalidades. Debemos tener en cuenta que un procesamiento más complejo de los resultados unido a la ejecución de los algoritmos de *machine learning* pueden tener un impacto significativo sobre el rendimiento obligando a aumentar el número de nodos del clúster, en todo caso, Spark en general, y tanto Spark Streaming como MLlib en particular, se han diseñado para escalar adecuadamente a medida que se aumentan los nodos del clúster.

9. Anexos

9.1. Instalación Docker

A continuación describiremos el proceso de instalación de Docker, en nuestro caso utilizaremos un equipo que tiene instalado **Ubuntu 13.10**. En todo caso, en la propia documentación de Docker se recoge una guía de instalación muy completa en la que se comenta este proceso para múltiples sistemas y versiones. Instalaremos la última versión de Docker disponible, que en el momento de realizar este proyecto es la **1.7.1**.

9.1.1. Prerrequisitos

Docker, de forma oficial, únicamente ofrece soporte para sistemas de **64 bit**. Además, en sistemas Linux necesita, como mínimo, la **versión 3.10 del kernel**. Para averiguar la versión que tenemos instalada simplemente podemos ejecutar el siguiente comando:

```
$ uname –r
3.11.0-12-generic
```

Comando 9.1

Como se puede ver, en nuestro caso contamos con una versión superior con lo que podremos continuar el proceso de instalación sin problema.

9.1.2. Instalación

Ejecutando el siguiente comando se instalará la última versión de Docker:

```
$ wget -qO- https://get.docker.com/ | sh
```

Comando 9.2

Pedirá la clave de *sudo* con lo que el usuario debe tener los permisos necesarios.

Con el fin de verificar la instalacióndne Docker podemos ejecutar el siguiente comando:

```
$ sudo docker run hello-world
```

Comando 9.3

Como se pude ver en la línea anterior, debemos ejecutar *docker* con *sudo*. Podemos evitar esto creando un grupo llamado *docker* y añadiendo los usuarios a dicho grupo. En todos los *script* que manejamos en este TFG no se incluye *sudo* con lo que, para que funcionen sin realizar ninguna modificación debemos realizar esta acción, para ello ejecutaremos el siguiente comando:

```
$ sudo usermod -aG docker usuario
```

Comando 9.4

Para que los cambios surtan efecto debemos cerrar sesión y volver a iniciarla, para comprobar que todo está configurado correctamente podemos ejecutar el siguiente comando:

```
$ docker run hello-world
```

Comando 9.5

9.2. Anonimizar log reales

El requisito RCA_004 establece que a los efectos de la realización de este Trabajo Fin de Grado se deben manejar datos anonimizados, para ello desarrollaremos un script Python que se encargue de realizar dicha tarea. Utilizaremos datos reales anonimizados en lugar de conjuntos de datos sintéticos con el fin de poder imitar los más fielmente posible el funcionamiento real del sistema de Dispensación Electrónica y ver si nuestro sistema se comporta según lo esperado en bajo pruebas de carga reales.

En la sección de análisis tecnológico se explican en detalle los formatos de las diferentes líneas de log que nos podemos encontrar. A continuación se pueden ver tres líneas ya anonimizadas que hacen referencia a la misma operación (el cuarto campo indica el *id* de la operación).

2015-04-29 22:05:09,214 | INFO | plataforma | [18c82e855a11c46e7d71b425f6abeab84c753bf7] | Petición | Servicio: ServicioDispensaciones | Operación: registroDispensacion_eReceita | Cliente: NCD

2015-04-29 22:05:10,792 | INFO | plataforma | [18c82e855a11c46e7d71b425f6abeab84c753bf7] | <datos><farmacia>FC00000</farmacia><farmaceutico>FT00000</farmaceutico><tipodocumento>CIP</tipodocumento><documento>DOC00000</documento><fuente>MANUAL</fuente></datos>

2015-04-29 22:05:21,072 | INFO | plataforma | [18c82e855a11c46e7d71b425f6abeab84c753bf7] | Respuesta | Servicio: ServicioDispensaciones | Operación: registroDispensacion_eReceita | Cliente: NCD | 5253 | 11859

El *id* se construye con información acerca de la operación (puede contener información como el identificador de la farmacia) con lo que este campo debe ser anonimizado, además anonimizaremos el identificador del farmacéutico, el identificador de la farmacia y el documento del paciente. Los demás campos los mantendremos sin modificar puesto que no contienen información sensible para los usuarios sino que únicamente informan acerca del funcionamiento del sistema.

Para generar nuevos *id* en los datos anonimizados utilizaremos una función *hash*, en concreto *SHA-1*, de este modo, conseguiremos que los *id´s* iguales generen el mismo *id* con lo que no perderemos la relación entre las líneas que se refieren a una misma operación. A la hora de sustituir los demás campos no utilizaremos esta función *hash* puesto que supondría un drástico incremento en el tamaño de los ficheros de salida ya que la longitud de estos identificadores es, en general, muy inferior a la del identificador de la operación. En este caso utilizaremos un diccionario que nos permita sustituir campos iguales siempre por el mismo valor, en el caso de las farmacias y los farmacéuticos utilizaremos el patrón FCXXXXX y FTXXXXX respectivamente, donde cada X representa las cifras de un número incremental de tal forma que la primera aparición de una farmacia se sustituye por FC00000, a la siguiente le correspondería FC00001 y así sucesivamente. En el caso de los documentos de pacientes utilizaremos el patrón DOCXXXXX. De este modo, conseguimos ocultar todos los datos que puedan ser sensibles para los usuarios, por el contrario, mantenemos los patrones de uso del sistema de cada usuario (aunque no se relacione esta actividad con el identificador real de dicho usuario) así como de las farmacia y los farmacéuticos.

A continuación mostraremos el seudocódigo utilizado:

INICIO:
```
  farmacias = {}
  farmaceuticos = {}
  documentos = {}
```

```
        PARA fichero EN listar_ficheros():

//algunos documentos de paciente contienen saltos de línea que es necesario eliminar para poder
//procesar el fichero línea a línea y caracteres especiales no permitidos en XML como '<' que también
//es necesario eliminar para poder extraer los datos originales y anonimizarlos por eso necesitamos
// un preprocesado
            preprocesado(fichero)
            lineas = leer_lineas(fichero)
            borrar_contenido(fichero)
            PARA linea EN contenido:
              campos = línea.split(" | ")
              id = getId(campos[3]) //Extraer "id" de "[id]"
              campos[3] = '[' + sha1(id) + ']'
              SI contiene_datos(linea):
                datos = paserXML(campos[4])
                SI datos.famaceutico != "":
                  SI datos.famaceutico en farmaceuticos
                    datos.famaceutico = farmaceuticos[datos.famaceutico]
                  SINO
                    id = siguiente_id(farmaceuticos)
                    farmaceuticos[datos.famaceutico] = id
                    datos.famaceutico = id

                SI datos.farmacia != "":
                  SI datos.farmacia en farmacias
                    datos.farmacia = farmacias[datos.farmacia]
                  SINO
                    id = siguiente_id(farmacias)
                    farmacias[datos. farmacia] = id
                    datos.farmacia = id

                SI datos.documento != "":
                  SI datos.documento en documentos
                    datos.documento = documentos[datos.documento]
                  SINO
                    id = siguiente_id(documentos)
                    documentos[datos. documento] = id
                    datos.documento = id

                campos[4] = convertir_a_texto(datos)
              linea = " | ".join(campos)
              escribir_linea(fichero, linea)
        Fin
```

9.3. Despliegue del sistema

En este anexo se recoge un breve resumen de los pasos necesarios para desplegar todos los componentes. A lo largo de la memoria se discuten todos los conceptos manejados en mayor profundidad, sobre todo, en la secciones de implementación y pruebas. Previamente, se debe instalar Docker siguiendo lo especificado en el correspondiente anexo.

En primer lugar se debe crear la imagen para los contenedores del clúster ejecutando el siguiente comando en la carpeta *TFG_250815/hdp-docker/imagen* que se encuentra en el material adjunto:

```
$ docker build -t user/hdp:1.0 .
```

Comando 9.6

A continuación deberemos crear los contenedores que forman el clúster (por defecto 3 contenedores):

```
$ . hdp-cluster.sh
$ hdp-create-cluster
$ hdp-provisioning
```

Comando 9.7

Debemos instalar Phoneix de forma manual, para ello, ejecutamos el siguiente comando en todos los contenedores:

```
$ yum install Phoenix
```

Comando 9.8

Para abrir una consola en cada contenedor para ejecutar el comando anterior debemos ejecutar:

```
$ docker exec --it “nombre contenedor” /bin/bash
```

Comando 9.9

Para que Phoenix funcione adecuadamente deberemos reiniciar tanto HBase como ZooKeeper. Podemos hacer esto desde la interfaz web de Ambari que hemos comentado.

También deberemos instalar de forma manual Spark, para ello ejecutamos en el contenedor *amb0* el siguiente comando:

```
$ /compartida/spark-install.sh install
```

Comando 9.10

El script *spark-install.sh* lo podemos encontrar en el material adjunto a esta memoria. Al igual que en el caso de Phoenix, para poder ejecutar el comando en el contenedor deberemos utilizar la siguiente sentencia:

```
$ docker exec --it amb0 /bin/bash
```

Comando 9.11

Tras instalar Spark, para realizar las pruebas hemos configurado el mecanismo de generación de log (ver sección *6.1.4.1 Instalación y configuración de Spark*).

Por último deberemos instalar y configurar el agente Flume que se ejecutará en el clúster Hadoop tal y como se comenta en la sección *6.1.4.3 Instalación y configuración de Flume*. De este modo, tenemos todos los componentes del clúster Hadoop que necesitamos instalados y desplegados.

Además del clúster Hadoop, debemos ejecutar los contenedores que simulan el funcionamiento del sistema de receta electrónica. Crearemos la imagen ejecutando el siguiente comando en la carpeta *TFG_250815/imgDISEL* que se encuentra en el material adjunto:

```
$ docker build -t user/hdp:1.0 .
```

Comando 9.12

A continuación, ejecutaremos el siguiente comando para crear cada uno de los cuatro contenedores que necesitamos:

```
$ docker run -it -v /usr/hdp:/compartida --link amb0:amb0 disel:1.0
/bin/bash
```

Comando 9.13

Ejecutaremos la aplicación de simulación del sistema con el siguiente comando desde la carpeta */compartida/log* siendo *args* el nombre de la carpeta en la que se encuentran los ficheros del nodo que se desea simular y el nombre de la instancia del sistema de Dispensación Electrónica dentro de dicho nodo.

```
$ java -jar /usr/local/bin/disel.jar com.hp.ereceta.disel.Principal args
```

Comando 9.14

En cada contenedor también deberemos ejecutar un agente flume, para ello ejecutamos el script “flume” que se encuentra en la ruta */usr/local/bin/apache-flume-1.6.0-bin* dentro del contenedor.

10. Bibliografía

[1] Ralph Kimball y Margy Ross, The Data Warehouse Toolkit: The Complete to Dimensional Modeling, John Wiley. ISBN:9781118530801.

[2] B. Marr, Big Data: Using smart big data analytics and metrics to make better decisions and improve performance, John Wiley & Sons, 2015.

[3] S. Mohanty, M. Jagadeesh y H. Srivatsa, Big Data Imperatives: Enterprise Big Data Warehouse, BI Implementations and Analytics, Apress. ISBN:9781430248736, 2013.

[4] M. Schroeck, R. Shockley, J. Smart, D. Romero-Morales y P. Tufano, «Analytics: el uso de big data en el mundo real».

[5] S. Wadkar, M. Siddalingaiah y J. Venner, Pro Apache Hadoop, Apress. ISBN: 9781430248644, 2014.

[6] «Apache Hadoop» https://hadoop.apache.org/.

[7] D. Jeffrey y G. Sanjay, «MapReduce: Simplified Data Processing on Large Clusters,» *Sixth Symposium on Operating System Design and Implementation,* 2004.

[8] S. Ghemawat, S.-T. Leung y H. Gobioff, «The Google File System,» de *19th ACM Symposium on Operating Systems Principles*, Lake George, 2003.

[9] T. White, Hadoop: The Definitive Guide, O'Reilly. ISBN: 9781449311520, 2012.

[10] S. Karanth, Mastering Hadoop: Go beyond the basics and master the next generation of Hadoop data processing platforms, Packt Publishing. ISBN: 9781783983643, 2014.

[11] «Cloudera» http://www.cloudera.com/.

[12] «Hortonworks Data Platform» http://hortonworks.com/hdp/.

[13] «MapR» https://www.mapr.com/.

[14] «Apache Avro» https://avro.apache.org/.

[15] «Apache Flume» https://flume.apache.org/.

[16] S. Hoffman, Apache Flume: Distributed Log Collection for Hadoop, Packt Publishing. ISBN: 9781782167914, 2013.

[17] «Apache Kafka» http://kafka.apache.org/.

[18] «Apache ZooKeeper» https://zookeeper.apache.org/.

[19] «Apache HBase» http://hbase.apache.org/.

[20] «Big Table» http://jimbojw.com/wiki/index.php?title=Understanding_Hbase_and_BigTable.

[21] Apache HBase Team, Apache HBase Reference Guide.

[22] K. Amandeep, «Introduction to HBase Schema Design» 2012.

[23] «Apache Hive» https://hive.apache.org/.

[24] «Hive con Spark» https://cwiki.apache.org/confluence/display/Hive/Hive+on+Spark.

[25] «Cloudera Impala» Available: http://impala.io/.

[26] «Apache Phoenix» https://phoenix.apache.org/.

[27] «Apache Spark» http://spark.apache.org.

[28] H. Karau, A. Konwinski, P. Wendell y M. Zaharia, Learning Spark, O'Reilly. ISBN: 9781449358624, 2015.

[29] «Spark con Flume» http://spark.apache.org/docs/1.2.1/streaming-flume-integration.html.

[30] «Apache Storm» https://storm.apache.org/.

[31] «Pentaho Reporting» http://community.pentaho.com/projects/reporting/.

[32] «Docker» https://www.docker.com/.

[33] J. Turnbull, The Docker Book, 2014.

[34] «Arquitectura Lambda» http://lambda-architecture.net/.

[35] E. Byron, Real-Time Analytics: Techniques to Analize and Visualize Streaming Data, John Wiley & Sons. ISBN: 9781118837917, 2014.

[36] «Summingbird» https://github.com/twitter/summingbird.

[37] «Limitaciones Arquitectura Lambda» http://radar.oreilly.com/2014/07/questioning-the-lambda-architecture.html.

[38] «Docker con Ambari» http://blog.sequenceiq.com/blog/2014/12/04/multinode-ambari-1-7-0/.

[39] «HDP en Docker» http://hortonworks.com/blog/docker-ships-hdp-cloud/.

[40] «Uso del API Pentaho Reporting» http://msvaljek.blogspot.com.es/2014/09/pentaho-reports-with-java-api.html.

[41] «Serf» https://www.serfdom.io/.

[42] «Ambari blueprint» http://www.pythian.com/blog/ambari-blueprints-and-one-touch-hadoop-clusters/.

[43] «Spark en Docker» http://blog.sequenceiq.com/blog/2015/01/09/spark-1-2-0-docker/.

[44] «Phoenix con HDP» http://docs.hortonworks.com/HDPDocuments/HDP2/HDP-2.1.3/bk_installing_manually_book/content/rpm-chap-phoenix.html.

[45] M. Massie, B. Li, B. Nicholes y V. Vuksan, Monitoring with Ganglia, O'Reilly. ISBN: 9781449329709, 2012.

[46] «Apache Ambari» https://ambari.apache.org/.

[47] «Docker con Ambari» http://blog.sequenceiq.com/blog/2014/06/17/ambari-cluster-on-docker/.