Universidad Nacional del Centro de la Provincia de Buenos Aires

Facultad de Ciencias Exactas

TRABAJO FINAL DE LA CARRERA DE INGENIERÍA DE SISTEMAS

Aether, un framework para facilitar la implementación de aplicaciones basadas en Cloud Computing

Marcos Maximiliano Steimbach

Joaquín Alejandro Pérez Fuentes

Director: Dr. Álvaro Soria

Agradecimientos

Queremos agradecer el apoyo incondicional de nuestras familias y amigos a lo largo de todos estos años. Al Dr. Álvaro Soria, que nos ha guiado con paciencia e interés durante el desarrollo de esta tesis. También a la Universidad y sus profesores por habernos acompañado en esta larga pero fructífera carrera.

A todos ellos, muchas gracias por hacer esto posible.

SÍNTESIS

El mercado IT está siendo transformado por el modelo de *Cloud Computing* y la tecnología como servicio. *Cloud Computing* es un modelo de acceso bajo demanda a recursos computacionales. Estos recursos pueden ser tomados y liberados rápidamente con un mínimo esfuerzo de configuración e interacción con el proveedor. Las principales ventajas de este paradigma radican en la rapidez y facilidad de implantación y gestión, la flexibilidad, el ahorro de costos, la capacidad de escalabilidad, de forma que estas puedan adaptarse instantáneamente a las necesidades específicas que una compañía pueda tener en un momento determinado, sin depender de adquisiciones de servidores o de licencias de software.

Existe en la actualidad un creciente abanico de proveedores para todo tipo de servicios de *Cloud Computing*, incluyendo a grandes nombres de la industria como Microsoft, Google, IBM o Amazon. Esto ha llevado a que los desarrolladores deban diseñar sus aplicaciones pensando en una potencial migración para aprovechar las ventajas competitivas de otras plataformas. Esto es útil ya que múltiples razones pueden llevar a que un usuario desee cambiar de proveedor, incluso teniendo su aplicación en producción. Por ejemplo, los costos de un proveedor pueden bajar y convertirlo en una alternativa interesante.

Debido a esto, surgen herramientas que tratan de darle solución a este problema abstrayendo una interfaz común que luego adaptan a las interfaces de los proveedores que soportan. Sin embargo, trasladan el problema a la herramienta, dejando dependiente a la aplicación de la herramienta y de los proveedores que esta soporta. La limitación principal de estas herramientas yace en que no consideran la posibilidad de migrar una aplicación ya codificada con otra herramienta. Supongamos el caso de tener una aplicación codificada con el SDK de Amazon y que, en un momento dado, los creadores de esta aplicación determinan que les resultaría conveniente migrar a Azure, proveedor no soportado por el SDK de Amazon. En este caso, el equipo de desarrollo se encontraría con la situación de tener que recodificar su aplicación con otra herramienta acorde al nuevo proveedor.

Como consecuencia de esto se presenta Aether, un framework basado en reflexión y reemplazo de código en tiempo de ejecución. Utilizando Aether se logró reducir significativamente el costo de migración de aplicaciones entre distintos proveedores de servicios en *Cloud*, logrando en algunos casos reducir el costo del orden de las mil líneas de código o configuración a tan solo un par de decenas.

Palabras clave: Cloud Computing, migración.

CONTENIDO

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

CAPÍTULO 1

1. INTRODUCCIÓN

El mercado IT (Alegsa, 2013) está siendo transformado por el modelo de *Cloud Computing* y la tecnología como servicio. Los expertos creen que en pocos años la gran mayoría de las empresas estarán utilizando la *Cloud*, de igual modo que adoptaron el correo electrónico o la PC en su momento. Las principales ventajas de *Cloud Computing* radican en la rapidez y facilidad de implantación y gestión, la flexibilidad, el ahorro de costes, prestaciones técnicas, capacidad de escalabilidad, de forma que estas puedan adaptarse instantáneamente a las necesidades específicas que una compañía pueda tener en un momento determinado, ante un pico de trabajo, sin depender de adquisiciones de servidores o de licencias de software. Esto genera una mayor agilidad en la toma de decisiones empresariales y, consecuentemente, una mayor competitividad de las empresas (Franco, 2013).

Cloud Computing es un modelo de acceso bajo demanda a recursos computacionales compartidos (servidores, almacenamiento, servicios, etc.) (Mell & Grance, 2011). Estos recursos pueden ser tomados y liberados rápidamente con un mínimo esfuerzo de configuración e interacción con el proveedor. Este modelo presenta algunas características principales como son por ejemplo: proveer self-service de recursos bajo demanda con lo cual el cliente puede obtener y dejar recursos de manera unilateral sin intervención del proveedor, proveer un pool de recursos para poder distribuir los mismos entre múltiples clientes de manera dinámica bajo la demanda de cada uno, permitir un crecimiento/decrecimiento rápido ya que el cliente puede obtener y dejar recursos rápidamente acorde a su demanda y proveer además un servicio medido lo cual permite a los sistemas en *Cloud* controlar el uso de los recursos de manera transparente al cliente y facturarle en base a esto (por ejemplo, para máquinas virtuales el uso suele medirse en horas mientras que en almacenamiento se mide en GBs. Todos los proveedores manejan variaciones de este mismo esquema).

Los servicios de *Cloud Computing* se agrupan en tres tipos básicos:

- SaaS (Software as a Service). Se le provee al cliente la posibilidad de utilizar una aplicación del proveedor que corre en una infraestructura de tipo *Cloud*. Ejemplos de esto son los servicios de webmail.
- PaaS (Platform as a Service). Se le provee al cliente la habilidad de desplegar una aplicación propia creada con el lenguaje, bibliotecas y herramientas soportadas por el proveedor. El cliente debe adaptarse al ambiente de ejecución provisto. Ejemplos de este tipo se servicio son Google App Engine (Google, 2013) o Heroku (Heroku, 2013).

- IaaS (Infrastructure as a Service). Se le provee al cliente la habilidad de obtener recursos computacionales en los cuales puede desplegar sus aplicaciones arbitrariamente. En este tipo de servicios el cliente no maneja la infraestructura subyacente pero tiene control sobre sistemas operativos, sistemas de archivos, máquinas virtuales y las aplicaciones que despliega. A diferencia de lo que sucede en PaaS, no se le obliga al cliente a trabajar con un ambiente de ejecución impuesto por el proveedor. Ejemplos de este tipo de servicio son Amazon EC2 (Amazon, 2013), Google Storage (Google, 2013) o Windows Azure (Microsoft, 2013).

Existe en la actualidad una variada oferta de proveedores para todo tipo de servicios siendo los más importantes Amazon, Microsoft, Rackspace (Rackspace, 2013), Google, IBM (IBM, 2013), Hewlett-Packard (Hewlett-Packard, 2013) y VMWare (VMWare, 2013) entre otros. Cabe destacar que pese a la diversidad de los servicios ofrecidos, en casi todos los casos el modelo de cobro es por el uso real de los recursos. Existen algunas variaciones a este modelo como por ejemplo la reserva de instancias por año (similar a un servidor usual), paquetes de almacenamiento fijo o descuentos por grandes cantidades de recursos utilizados (Amazon, 2013).

1.1 Motivaciones y objetivos de la tesis

El continuo crecimiento de la oferta de proveedores de servicios de *Cloud Computing* ha llevado a que los desarrolladores deban diseñar sus aplicaciones pensando en una potencial migración para aprovechar las ventajas competitivas de otras plataformas (Leavitt, 2009) (Armbrust, et al., 2010) (Hofmann & Woods, 2010) (Briscoe & Marinos, 2009). Esto es útil ya que múltiples razones pueden llevar a que un usuario desee cambiar de proveedor, incluso teniendo su aplicación en producción. Por ejemplo, los costos de un proveedor pueden bajar y convertirlo en una alternativa interesante. Otro caso podría ser la velocidad de red que posea determinado proveedor para una ubicación geográfica específica. Incluso se podrían usar varios servicios en simultáneo para generar redundancia.

Debido a esto surgen herramientas que tratan de darle solución a este problema siguiendo el enfoque de intentar homogeneizar interfaces. Estas herramientas abstraen una interfaz común que luego adaptan a las interfaces de los proveedores que soportan. Sin embargo, trasladan el problema a la herramienta, dejando dependiente a la aplicación de la herramienta y de los proveedores que esta soporta. jClouds (jClouds, 2013), Dasein (Dasein, 2013) y libcloud (Apache, 2013) entre otros integran este grupo de herramientas. La limitación principal de estas herramientas yace en que ninguna considera la posibilidad de migrar una aplicación ya codificada con otra herramienta. Supongamos el caso de tener una aplicación codificada con

el SDK de Amazon (Amazon, 2013). En un momento dado los creadores de esta aplicación determinan que les resultaría conveniente migrar a Google Storage, pero este no es soportado por el SDK de Amazon. Como consecuencia, el equipo de desarrollo se encontraría con la situación de tener que recodificar su aplicación con otra herramienta acorde al nuevo proveedor.

Esta problemática será abordada en los capítulos siguientes mediante la presentación de un enfoque para reducir el esfuerzo de codificación al migrar una aplicación entre proveedores de servicios de *Cloud Computing*.

1.2 Estructura de la tesis

A continuación se presenta la estructura de la tesis, la cual fue organizada en 6 capítulos.

En el capítulo 2 se presenta un relevamiento de las herramientas disponibles en la actualidad para trabajar con *Cloud Computing*, exponiendo las ventajas y desventajas de utilizar cada una de estas. El análisis de los trabajos relacionados se centra en las facilidades que presenta cada herramienta para migrar entre proveedores de servicios de *Cloud Computing*. La conclusión del capítulo nos muestra las falencias de las herramientas actuales, precisando que se necesita mejorar para trabajar de forma más efectiva con este tipo de tecnologías.

En el capítulo 3 se detalla el enfoque adoptado para accionar sobre los problemas expuestos en el capítulo anterior. Se elabora sobre un modelo que permite reducir significativamente los costos de migración de aplicaciones entre proveedores de servicios de *Cloud Computing.*

En el capítulo 4 se muestra el diseño y la implementación del framework *Aether*, haciendo énfasis en las decisiones de diseño e implementación que se tomaron para materializar el modelo del capítulo anterior en el lenguaje Java.

En el capítulo 5 se presentan los resultados experimentales de la utilización del framework desarrollado en comparación con las herramientas pre existentes. El foco estará puesto en evaluar las capacidades de migración, el consumo de recursos y que tan fácil resulta el uso de cada herramienta. Para tal fin se utilizarán tres aplicaciones open source como base que serán modificadas para evaluar los puntos mencionados.

En el capítulo 6 se presentan las conclusiones generales del trabajo junto a las posibilidades de mejoras futuras.

CAPÍTULO 2

2. ESTADO DEL ARTE

El modelo de *Cloud Computing* que hoy conocemos tiene su origen a principio de los años 2000. La industria del software acababa de pasar por la denominada "Burbuja punto com" (Dot-com bubble) (Mandel, 2001), una corriente económica especulativa muy fuerte que se dio entre 1997 y 2001. Este período fue marcado por la fundación y el rápido crecimiento en bolsa de compañías basadas en Internet, designadas comúnmente empresas punto com. Al pasar el tiempo, muchas de estas empresas quebraron o dejaron de operar y las que sobrevivieron se reorganizaban en busca de recuperarse.

Una de estas empresas, y pionera en *Cloud Computing*, fue Amazon. Por experiencia, Amazon sabía que el costo de mantener servidores era muy elevado ya que resultaba necesario mantener hardware pensando en picos de carga que quizás no fuesen muy frecuentes (Clark, 2013). Teniendo esto en mente se comenzó a pensar en desacoplar y descentralizar la infraestructura, proveyéndola como servicio. La idea inicial consistía en trabajar sobre estos conceptos para reorganizar la infraestructura interna de Amazon, aunque luego la idea creció para darle lugar a la posibilidad de proveer esto como servicio público. Así fue como años más tarde nace Amazon Web Services (AWS), un pionero del *Cloud Computing* que hoy consumimos.

Como era de esperarse, tras el éxito de AWS comenzaron a lanzarse alternativas. Algunas plataformas como Eucalyptus (Eucalyptus Systems, 2013) se basaban en las interfaces ya definidas por Amazon, mientras que otras como OpenNebula (OpenNebula Project, 2013) lanzaron su propio set de interfaces. El mercado del *Cloud Computing* siguió creciendo de la misma manera hasta llegar a la actualidad, donde contamos con gran oferta de proveedores y prácticamente la misma cantidad de interfaces diferentes para servicios que en definitiva se comportan de la misma manera.

Si bien hoy en día existen iniciativas de estandarización (Computer World, 2013) (NIST, 2013) (InfoWorld, 2013) (TrainSignal, 2013), estas han llegado muy tarde encontrándose con plataformas establecidas en posiciones dominantes del mercado que difícilmente se vean motivadas en cambiar. La alternativa a los estándares viable en el corto plazo fue pensar en herramientas de terceros que hagan de intermediario entre el desarrollador y los diferentes servicios de *Cloud Computing*. Con esta idea nacieron herramientas como jClouds (jClouds, 2013), Dasein (Dasein, 2013) y libcloud (Apache, 2013). Estas herramientas proveen una interface común que internamente debe lidiar con la traducción a la interface provista por el proveedor que se desea utilizar. Este modo de trabajar traslada el problema de la migración

a la herramienta, dejando dependiente a la aplicación de ella y de los proveedores que esta soporta. La limitación principal de estas herramientas yace en que ninguna considera la posibilidad de migrar una aplicación ya codificada con otra. Supongamos el caso de tener una aplicación codificada con el SDK de Amazon. En un momento dado los creadores de esta aplicación determinan que les resultaría conveniente migrar a Google Storage, pero este no es soportado por el SDK de Amazon. Como consecuencia, el equipo de desarrollo se encontraría con la situación de tener que recodificar su aplicación con otra herramienta acorde al nuevo proveedor.

2.1 Frameworks actuales

Las secciones siguientes analizan los frameworks actuales de código abierto entre los que se encuentran CloudLoop (Cloudloop, 2013), Dasein, Deltacloud (Apache Software Foundation, 2013), Java-Storage (java-storage, 2013), jClouds, JetS3t (JetS3t, 2013) y libcloud. El análisis de cada uno de estos frameworks será enfocado en las facilidades que posea cada uno para cumplir con los siguientes puntos:

- Soporte para las primitivas básicas de almacenamiento. Esto significa evaluar si el framework es capaz de subir y bajar archivos, listarlos, extraer sus metadatos o borrarlos.

- Capacidad de migrar entre protocolos sin producir mayores cambios en la aplicación. Por ejemplo, una aplicación codificada para S3, ¿puede migrarse a Google Storage? Si es así, ¿impacta en el código fuente? ¿Puede hacerse de manera simple por archivos de configuración?

- Facilidades para migrar una aplicación ya codificada desde otro framework. Por ejemplo, si en un primer momento la aplicación se codifico contra el SDK de Amazon, ¿se provee algún mecanismo para migrar el código a Google Storage? Si esto no es así, ¿el usuario debería recodificar su aplicación desde cero para hacer uso del nuevo servicio?

El capítulo concluye con una serie de observaciones generales sobre los puntos analizados.

2.1.1. CloudLoop

CloudLoop es un proyecto de código abierto compuesto de una aplicación que presenta una API homogénea para distintos proveedores de servicios de almacenamiento de datos en Cloud. Permite almacenar, administrar y sincronizar los datos del usuario entre los proveedores más comunes.

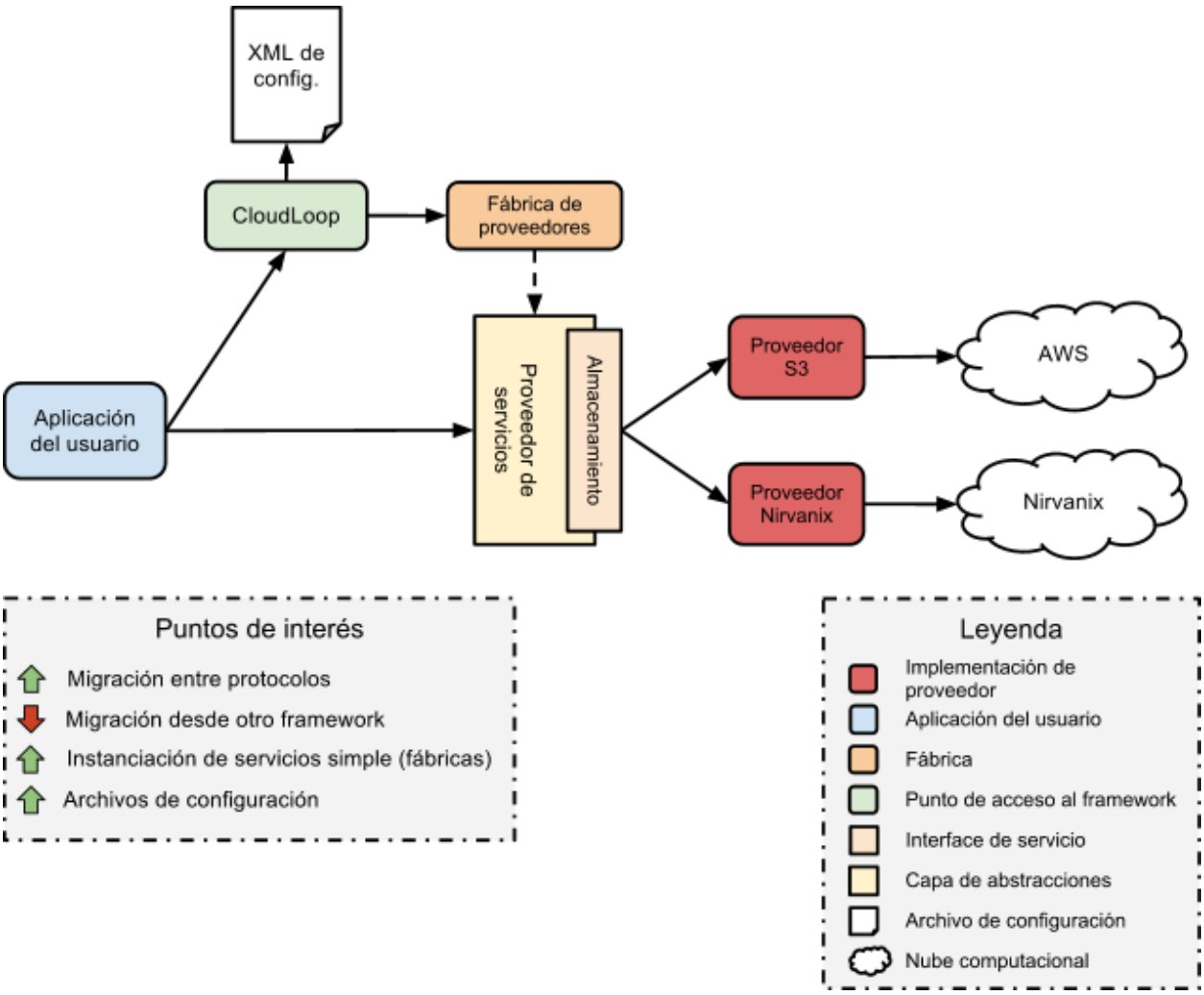

Figura 2.1 Vista de alto nivel de la arquitectura de CloudLoop.

Como se puede apreciar en la figura 2.1, para utilizar esta herramienta es necesario introducir la configuración para el proveedor deseado en un archivo XML. En estos archivos se definen datos básicos del servicio que se desea instanciar como credenciales, directorios a utilizar, etc. En base a la configuración introducida la aplicación del usuario le puede pedir a CloudLoop que prepare una instancia

del proveedor deseado sin complicaciones. La instancia resultante implementa una interface genérica, gracias a lo cual se minimizan las modificaciones de código en caso de cambios de proveedores.

Las interfaces de CloudLoop soportan todas las primitivas básicas de almacenamiento excluyendo la posibilidad de extraer metadatos (solo es posible obtener el tamaño de un archivo). El uso de cada método es sencillo y tiene documentación clara que acompañan al desarrollador. Como consecuencia de esto, agregar soporte para un nuevo proveedor a la herramienta es una tarea sencilla ya que tan solo se debe implementar una interface clara que define todos los métodos comunes a los diversos proveedores de almacenamiento.

La plataforma también permite cambiar el proveedor de servicios de forma muy simple ya que casi no es necesario realizar modificaciones en el código fuente de la aplicación. Para esto tan solo se debe cambiar la configuración por medio del XML de configuración, definiendo todos los datos necesarios para el nuevo servicio a utilizar.

Uno de los puntos flojos de la plataforma es la imposibilidad de migrar desde otro framework. En estos casos el desarrollador no tiene otra salida que recodificar la aplicación para adaptarse al modelo de CloudLoop. Por otro lado, el proyecto se encuentra inactivo e incompleto desde finales del año 2008, provocando que no sea utilizado en desarrollo de aplicaciones por la falta de soporte y futuras actualizaciones del framework.

2.1.2. Dasein

Dasein es un proyecto introducido en 2009 por la empresa enStratus Networks LLC. Provee abstracciones para servicios de los tipos compute y almacenamiento, aunque existen otras implementaciones para servicios menores de diversos proveedores. El proyecto se encuentra activo.

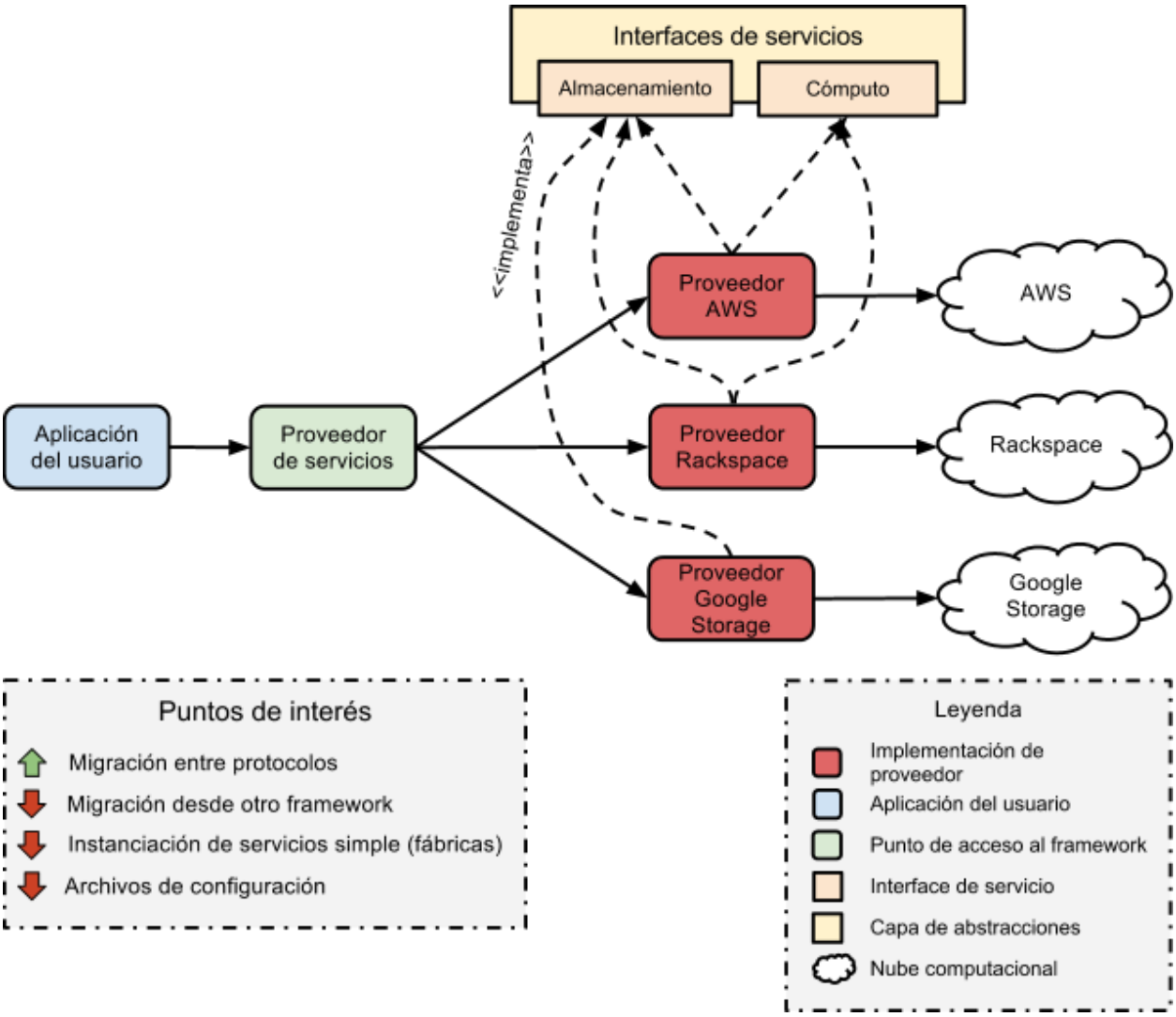

Figura 2.2 Vista de alto nivel de la arquitectura de Dasein.

Como podemos apreciar en la figura 2.2, el punto de acceso al framework se concentra en los proveedores de servicios. Cada proveedor define qué tipo de servicios implementa sin necesidad de tener que soportarlos a todos. Por ejemplo, un proveedor podría soportar únicamente almacenamiento, mientras que otro implementa almacenamiento y cómputo. Un punto a destacar es que en la arquitectura de Dasein no se contempla el uso de archivos de configuración o mecanismos de construcción de instancias de proveedores. Como consecuencia de esto, la configuración e instanciación de los servicios y nubes se realiza completamente por código. Esto genera que el desarrollador deba involucrarse con las particularidades de cada servicio para poder utilizarlos.

La plataforma soporta la mayoría de las primitivas básicas para almacenamiento. Pese a esto, Dasein no provee mecanismos para copiar o buscar archivos mediante un sistema de prefijos o expresiones regulares. Utilizar y descubrir los métodos necesarios no es simple dada la falta total de documentación y comentarios de código. El usuario debe imaginarse que hace cada clase y de qué manera lo hace, todo a prueba y error. Es habitual cruzarse con parámetros poco claros. Por ejemplo, en el método "upload" que se utiliza para subir un archivo, se encuentra el parámetro "multipart". Este parámetro se encuentra en una interface pero solo tiene sentido para Amazon S3 ya que habilita 'multipart upload', un elemento único de su interface (Amazon, 2013). Pese a todo esto, una vez que se comprende qué se debe utilizar y cómo, el framework trabaja correctamente.
En la misma línea, la implementación de un nuevo proveedor tiene algunas complicaciones. Se deben generar implementaciones para la nube (ej.: Amazon) y el servicio particular (ej.: S3). Si bien esto no supone gran complejidad, descubrir que debe hacer cada método que se debe implementar no es trivial. Otra vez se vuelve a caer en la falta de documentación de la plataforma.

En cuanto a la migración entre proveedores dentro de Dasein podemos notar que es posible sin hacer demasiadas modificaciones al código. Para esto es necesario cambiar la nube que se está utilizando y reconfigurar sus propiedades manualmente.

Si hablamos de migrar hacia Dasein desde un producto alternativo el resultado no es positivo. Dasein no provee ningún mecanismo para facilitar este tipo de migración y el usuario se ve obligado a recodificar su aplicación utilizando la nueva tecnología.

2.1.3. Deltacloud

Deltacloud es un proyecto en actividad de la Fundación Apache. Se distingue del resto de los proyectos de su tipo gracias a su arquitectura cliente-servidor basada en una API REST. Gracias a esto se puede hacer uso de Deltacloud desde cualquier lenguaje de programación. Los desarrolladores mantienen un cliente CLI para Ruby y existen bases de un cliente Java, aunque por el momento solo soporta servicios de tipo compute. Si no es posible utilizar Ruby o si la funcionalidad en java es insuficiente, se debe generar un nuevo cliente implementando la API de Deltacloud. Esto se traduce en un incremento en el tiempo y costo de desarrollo.

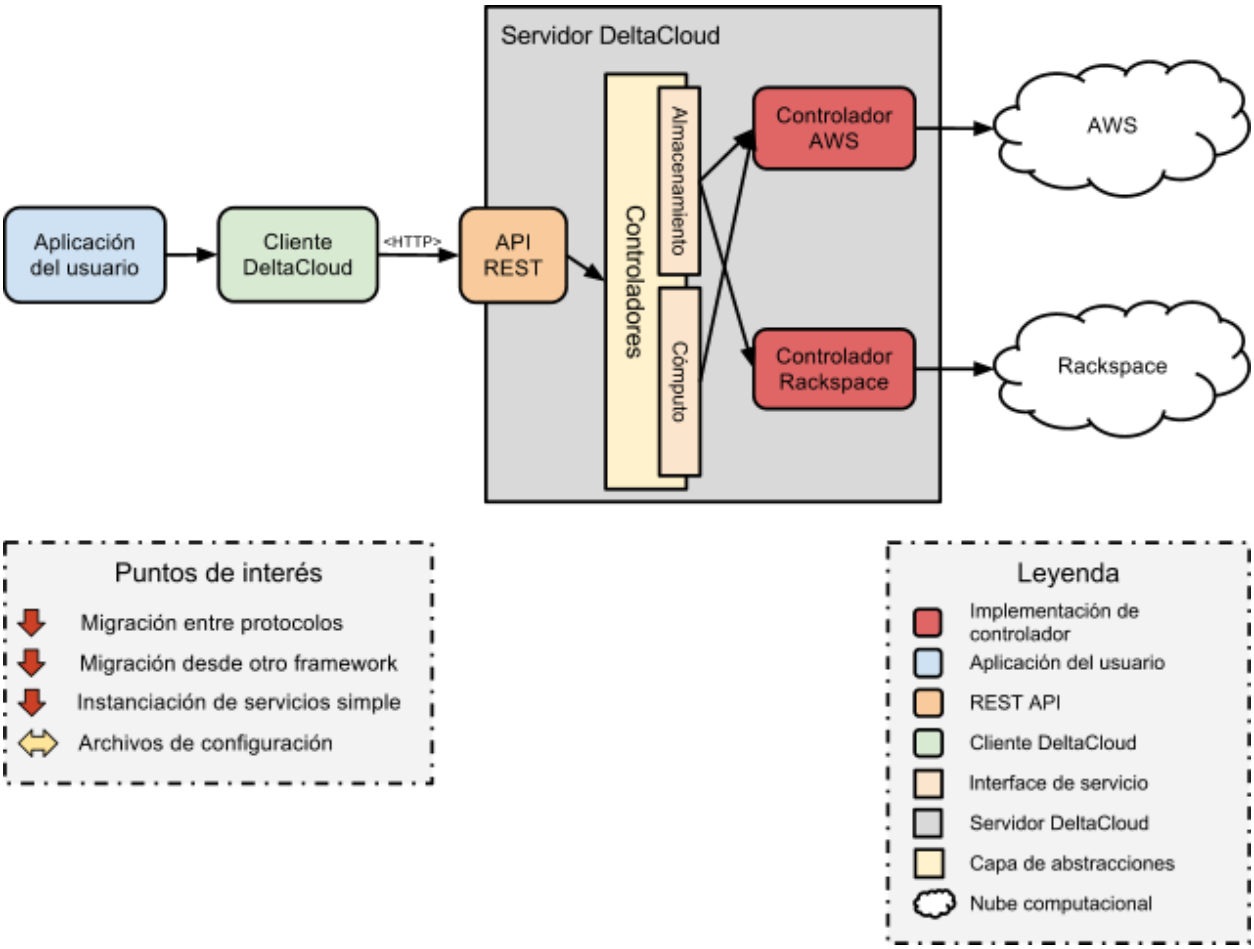

Figura 2.3 Vista de alto nivel de la arquitectura de DeltaCloud.

Como se puede apreciar en la figura 2.3, la arquitectura de DeltaCloud está basada en un modelo cliente servidor, donde el servidor encapsula toda la lógica de acceso a distintos proveedores. Esta decisión de diseño genera que el cliente trabaje de manera muy simple enviando al servidor las credenciales del proveedor y los datos sobre los archivos con los que el usuario quiere trabajar. Para instanciar un servicio se debe configurar tanto el servidor como el cliente. El primero para que levante el servicio puntual y el segundo para enviar los datos del nuevo servicio (credenciales, etc.).

En cuanto al soporte de las funciones básicas de almacenamiento, Delta Cloud tiene algunas limitaciones. Solo se provee soporte para bajar, subir y eliminar archivos. Por el momento no es posible listar, mover o copiar archivos. Las primitivas están definidas en una API clara y con buena documentación. Esto último facilita la tarea a la hora de implementar un nuevo servicio. Para lograr esto el desarrollador debe crear un nuevo Controlador definiendo el tipo de servicio que se está implementando (almacenamiento, compute) lo que define el conjunto de funciones a implementar.

La migración entre protocolos puede ser un problema para la plataforma ya que no es posible levantar un servidor multi cloud. Esto genera que sea necesario contar con varios servidores en varios puertos si se desea trabajar con más de una nube. Como consecuencia directa de este último punto el desarrollador tendrá que modificar tanto el cliente (envío de credenciales, reconfiguración de parámetros de conexión, etc.) como el servidor (reconfiguración) en el caso de querer migrar de un proveedor a otro.

Delta Cloud tampoco provee ningún tipo de mecanismo para facilitar la migración desde un producto competidor (por ejemplo, desde jClouds) por lo que el desarrollador debe recodificar su aplicación.

2.1.4. Java-Storage

Este proyecto tiene como objetivo permitir el acceso a diferentes servicios de almacenamiento de forma genérica. Se distingue del resto de los proyectos por proveer una re implementación de “File” como interface principal. El proyecto se encuentra congelado desde finales de 2009.

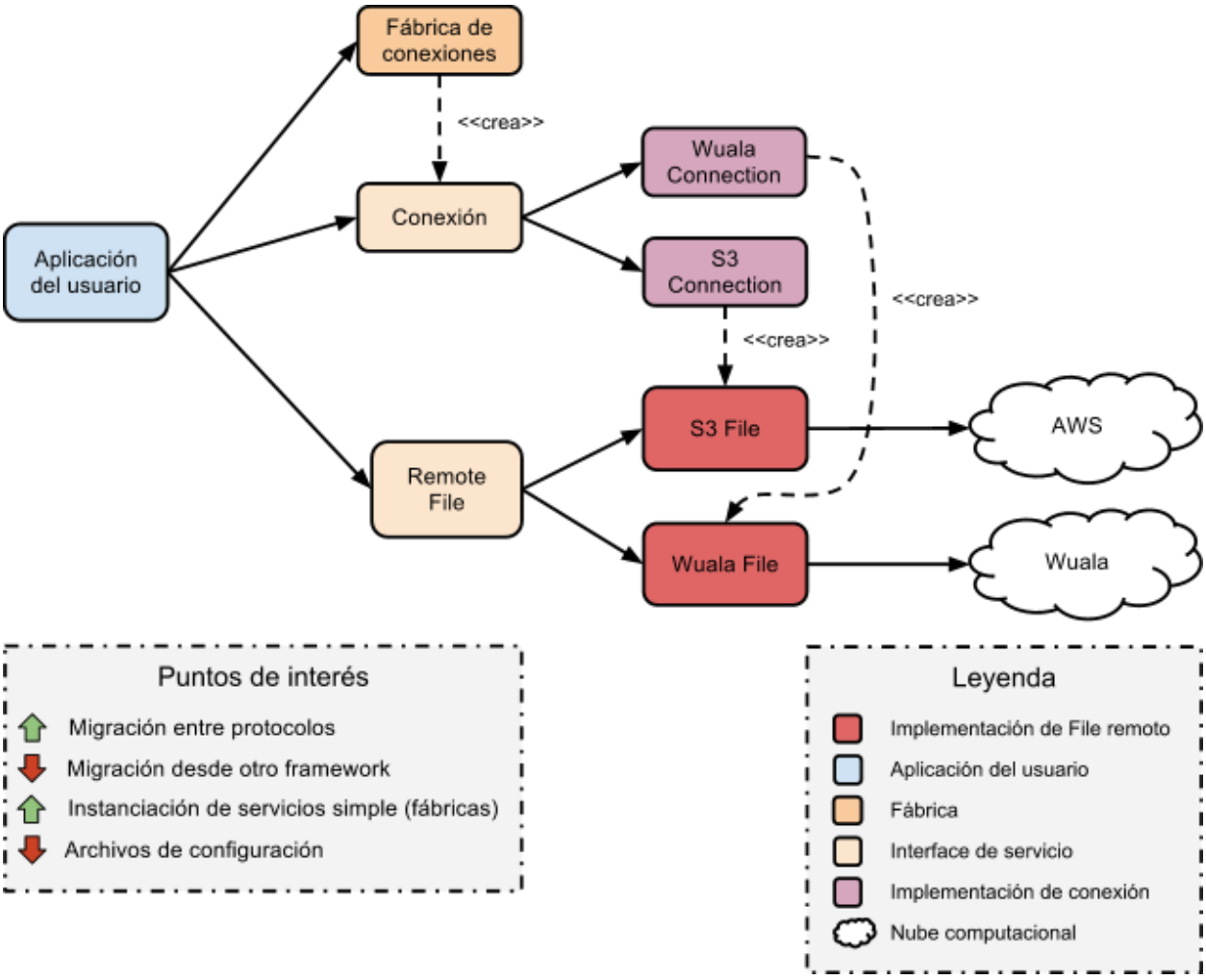

Figura 2.4 Vista de alto nivel de la arquitectura de Java-Storage.

Como se puede apreciar en la figura 2.4, la arquitectura de Java-Storage cuenta con varias particularidades. Por un lado se debe notar que solo soporta servicios de almacenamiento. Por otro lado, el equipo de Java-Storage decidió utilizar como base de su framework a los clásicos File de Java. La idea fue tomar las interfaces de File y extenderlos para que puedan utilizarse para almacenar o acceder archivos en la nube. Cada uno de estos Files es instanciado con la ayuda de una fábrica de conexiones que le retorna al desarrollador una instancia para gestionar la conexión a un proveedor puntual. Contando con esta conexión se pueden obtener los Files como si se tratase de un sistema de archivos normal. Con este modelo se gana mucho en claridad ya que cualquier desarrollador Java seguramente esté familiarizado con las funcionalidades de File lo que baja la curva de aprendizaje.

Java-Storage soporta métodos básicos para descargar y listar archivos, pero carece totalmente de soporte para subir archivos. Esto se suma a la falta de soporte para metadatos, listar, borrar, mover o copiar

archivos. Si hablamos de implementar un nuevo protocolo tan solo es necesario implementar una serie de interfaces bien definidas. A esto contribuye que el usuario de Java típico ya está familiarizado con el uso de File y entiende que debería hacer cada método.

La migración entre protocolos dentro del framework es simple. Tan solo debe cambiarse la instancia de File por la requerida para el nuevo protocolo.

Un punto flojo del proyecto es que no cuenta con facilidades para migrar una aplicación desde otro framework, generando que el desarrollador deba codificar nuevamente su aplicación.

2.1.5. jClouds

jClouds es un framework de código abierto muy útil para comenzar a trabajar en *Cloud Computing* y provee abstracciones para los servicios de compute y almacenamiento. En su desarrollo se tuvo en cuenta la performance y el consumo de recursos ya que fue diseñada también para ser utilizada en ambientes limitados como Google Apps. El proyecto se encuentra activo y es muy usado, incluso por otros frameworks que utilizan sus implementaciones de servicios de bajo nivel.

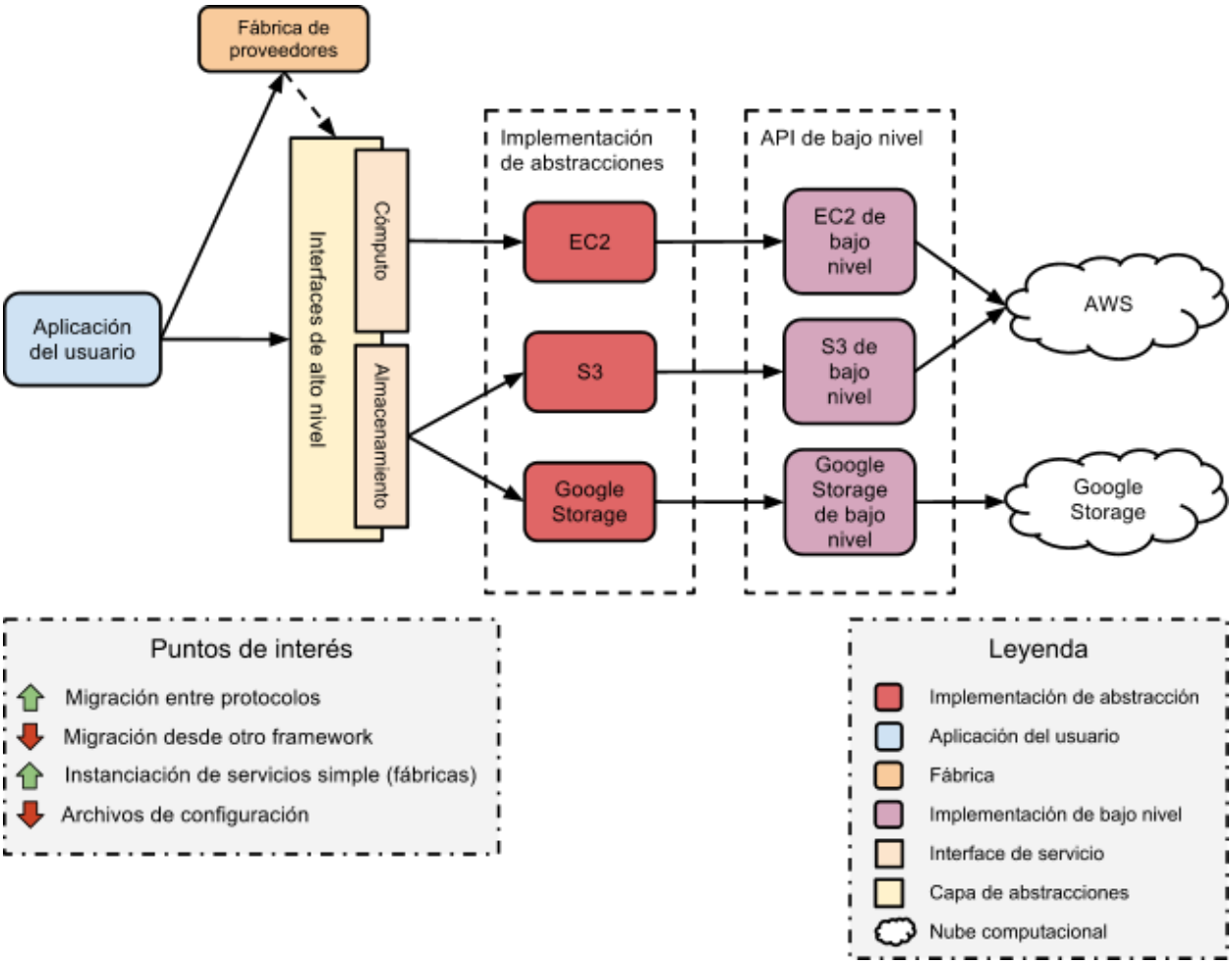

Figura 2.5 Vista de alto nivel de la arquitectura de jClouds.

Como se puede observar en la figura 2.5, jClouds posee una arquitectura elegante y bien llevada a cabo. Cada tipo de servicio soportado (almacenamiento, cómputo, etc.) cuenta con una interface genérica que es implementada por todos los proveedores. Gracias a esto, cada una de las instancias generadas por la fábrica de servicios se maneja de manera homogénea al resto. Un punto que distingue a jClouds es que le brinda la posibilidad al usuario de acceder a métodos y datos de más bajo nivel, presentes únicamente en un proveedor particular. Esto significa que es el usuario y no el framework el que decide con qué nivel de abstracción desea trabajar.

La configuración de los servicios se realiza por medio de código, pero el uso de fábricas con interfaces simples para realizar las instanciaciones hace que la configuración se limite a credenciales que pueden fácilmente volcarse en un archivo properties.

La plataforma soporta la totalidad de las primitivas de almacenamiento por medio de interfaces simples y bien documentadas. En el caso de querer acceder a funcionalidad de bajo nivel jClouds brinda la posibilidad de acceder a la API subyacente con todos sus métodos particulares. Siguiendo la misma tendencia, jClouds hace que la tarea de implementar un nuevo protocolo sea clara y sin mayores complicaciones. Solo se debe implementar la interface definida para almacenamiento. Los desarrolladores de jClouds mantienen documentación simple pero suficiente para esta y otras tareas relacionadas al desarrollo del framework.

Migrar entre protocolos dentro de jClouds es muy simple. Esto es así gracias al uso de fábricas que hacen que cambiar de protocolo resulte tan complejo como cambiar el set de credenciales y demás elementos de configuración del nuevo protocolo.

Uno de los puntos que jClouds no considera es la posibilidad de migrar desde un framework alternativo. El desarrollador se ve obligado a re implementar su aplicación para acceder a las bondades de esta plataforma.

2.1.6. JetS3t

JetS3t es un conjunto de herramientas open-source para los servicios de Amazon Simple Storage Service (Amazon S3) y Google Storage. Proporciona a los programadores Java una API potente y sencilla para interactuar y gestionar datos en servicios de almacenamiento en *Cloud*. JetS3t se encuentra activo y cuenta con una gran base de usuarios, principalmente como conector a S3.

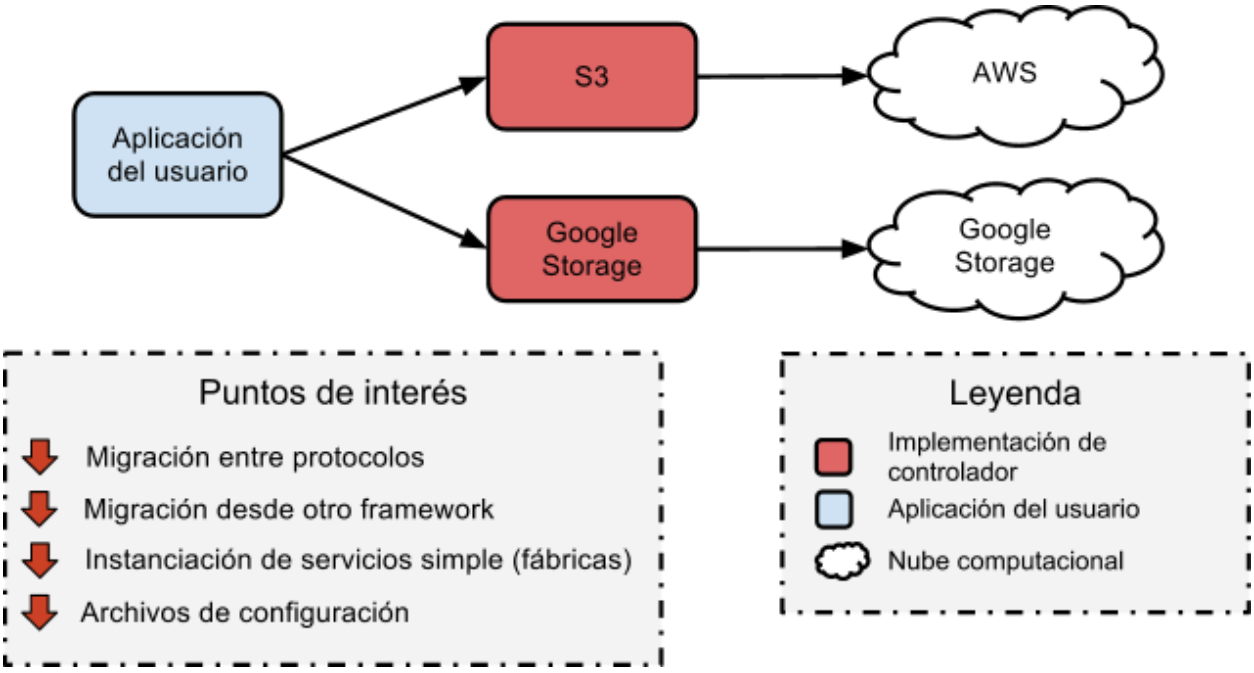

Figura 2.6 Vista de alto nivel de la arquitectura de JetS3t.

Como se puede apreciar en la figura 2.6, la arquitectura de JetS3t es muy básica. El desarrollador debe instanciar directamente el controlador para el proveedor deseado ya que no se cuenta con interfaces. Esto significa que no se cuenta con mecanismos para instanciar objetos fácilmente, ni facilidades para configurar los servicios por medio de archivos o mecanismos similares. Esto provoca que cualquier cambio de proveedor o de configuración de los mismos impacte sobre el código del usuario.

La plataforma soporta todas las primitivas básicas de almacenamiento. Su uso es sencillo, en particular para desarrolladores con conocimientos de la API de Amazon S3, sobre la cual JetS3t basa su modelo. De esto último surge el inconveniente más grande de JetS3t y es que no fue concebido como framework. El agregado de Google Storage generó un set aislado de clases y diversos métodos que cambian su comportamiento utilizando “instanceof” para reconocer S3 de Google Storage. Esto provoca que implementar un nuevo protocolo no sea una tarea trivial y que la migración entre proveedores tenga dificultades y no sea un simple cambio de credenciales.

El proyecto tampoco cuenta con facilidades para migrar una aplicación desde otro framework, generando que el desarrollador deba codificar nuevamente su aplicación.

2.1.7. libcloud

El proyecto libcloud tiene por objetivo la generación de una interface única y simple a los servicios de tipo “compute” de distintos proveedores. Además, también cuenta con soporte básico para servicios de almacenamiento. Es principalmente un módulo para Python, pero también cuenta con desarrollo para Java, aunque por el momento con capacidades reducidas, ya que su desarrollo se encuentra congelado desde noviembre de 2010.

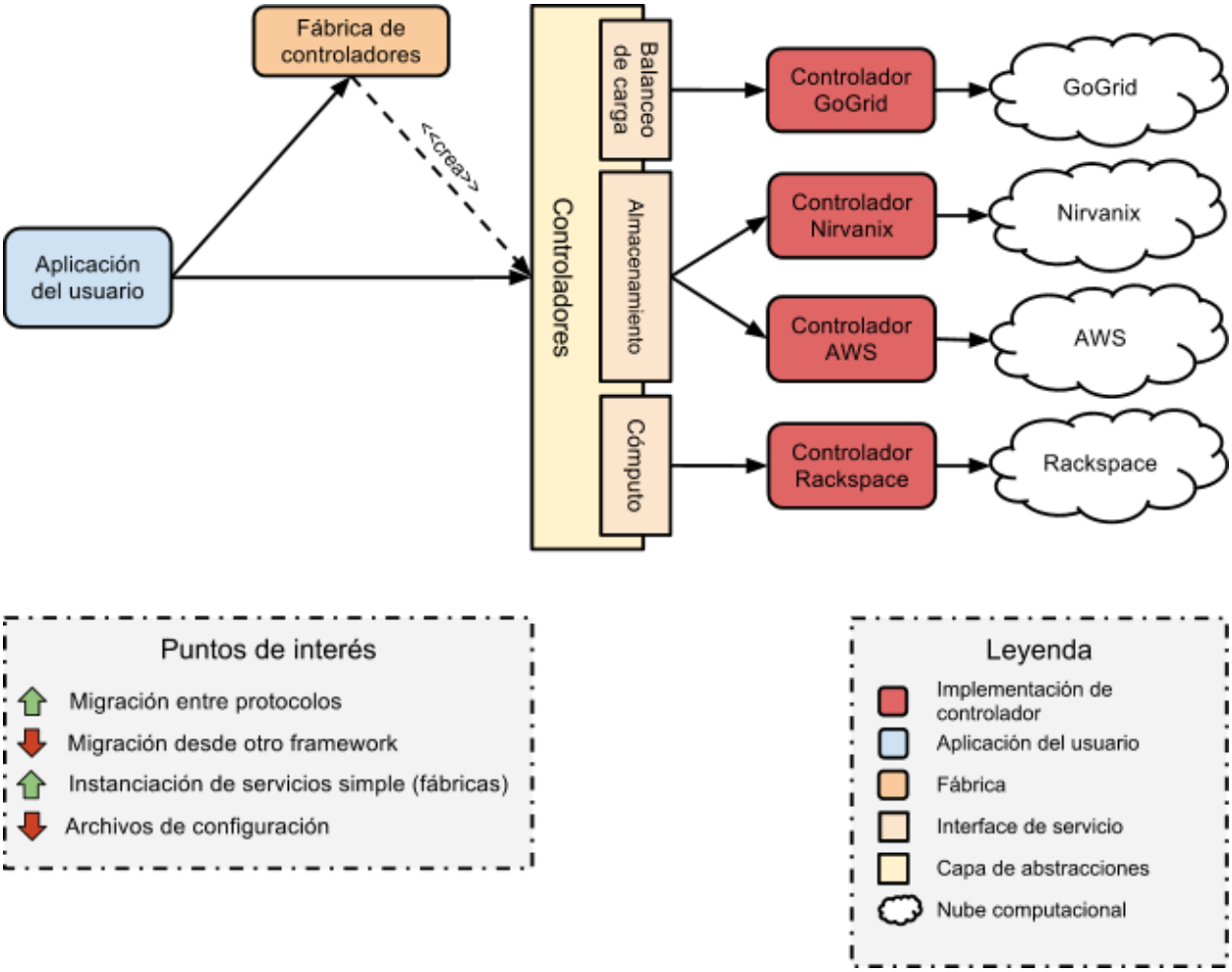

Figura 2.7 Vista de alto nivel de la arquitectura de libcloud.

Como se puede apreciar en la figura 2.7, la arquitectura de libcloud es simple. Por cada tipo de servicio soportado libcloud provee una interface genérica que cada proveedor debe implementar. Como complemento de esto, se provee una fábrica de controladores que gestiona la instanciación de los objetos

del servicio deseado. Esto evita que el desarrollador tenga que conocer detalles de bajo nivel de la construcción de cada objeto, aunque podría mejorarse con el agregado de archivos de configuración.

En cuanto a su funcionalidad, libcloud soporta todas las primitivas básicas de almacenamiento (subir, bajar, listar, buscar, borrar y extraer metadatos). Para comenzar a trabajar con libcloud para almacenamiento tan solo tenemos que crear instancias de alguna implementación de la interface de almacenamiento. Estas instancias presentan un set suficiente y claro de primitivas para trabajar con cada proveedor. Es igualmente simple implementar un nuevo servicio de almacenamiento. Para esto debe extenderse la interface provista implementando todos los métodos necesarios.

Si hablamos de migración entre proveedores dentro de libcloud podemos notar que si se utilizan correctamente las interfaces provistas el pasaje es muy simple. Tan solo se deben cambiar el tipo de la clase concreta y los parámetros necesarios del constructor.

Un punto que libcloud no contempla es su utilización dentro de una aplicación ya codificada con otro framework. Por ejemplo, si el usuario tiene implementada una aplicación para S3 con JetS3 y quiere utilizar Nirvanix con libcloud no tiene otra salida que volver a codificar su aplicación desde cero.

2.2. Conclusiones, puntos fuertes y débiles de cada herramienta.

Del análisis de cada herramienta se desprenden puntos fuertes y débiles que analizados en conjunto ofrecen conclusiones interesantes. La Tabla 2.1 presenta los detalles básicos de cada aplicación. La columna “Proyecto” presenta el nombre del framework. La columna “Activo?” indica si el proyecto sigue siendo mejorado en la actualidad. La columna “Soporte de almacenamiento para Java?” indica si un framework en particular soporta primitivas de almacenamiento en el lenguaje Java. La columna “Documentación” indica la calidad de la documentación de cada framework. Diremos que la documentación es mala cuando no se tiene ningún tipo de comentario, javadoc o manual para utilizar el framework. Una aplicación tendrá documentación regular si al menos posee javadocs para sus interfaces y será buena si tiene al menos javadocs y los desarrolladores proveen ejemplos de código.

Proyecto	Activo?	Soporte de almacenamiento para Java?	Documentación	URL
libcloud	No (Java)	Si	Buena	http://libcloud.apache.org
jClouds	Si	Si	Buena	http://code.google.com/p/jclouds/
Dasein	Si	Si	Mala	http://dasein-cloud.sourceforge.net/
Deltacloud	Si	No	Buena	http://incubator.apache.org/deltacloud/index.html
JetS3t	Si	Si	Buena	https://github.com/geemus/fog
CloudLoop	No	Si	Regular	https://cloudloop.dev.java.net/
Java-Storage	No	Si	Buena	http://www.simplecloud.org/api

Tabla 2.1 Información básica de los frameworks actuales.

Es interesante resaltar que tres de los proyectos evaluados no se encuentran en actividad (libcloud, CloudLoop y Java-Storage). Esto hace que no sean utilizados en ambientes de producción, al menos que el equipo que lo utilice también se encargue de mantenerlo. También puede apreciarse que salvo Deltacloud todos los frameworks soportan almacenamiento en Java. El dato negativo más destacable de Dasein es la inexistencia total de documentación sumada a una definición de interfaces poco claras con parámetros mal nombrados.

La tabla 2.2 profundiza sobre el soporte de almacenamiento de cada framework. La columna “Servicios de almacenamiento soportados” muestra que proveedores son soportados por cada framework. Las columnas englobadas en “Soporte de primitivas básicas” indican si cada framework soporta una primitiva en particular. “Subir” se refiere a escribir sobre el sistema de almacenamiento remoto. “Bajar” se refiere a poder leer datos remotos. “Listar” es la habilidad de poder detallar el contenido de una unidad de almacenamiento. “Buscar” consiste en poder realizar una búsqueda por clave o similitud de nombre dentro de un sistema de almacenamiento. “Borrar” se refiere a tener la habilidad de poder eliminar datos remotos. Y por último “Metadatos” se refiere a poder leer y escribir los metadatos de un elemento remoto, por ejemplo su tamaño en bytes o su hash MD5. Por último, la columna “Facilidad en el uso de las primitivas” muestra que tan simple resulta utilizar cada una de las primitivas mencionadas en cada framework. En este contexto diremos que utilizar las primitivas es “Fácil” si la interface provista tiene comentarios en los métodos y sus parámetros correctamente nombrados, caso contrario será considerada “Difícil” de utilizar.

Proyecto	Servicios de almacenamiento soportados	Soporte de primitivas básicas						Facilidad en el uso de las primitivas
		Subir	Bajar	Listar	Buscar	Borrar	Metadatos	
libcloud	Nirvanix, S3	Si	Si	Si	Si	Si	Si	Fácil
jClouds	Atmos, Azure, Rackspace, S3, CloudFiles, Google Storage, etc	Si	Si	Si	Si	Si	Si	Fácil
Dasein	S3, GoGrid, Rackspace, ReliaCloud, Azure, AT&T	Si	Si	Si	No	Si	Si	Difícil
Deltacloud	S3, Rackspace, Azure, Walrus	Si	Si	No	No	Si	No	Fácil
JetS3t	S3, Google Storage	Si	Si	Si	Si	Si	Si	Fácil
CloudLoop	Nirvanix, S3	Si	Si	Si	Si	Si	No	Fácil
Java-Storage	S3, Wuala	No	Si	No	No	No	No	Fácil

Tabla 2.2 Soporte para servicios de tipo almacenamiento

Como se podía suponer dado su alcance en el mercado todos los frameworks soportan Amazon S3 pero el resto de la oferta de proveedores está repartida entre los frameworks evaluados. Algunos soportan Google Storage mientras que otros no lo hacen y si soportan Nirvanix, Rackspace u otro. De estos datos se desprende que no existe un único framework que soporte todos los proveedores. También puede apreciarse que la mayoría de los productos evaluados soportan todas las primitivas básicas de almacenamiento. Es notable que Java-Storage solo soporte descargar archivos y que DeltaCloud solo permita subir, descargar o borrar. Por otro lado, podemos ver que en general el uso de las primitivas es sencillo, a excepción de Dasein que dada la inconsistencia de sus interfaces y su escasa documentación puede tornarse problemático.

Los datos más relevantes se presentan en la tabla 2.3. La columna “Facilidad para implementar servicios” indica que tan sencillo resulta implementar un nuevo servicio de un nuevo proveedor en el framework. Diremos que es “Fácil” si las interfaces a implementar cuentan con javadocs, los parámetros tienen nombres claros y existe un ejemplo que pueda seguirse. Si alguna de estas condiciones no se cumple diremos que es “Difícil” implementar un nuevo servicio en el framework. La columna “Facilidad para migrar entre protocolos” indica que tan sencillo es migrar una aplicación funcionando en un protocolo X a un protocolo Y, ambos soportados por el framework. Diremos que es “Fácil” si lo único que debe

cambiarse es la instanciación de la clase que modela el servicio y si esta respeta una interface que puede utilizarse libremente sin atarse a una implementación particular. En caso contrario diremos que este tipo de migraciones es "Difícil". La columna "Provee mecanismos para migrar desde otro framework" indica si un framework es capaz de utilizar como entrada código construido para otro framework. La columna "Provee archivos de configuración" muestra si un framework en particular puede configurarse por medio de archivos sin tener que cambiar el código fuente. Por último, la columna "Provee fábricas para instanciar servicios" indica si un framework le facilita al desarrollador la instanciación de sus objetos o si este tiene que instanciar manualmente todos los objetos.

Proyecto	Facilidad para implementar servicios	Facilidad para migrar entre protocolos	Provee mecanismos para migrar desde otro framework	Provee archivos de configuración	Provee fábricas para instanciar servicios
libcloud	Fácil	Fácil	No	No	No
jClouds	Fácil	Fácil	No	No	No
Dasein	Difícil	Fácil	No	No	No
Deltacloud	Fácil	Difícil	No	No	No
JetS3t	Difícil	Difícil	No	No	No
CloudLoop	Fácil	Fácil	No	Si	Si
Java-Storage	Fácil	Fácil	No	No	No

Tabla 2.3 Capacidades de migración, configuración e inclusión de nuevos servicios.

El dato más destacable es que ninguna de las herramientas posee facilidades para migrar desde otro framework. Esto obliga a que el desarrollador deba recodificar su aplicación en caso de querer migrar. Otro de los datos interesantes es que salvo CloudLoop, ninguna de las herramientas trabaja la instanciación y configuración de servicios por medio de archivos y fábricas o mecanismos similares. En estos casos, el código debe modificarse cuando se desea cambiar de proveedor, credenciales, etc. También se puede apreciar que en general se puede decir que es simple implementar nuevos servicios en todos los frameworks sacando Dasein y JetS3t. Estos últimos productos tienen deficiencias en la definición de sus interfaces y/o documentación que hacen que implementar un nuevo servicio no sea una tarea trivial.

CAPÍTULO 3

3. ENFOQUE

Durante el desarrollo del capítulo anterior se analizaron los puntos fuertes y débiles de cada una de las herramientas actuales. El resultado de este análisis evidenció una serie de problemas comunes a todas las herramientas que se relacionan fundamentalmente con las capacidades de migración. Por un lado podemos distinguir la migración de aplicaciones entre servicios, por ejemplo de Amazon S3 a Google Storage. Este tipo de migración suele ser tenida en cuenta por los diseñadores de las herramientas para *Cloud Computing*, pero no se encuentra disponible un mecanismo que realmente facilite la tarea al usuario que en muchos casos debe volver a codificar para adaptarse a los objetos del nuevo servicio. Por otro lado, se requiere la capacidad de migrar de aplicaciones entre herramientas o frameworks para *Cloud Computing*. Un ejemplo de esto sería tener una aplicación codificada con el SDK de Amazon S3 y querer comenzar a utilizar Google Storage provisto por jClouds. Ninguna de las herramientas disponibles en la actualidad considera este tipo de migración y esto deriva en que el desarrollador deba recodificar la aplicación para utilizar la nueva plataforma.

Si analizamos una aplicación de *Cloud Computing* actual, vamos a notar que la interacción con el proveedor de servicios está planteada de dos maneras. La primera, que suele ser el enfoque inicial desarrollado por cualquier aplicación, es utilizar directamente la API del proveedor de servicios. La segunda, ya más elaborada, es utilizar un framework que soporte internamente la API del proveedor de servicios ganando cierto grado de flexibilidad a la hora de querer migrar. Sin embargo, no se tiene la garantía de que el framework elegido vaya a soportar al proveedor que se decida utilizar en el futuro.

De acuerdo a la problemática planteada, el objetivo de este trabajo fue crear un framework, llamado Aether, que reduzca el esfuerzo de codificación para migrar una aplicación entre proveedores de servicios de *Cloud Computing*. Si queremos lograr que el usuario tenga costos mínimos para migrar no existe otra manera que accionar directamente sobre el punto compartido entre cualquier aplicación de *Cloud Computing*: la comunicación entre la aplicación y el proveedor. En términos técnicos se debe abordar la problemática asociada al monitoreo y análisis de las invocaciones de una aplicación a las APIs de los proveedores de servicios. Una vista conceptual de cómo Aether aborda esta problemática se muestra en la figura 3.1.

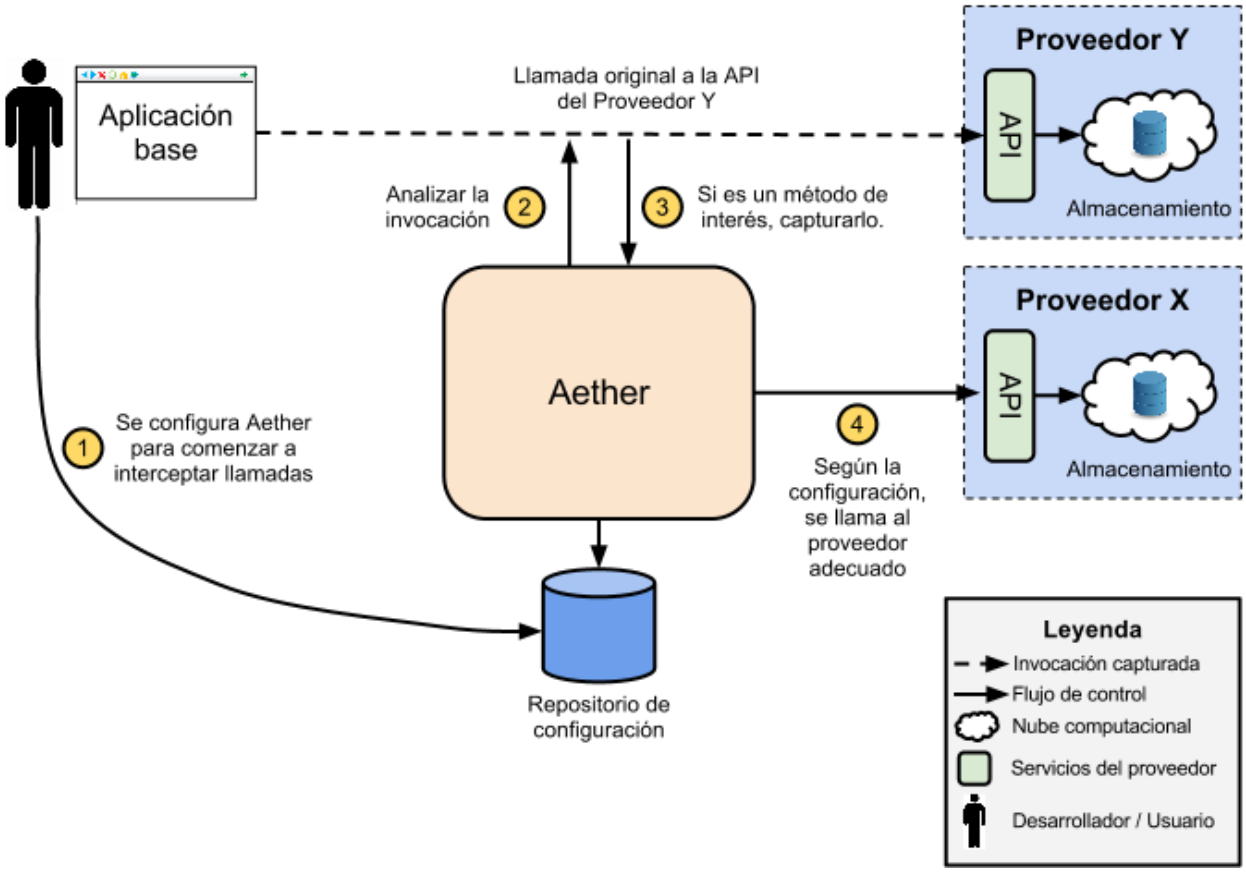

Figura 3.1 Vista Conceptual de Aether.

Como puede apreciarse, el primer paso para utilizar Aether consiste en configurar el framework para que comience a interceptar las llamadas de la aplicación del usuario (1). Tan solo se necesitan las credenciales del proveedor que se desea utilizar e información sobre la aplicación del usuario para que Aether reconozca que métodos debe monitorear. Una vez configurado, Aether analiza cada una de las llamadas que la aplicación del usuario (aplicación base) realiza a la API del proveedor del servicio *Cloud* (2). Este modelo sencillo tiene como base principal al concepto de "reflexión" o "reflexión computacional" (Forman & Forman, 2005) (Sobel & Friedman, 2013).

Definimos reflexión como la habilidad de un programa de computación para examinarse y modificar su propia estructura y comportamiento en tiempo de ejecución. En otras palabras, un lenguaje de programación que soporta reflexión provee una serie de características de tiempo de ejecución que posibilitan entre otras cosas:

- Detectar y modificar construcciones de código en tiempo de ejecución.

- Convertir un string que representa el nombre de una clase o método en una referencia o una invocación a esa clase o método.
- Evaluar un string como si fuese código fuente en tiempo de ejecución.

En un lenguaje orientado a objetos como Java, esto significa tener la habilidad de inspeccionar clases, interfaces, métodos, campos, etc., sin tener pleno conocimiento de sus nombres en tiempo de compilación (Oracle, 2013). Más interesante para nuestro caso es la habilidad de cambiar el comportamiento de clases o métodos según decisiones tomadas en tiempo de ejecución.

Haciendo uso de este tipo de funcionalidades Aether inspecciona cada método ejecutado por la aplicación del usuario hasta encontrar uno que coincida con la configuración seleccionada. Cuando se detecta la ejecución de uno de estos métodos se utiliza reflexión para cederle el control a Aether y modificar el comportamiento de la invocación en tiempo de ejecución (4). Las modificaciones consisten en traducir estas llamadas a las primitivas equivalentes provistas por el proveedor configurado en Aether. Por ejemplo, si se estaba ejecutando el método "subir archivo" de Amazon S3 y el usuario desea utilizar Google Storage, esta traducción resultaría en el método equivalente de la API de Google. Trabajar de este modo hace posible el intercambio de proveedores en tiempo de ejecución sin la necesidad de modificar el código de la aplicación.

Internamente, Aether realiza sus traducciones valiéndose de dos módulos. Por un lado, tenemos a los protocolos de traducción. Cada uno de estos protocolos implementa las reglas para traducir de un modelo particular al modelo interno de Aether. Por el otro lado, se cuenta con una capa de abstracción de servicios que permite trabajar de manera uniforme con los servicios de distintos proveedores. Estos dos componentes se describen en las secciones 3.1.1 y 3.1.2 respectivamente.

3.1.1. Capa de abstracción de servicios en Cloud

Los servicios de *Cloud Computing* se agrupan en categorías como almacenamiento y computo. A su vez, para cada categoría existe una gran variedad de proveedores que implementan servicios similares que, si bien difieren en detalles, mantienen una línea según el tipo de servicio. Teniendo esto en mente, podemos pensar en abstraer la funcionalidad común según el tipo se servicio. Esta es la función de la capa de abstracción de servicios en *Cloud* de Aether. La idea es que cada implementación de servicios que Aether soporta debe respetar una serie de lineamientos definidos por esta capa con el objetivo de hacer

compatibles a los servicios de distintos proveedores. Por ejemplo, si hablamos de servicios de almacenamiento esta capa debe ser capaz de exponer una interface uniforme para funcionalidades tales como subir y bajar archivos. Puertas adentro, esta capa se encarga de la invocación final al proveedor deseado realizando las traducciones que sean pertinentes.

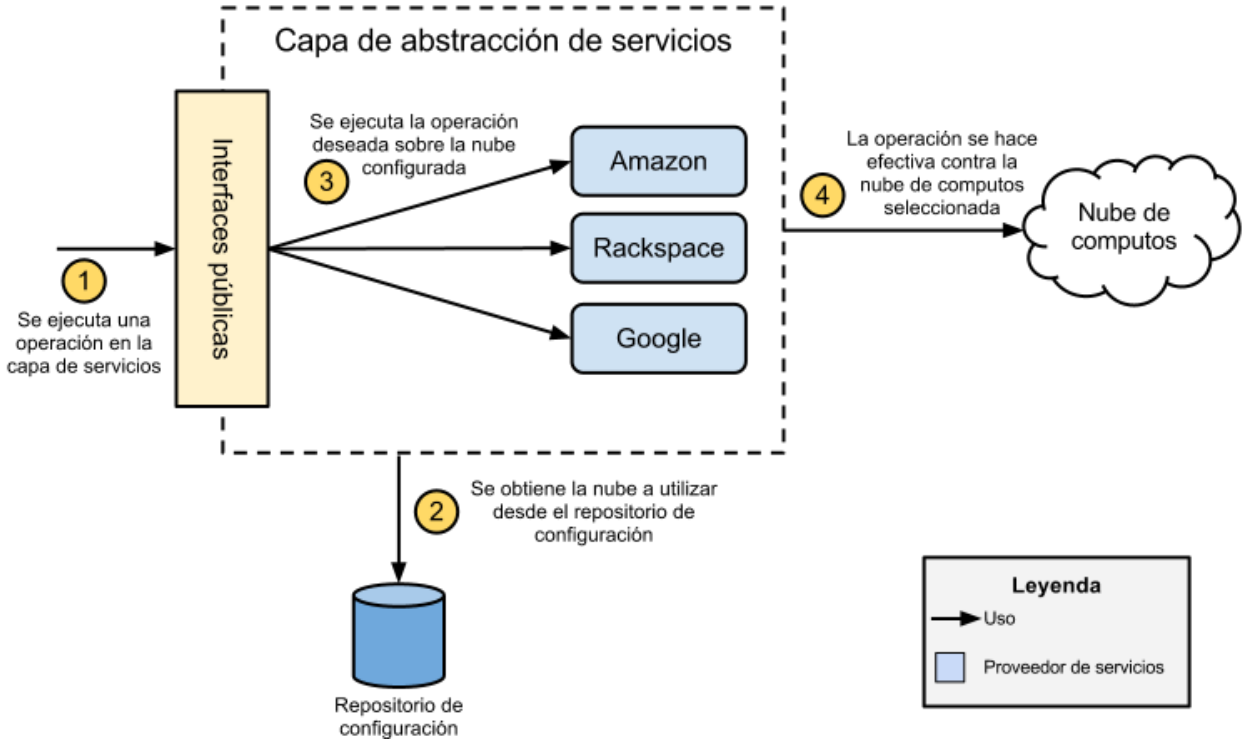

Figura 3.2 Funcionamiento de la capa de abstracción de servicios.

Como se puede apreciar en la figura 3.2, descomponemos a la capa de abstracción de servicios en dos unidades, las interfaces públicas y las implementaciones de los proveedores particulares. Las interfaces públicas son las que proveen el punto de acceso estable y uniforme a la capa. Una vez recibida una invocación se debe chequear la configuración introducida en el repositorio de configuración. Esto incluye datos que se utilizaran para redireccionar la llamada al proveedor de servicios que sea necesario. Por ejemplo, si la configuración especifica que debe utilizarse Amazon S3 para tratar invocaciones de almacenamiento se deberán proveer las credenciales necesarias para poder trabajar con el servicio. Una vez obtenida la configuración tan solo se debe invocar a la nube necesaria a través de la implementación del proveedor seleccionado. Esta mecánica facilita el uso de la capa ya que no se debe tener conocimiento de quien va a realizar la invocación efectiva y puede cambiarse el comportamiento por medio de configuración.

En la próxima sección se describe como los protocolos de traducción de Aether utilizan esta capa de abstracción de servicios para realizar su tarea.

3.1.2. Protocolos de traducción de métodos

En el contexto de Aether, un protocolo de traducción es una unidad funcional que define métodos a interceptar y como debe realizarse su traducción. La idea consiste en contar con un traductor que sea capaz de transformar una invocación en un set de invocaciones equivalentes dentro de Aether. Para lograr esto, estos traductores deben hacer uso de la capa de abstracción de servicios provista por Aether. De esta manera, los protocolos no generan dependencia con un proveedor en particular y pueden cambiar su comportamiento dependiendo de la configuración introducida por el usuario para la capa de abstracción de servicios. Cabe destacar que los protocolos de traducción actúan sobre una API, una biblioteca o un framework de terceros. A partir de la creación del protocolo, cualquier aplicación del usuario que utilice alguno de estos elementos será soportada automáticamente para trabajar con Aether.

Para ejemplificar el funcionamiento de este mecanismo pensemos en un protocolo para interceptar y transformar invocaciones a S3 desde el framework jClouds. El protocolo correspondiente tendría la lógica expresada en la figura 3.3.

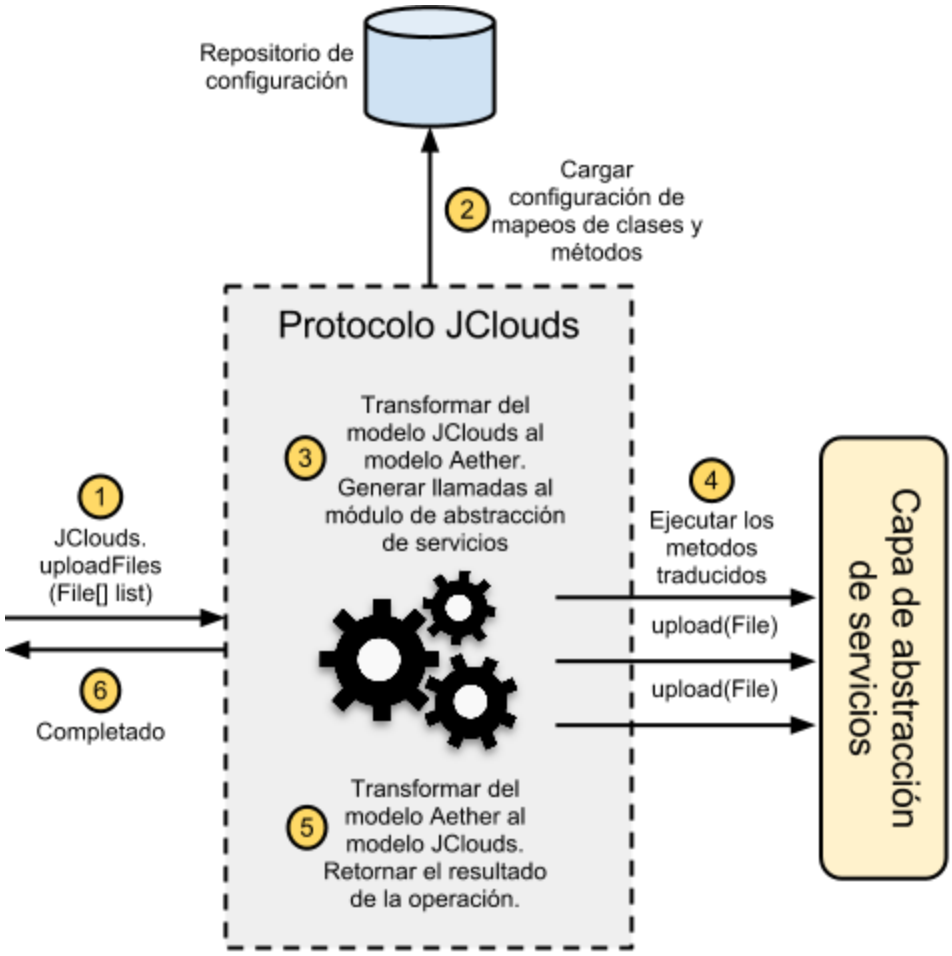

Figura 3.3 Funcionamiento de un protocolo para jClouds.

El funcionamiento de este protocolo es simple. Una vez detectada una invocación al método de interés (1) (uploadFiles en el ejemplo) se debe transformar la invocación original en un set equivalente de invocaciones a la capa de abstracción de servicios (2)(3). Esto no solo incluye mapear métodos uno a uno sino que también puede incluir transformaciones de parámetros y otras operaciones más complejas. Aether puede realizar esto gracias al uso de reflexión que nos permite modificar el código original de la invocación en tiempo de ejecución. Siguiendo el flujo de control, ya contamos con el método traducido por lo que solo resta ejecutarlo contra la capa de abstracción de servicios (4). Por último, luego de completar la ejecución el protocolo realiza una nueva traducción, esta vez desde el modelo interno de Aether al modelo de jClouds (5) para luego retornar el control al llamador original (6). Esta última traducción completa el circuito que garantiza un modelo transparente desde el punto de vista de la aplicación original que nunca debió enterarse que el código estaba siendo interceptado y sus llamadas modificadas. Cabe destacar que todo esto se realiza por medio de los mecanismos de reflexión presentados anteriormente.

3.3 Conclusiones

En este capítulo se describió el esquema con el que Aether logra mitigar los problemas de migración existentes en las herramientas actuales. La idea consistió en generar un framework que permite la migración entre proveedores de servicios en *Cloud* de aplicaciones ya codificadas. Este framework se vale de dos unidades funcionales para cumplir su tarea, la capa de abstracción de servicios y los protocolos de traducción. La primera es la encargada de homogeneizar las invocaciones hacia distintos servicios de *Cloud Computing*. Es decir, esta capa se encarga de administrar las invocaciones salientes de Aether. Por último, los protocolos se encargan de capturar métodos de otras herramientas mediante mecanismos de reflexión. Cada uno de los métodos capturados se traduce a un set equivalente de operaciones provistas por la capa de abstracción de servicios para que puedan ejecutarse en el contexto de Aether. Con la unión de estos dos elementos logramos que Aether sea capaz de tomar invocaciones apuntadas a un proveedor A y redirigirlas a un proveedor B de manera transparente al código original, ya que no se vio modificado. Cabe destacar que este mecanismo puede aplicarse a cualquier tipo de servicio en *Cloud*, ya sea almacenamiento, colas distribuidas, procesamiento u otro.

CAPÍTULO 4

4. DISEÑO E IMPLEMENTACIÓN DE AETHER

En éste capítulo se detalla el diseño e implementación del framework Aether. Este framework cuenta con tres módulos bien definidos que interactúan entre si siguiendo el estilo arquitectónico de capas. La figura 4.1 se presenta los módulos y la interacción entre ellos. En la capa más baja se encuentra el Módulo Cargador de Adapters (aether-loader), encargado de interceptar las llamadas a métodos de otros frameworks y realizar la invocación al correspondiente del adapter indicado; esto lo logra haciendo uso de las funcionalidades que posee para detectar la carga de clases y modificarlas en tiempo de ejecución según sea necesario. En la capa intermedia se encuentra el Módulo de Adapters (aether-adapters), el cual provee las facilidades para convertir las llamadas de una aplicación desarrollada utilizando otro framework a llamadas equivalentes de Aether; básicamente está formado por sub-proyectos independientes, uno para cada herramienta o framework desde el cuál se desee realizar la migración. Por último, en la capa superior se encuentra el Módulo de Abstracción de Servicios (aether-core), el cual se encarga de abstraer los servicios prestados por diferentes proveedores y dar acceso a estos mediante una única interface.

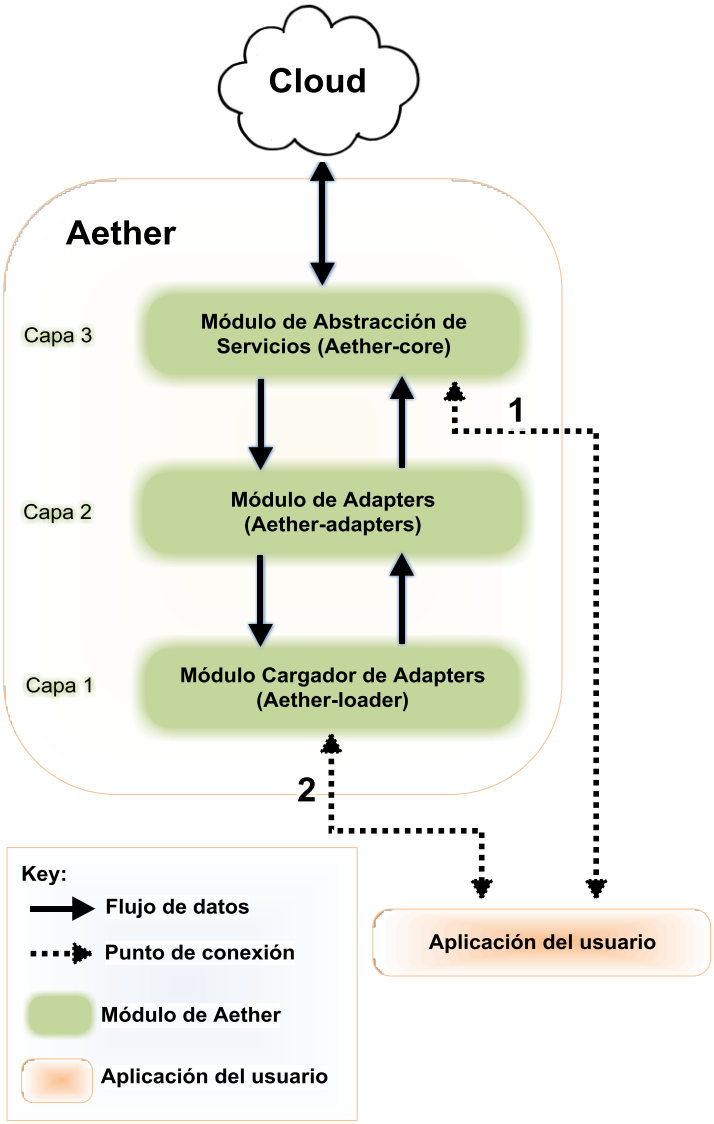

Figura 4.1 Módulos de Aether y su interacción.

Como se puede apreciar, existen dos de utilizar el framework. La primera de ellas es haciendo uso del Módulo de Abstracción de Servicios de manera directa (1). Esto es útil para los casos en que se desarrolle la aplicación de cero utilizando este framework. La segunda opción es hacer uso de éste por medio del Módulo Cargador de Adapters (2). De esta manera, el usuario puede utilizar Aether en aplicaciones desarrolladas sobre algún otro framework. En estos casos, el Cargador de Adapters intercepta las llamadas al framework original y las redirige al adapter de Aether correspondiente.

Con respecto a las herramientas utilizadas para el desarrollo del framework podemos mencionar al administrador de dependencias Maven (Apache, 2013), el cual posee un modelo de configuración de

construcción simple basado en un formato XML (W3C, 2013) permitiendo, entre otras cosas, una sencilla administración de dependencias. Adicional a esta herramienta fue necesario utilizar un IDE para el desarrollo del código. El entorno de desarrollo que se utilizó fue Eclipse (Eclipse Foundation, Inc., 2013) ya que facilita la integración para el manejo de dependencias con Maven además de permitir el agregado de diversos complementos por medio plugins disponibles en la web.

En las secciones siguientes se profundiza sobre cada uno de los módulos, detallando las principales decisiones referidas al diseño de Aether.

4.1 Módulo de Abstracción de Servicios (Aether-core)

El Módulo de abstracción de servicios, como su nombre lo indica, tiene como función principal abstraer las interfaces de diferentes proveedores para diversos tipos de servicios. En este contexto, se debe considerar la existencia de servicios de distintas naturalezas como pueden ser almacenamiento, cómputo y colas distribuidas.

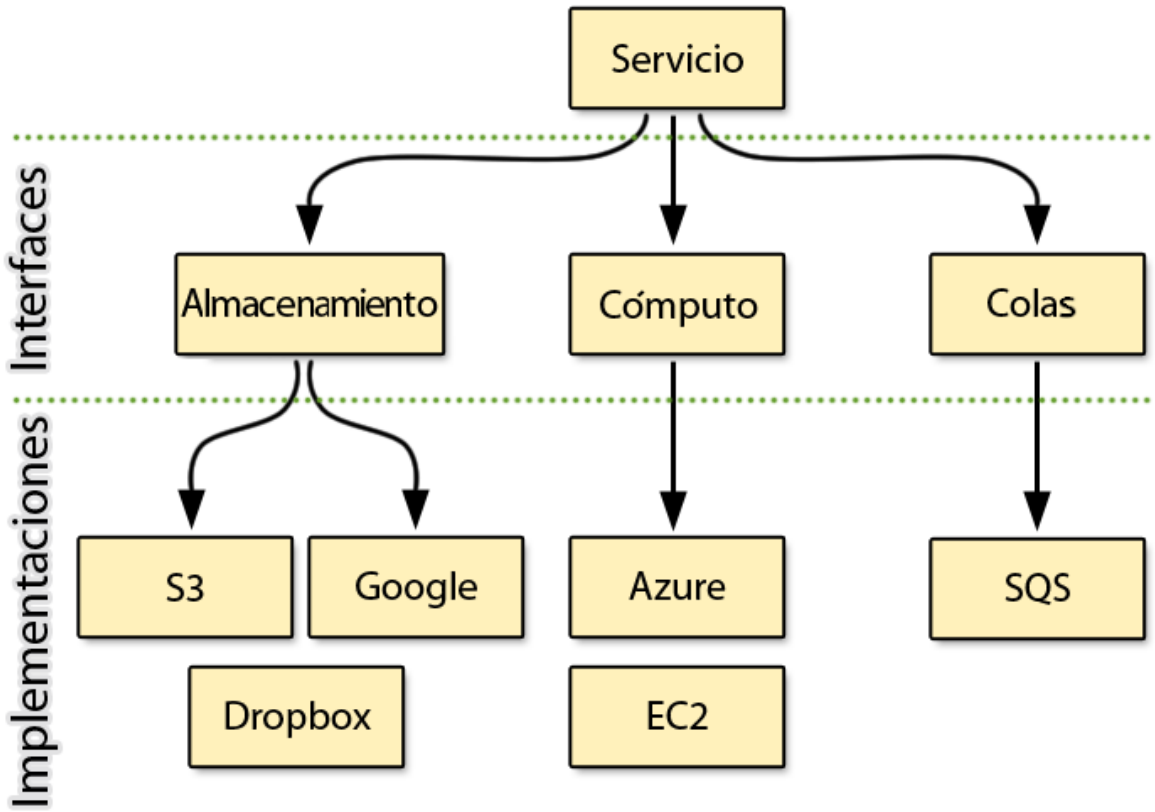

Figura 4.2 Vista abstracta del Módulo de Abstracción de Servicios.

En la figura 4.2 se muestra de forma abstracta la organización de éste módulo. Podemos apreciar que se definió una interfaz independiente para cada uno de los tipos de servicio, es decir, una interfaz para el Almacenamiento, otra para Cómputo y una última para Colas. De esta manera se agrupan los servicios de funcionalidad similar bajo una interface homogénea que facilita el uso y migración entre ellos. Cada una de estas interfaces contiene todos los métodos requeridos para la utilización del tipo de servicio que se está tratando. Por ejemplo, para implementar un servicio de almacenamiento la interface genérica contiene métodos para subir, bajar, eliminar y copiar archivos; para un servicio de colas contiene los métodos para encolar y obtener tareas y para un servicio de cómputo, la interface genérica contiene métodos para crear y administrar máquinas virtuales. Trabajar con estas interfaces como base facilita la inclusión de nuevos servicios concretos como S3 o SQS (Amazon, 2013). Estas implementaciones pueden verse en la parte inferior del gráfico en dónde se aprecian los servicios de S3, Google y Dropbox (Dropbox, 2013) para el almacenamiento, Azure (Microsoft, 2013) y EC2 (Amazon, 2013) para el cómputo y SQS para el servicio de colas.

Trabajar de este modo es interesante debido a que en el mercado actual existe una gran diversidad de proveedores para cada uno de los tipos de servicios de *Cloud Computing*. Cada uno de estos proveedores posee ciertas particularidades que lo distingue del resto como por ejemplo límites de almacenamiento, velocidad de subida y descarga de archivos, velocidad de procesamiento, costos de cada servicio, etc. Estas diferencias pueden conducir a que los usuarios deseen migrar su aplicación a un proveedor diferente del utilizado. Una migración de este estilo resulta incómoda y además en una pérdida de tiempo al surgir la necesidad de recodificar la aplicación para adaptarse a las interfaces del nuevo proveedor. Debido a estos problemas, se diseñó Aether para facilitar el cambio entre proveedores de forma dinámica y sin necesidad de alterar el código fuente de la aplicación. Este diseño se basó en dos ejes fundamentales: la configuración del framework por parte del usuario y la funcionalidad para interpretar la configuración y crear los servicios indicados.

Para la configuración del framework por parte del usuario se utilizó un archivo con formato XML en el cual se especifican los servicios y parámetros necesarios para poder instanciarlos. La estructura del XML es muy sencilla ya que se distinguen sólo tres elementos. Supongamos estar desarrollando una aplicación para realizar un backup automático de datos utilizando Aether para almacenar los datos utilizando los servicios de Google Storage. A continuación en la figura 4.3 se presenta la estructura del archivo XML de configuración que se debería definir para este caso.

```
<storageServices>
      <storageService class="aether.core.services.storage.imp.GoogleV1StorageService">
          <parameter key="container" value="container1"/>
           <parameter key="googleStorageAccessKey" value="MyKey"/>
           <parameter key="googleStorageSecretKey" value="MySecretKey"/>
      </storageService>
</storageServices>
```

Figura 4.3 Estructura del archivo XML de configuración para el servicio de almacenamiento.

El primer elemento define la sección de servicios disponibles correspondientes a un tipo específico, por ejemplo para el servicio de almacenamiento se definió con el nombre de storageServices. El segundo elemento, llamado storageService, posee un atributo "class" por medio del cual se indica el objeto que se deberá crear y configurar al instanciar el servicio. Este elemento tiene además subelementos llamados "parameter" los cuales deben tener obligatoriamente dos atributos: key y value. El valor del atributo "key" corresponde al nombre del parámetro que reconocerá el framework, mientras que el texto del atributo "value" indica su valor.

Como puede apreciarse, en el archivo de configuración del framework se encuentra definida la clase y parámetros que se deben utilizar para interactuar con el proveedor del servicio, con lo cual una migración a un proveedor diferente sólo recae en la modificación de los datos del archivo estableciendo la nueva clase que permitirá manejar las llamadas a los servicios del nuevo proveedor y los parámetros necesarios para poder interactuar con él.

Para poder interpretar el archivo de configuración y poder crear los servicios indicados el framework trabaja con dos fábricas llamadas ServiceFactory y ServiceParser. A continuación en la figura 4.4 se presenta el diseño de clases que hace posible esto y sobre el cuál se indicará la funcionalidad de cada componente.

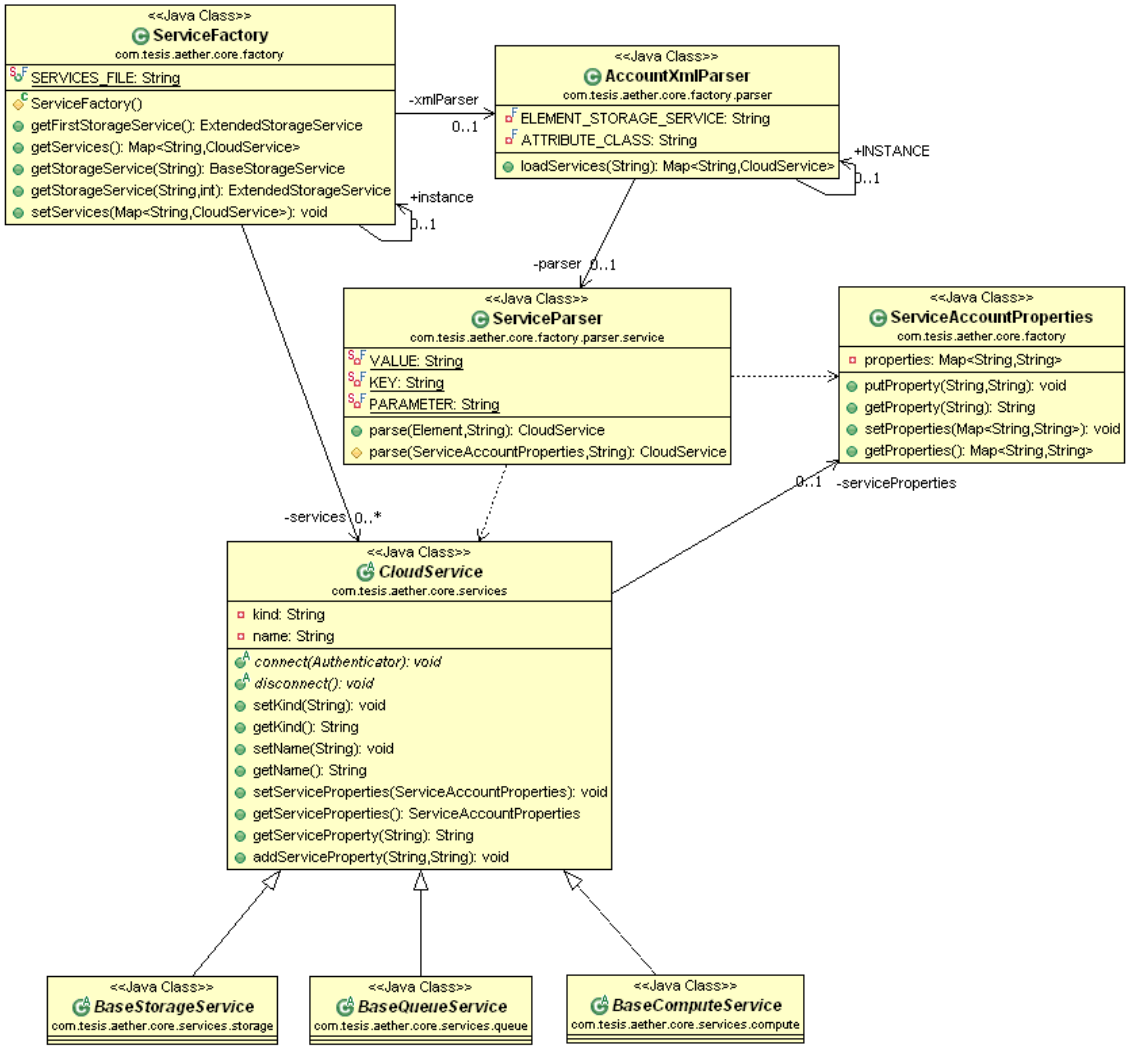

Figura 4.4 Diagrama de clases para la abstracción de servicios.

Como puede apreciarse en la figura anterior, el problema se atacó utilizando dos fábricas (o factories) para crear y mantener disponibles cada uno de los servicios. De esta manera el usuario no necesita conocer detalles de implementación de cada proveedor particular ya que las fábricas "ServiceFactory" y "ServiceParser" lo encapsulan. Básicamente la función de ServiceFactory es gestionar la carga de los servicios que se encuentren especificados en el archivo de configuración y mantenerlos disponibles para cuando se soliciten, mientras que la principal función de ServiceParser es crear las instancias de cada uno de los servicios y retornarlos según sea necesario. A continuación se describe más en detalle las responsabilidades de cada una de las clases presentes en el diagrama anterior.

La clase *ServiceFactory* es el punto de acceso para el usuario y tiene como función gestionar la carga de los servicios que se encuentren especificados en el archivo XML de configuración presentado

anteriormente y mantenerlos disponibles para cuando se soliciten. Para el caso de ejemplo anterior el servicio que cargará es el indicado en atributo “class” del elemento “storageService”: “aether.core.services.storage.imp.GoogleV1StorageService”. *ServiceFactory* se encuentra implementada respetando el patrón de diseño Singleton con el objetivo de garantizar que sólo exista una instancia y de ésta manera proporcionar un punto de acceso global a ella. Es útil destacar que las instancias de los servicios retornados por este factory son únicas, por lo que diferentes llamados a los métodos "*get*" con los mismos parámetros retornan las mismas instancias. Esto es importante para poder gestionar la concurrencia en los servicios provistos, ya que de otra manera seria imposible.

La funcionalidad de las tres clases restantes (CloudService, AccountXmlParser y ServiceParser) es sencilla. CloudService contiene el comportamiento genérico de un servicio en la nube tales como facilidades para conexión y desconexión, las propiedades del servicio que fueron cargadas desde el archivo XML de configuración definido por el usuario, el tipo de servicio y su nombre. AccountXmlParser se encarga de cargar y analizar el archivo XML de configuración. Para esto, AccountXmlParser, lee cada nodo del XML e invoca a ServiceParser para que construya la instancia efectiva del servicio. Debido a que ServiceParser no conoce a priori los objetos que puede crear, no es posible realizar la instanciación de estos de la forma tradicional. Para resolver este problema se utilizó una API de java llamada “Reflection” que permite crear instancias de objetos indicando un nombre de paquete y clase. Por medio de este mecanismo, Aether puede crear instancias específicas de “CloudService” y posterior a esto agregarle las propiedades indicadas en el archivo de configuración. Debido al uso de reflexión para la carga de clases es de suma importancia que la configuración presente en el archivo sea la correcta.

Los servicios que se crean deben pertenecer a algún subtipo de CloudService, los cuales se aprecian en la figura con los nombres BaseStorageService, BaseQueueService y BaseComputeService y corresponden a los servicios de *Cloud Computing* de almacenamiento, colas y cómputo respectivamente. A su vez, cada uno de estos tres servicios posee implementaciones específicas ya que las clases anteriores sólo describen la estructura básica que debe respetar cada implementación particular. A continuación en la figura 4.5 se presentan como ejemplo algunas de estas implementaciones particulares para el servicio de almacenamiento y las superclases involucradas.

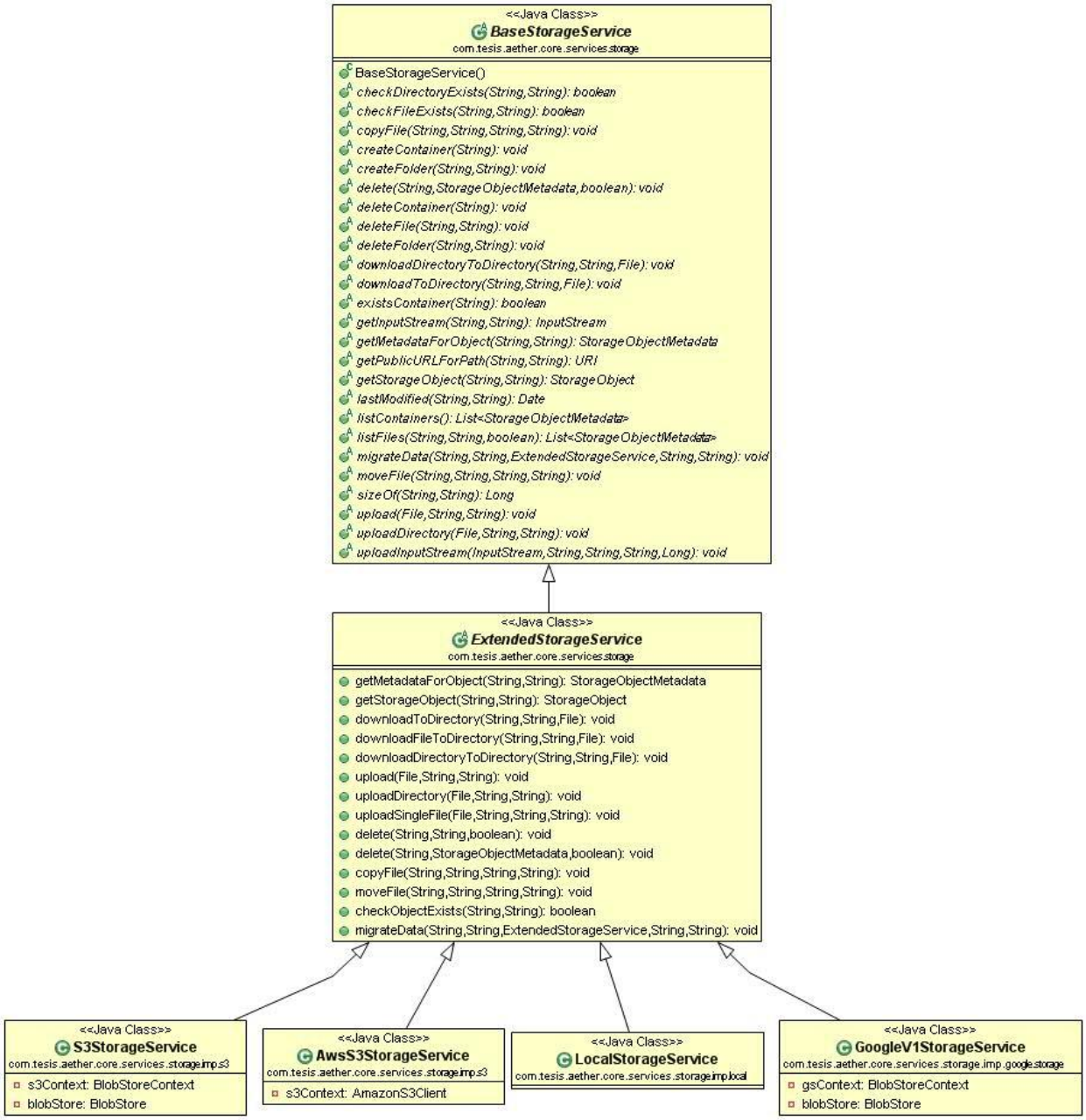

Figura 4.5 Especificaciones del servicio de almacenamiento.

En la Figura 4.5 puede verse que ExtendedStorageService extiende la clase básica del servicio de almacenamiento (BaseStorageService) agregando comportamiento extra. Luego, S3StotageService, AwsS3StorageService, LocalStorageService y GoogleV1StorageService extienden a esta última para implementar el comportamiento correspondiente a cada servicio particular. En estos casos, las implementaciones particulares que pueden apreciarse corresponden respectivamente a los servicios de

almacenamiento S3 y AWS de Amazon, un servicio de almacenamiento local (por medio del uso de File) y por último un servicio de almacenamiento en *Cloud* de Google.

Para finalizar con el Módulo de Abstracción de Servicios presentaremos a continuación en la figura 4.6 un diagrama de secuencia en donde por medio de un ejemplo de instanciación de un servicio se puede ver de forma gráfica la interacción entre las clases mencionadas hasta el momento.

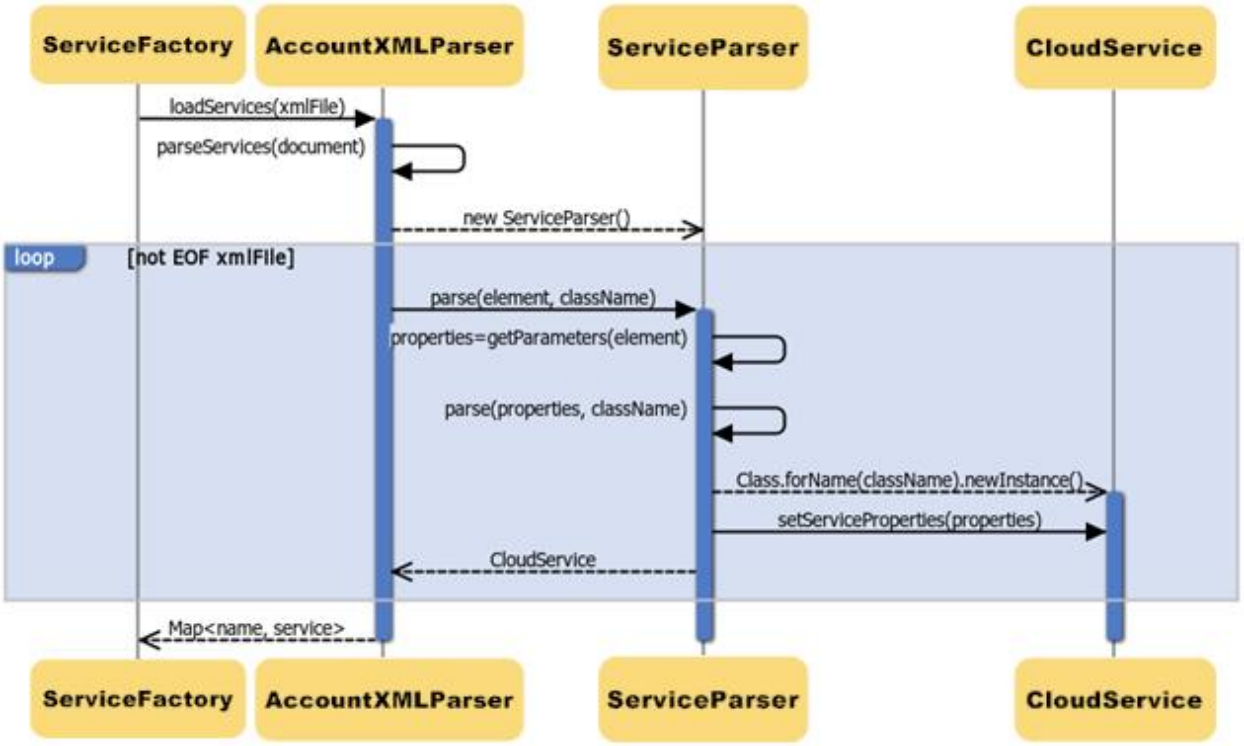

Figura 4.6 Secuencia de instanciación de un servicio.

En la figura anterior puede verse que al momento de crear el Singleton de "*ServiceFactory*", se cargan los servicios desde el archivo *xml* de configuración. Para esto se realiza una llamada al método "*loadServices*" correspondiente a la clase "*AccountXmlParser*" pasándole como parámetro la ruta al archivo de configuración. El parseo del XML es sencillo, AccountXmlParser simplemente lee cada nodo del XML e invoca a ServiceParser mediante el método parse(element, className) para que construya la instancia efectiva del servicio. La clase ServiceParser lee todos los elementos ingresados por el usuario para un servicio particular en el XML de configuración haciendo uso de sus métodos getParameters(element) y parse(properties, className) para luego, en base al atributo "class" del XML, construir una nueva instancia del servicio por medio de mecanismos de reflexión. Adicionalmente, cada elemento "parameter" del XML es leído e inyectado en la instancia del servicio recién creada.

4.2 Módulo de adapters para frameworks de terceros (Aether-adapters)

Una vez desarrollada la aplicación del usuario utilizando el módulo de abstracción de servicios de Aether, es sencillo realizar el cambio de proveedor, pero ¿qué sucedería si el usuario ya posee una aplicación funcional que utiliza tecnología ajena a ésta plataforma y desea migrar a Aether? Por ejemplo, supongamos que la aplicación del ejemplo anterior ya se encuentra implementada utilizando el framework JetS3t para acceder a Amazon S3 y se desea migrarla a Google Storage con nuestra plataforma. Al usuario difícilmente le caería bien la idea de volverla a codificar utilizando el módulo de abstracción de servicios ya que perdería una base de código estable y testeado. Es por este motivo que se agregó al framework una capa de adapters para tecnologías ajenas a la plataforma. Con esto se logra que un usuario pueda utilizar Aether de manera transparente, manteniendo su base de código actual desarrollada para otro producto.

Cada uno de los adapters del módulo realiza traducciones entre una tecnología específica y Aether, haciendo uso de las interfaces del módulo de abstracción de servicios. Siguiendo el ejemplo, la plataforma proveerá un adapter para el framework JetS3t de tal manera que la invocación a un método de JetS3t será traducido a un set de llamadas equivalentes de Aether. Es importante destacar que cada adapter debe tener en cuenta la traducción de objetos desde y hacia las tecnologías de terceros, es decir, realizar la serie de llamadas correspondientes al framework Aether y luego transformar los resultados para brindar la salida correspondiente al framework original. Para lograr esto se utilizó el patrón de diseño “Adapter” (también conocido como “Wrapper”).

Debido a que todos los adapters poseen ciertas características comunes, se tomó la decisión de crear una clase abstracta sobre Aether-core que agrupe estas funcionalidades. Esta clase debe ser extendida por cada uno de los adapters que se implementen para poder mantener la consistencia entre ellos. A continuación en la figura 4.7 se presenta un diagrama de clases simplificado en el cuál se muestra la clase abstracta y las implementaciones particulares de algunos adapters.

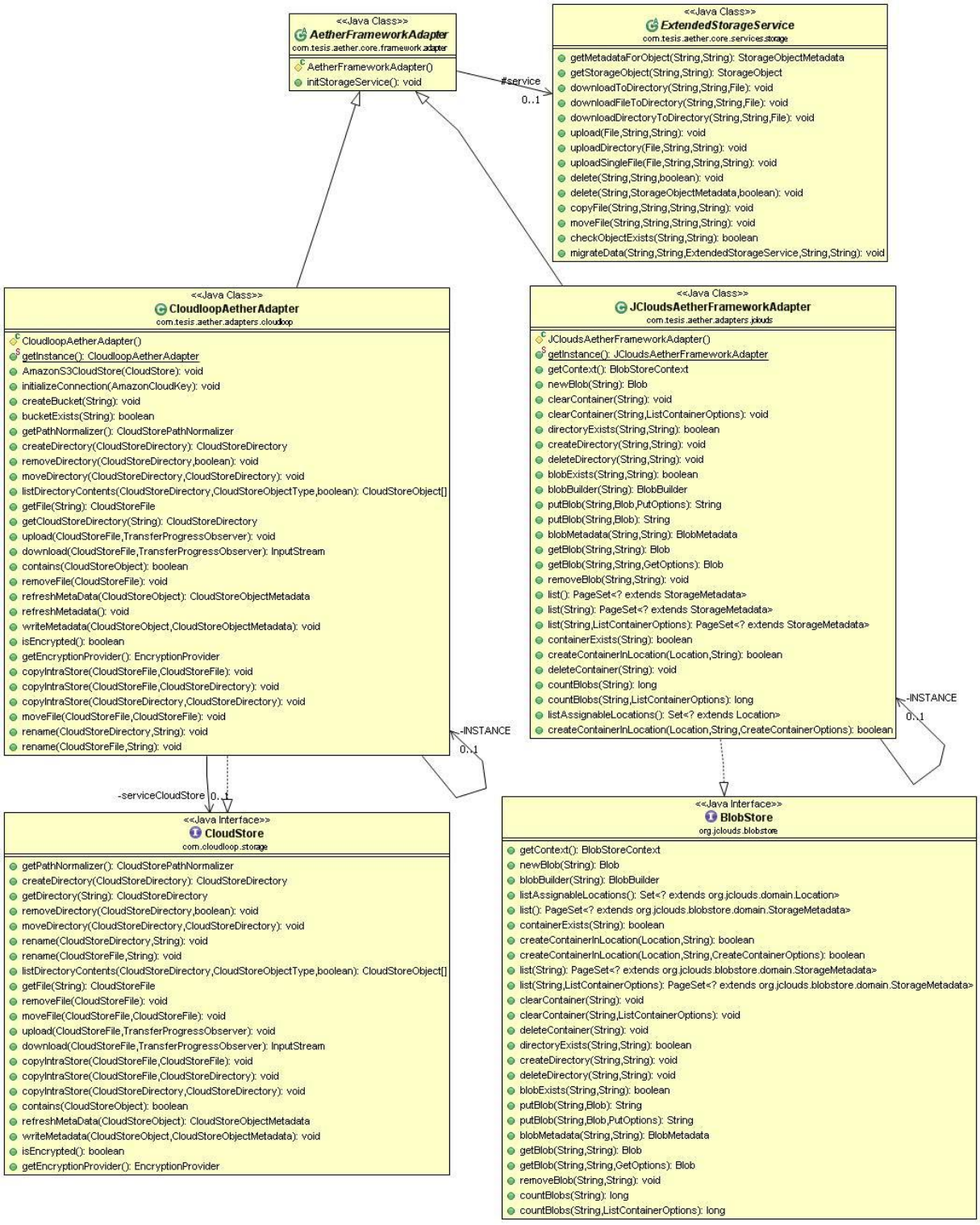

Figura 4.7 Abstracción de adapters para frameworks de terceros.

En la imagen anterior se puede apreciar la clase abstracta perteneciente a Aether-core denominada "AetherFrameworkAdapter" y las implementaciones particulares de los adapters para los frameworks

Cloudloop (CloudloopAetherFrameworkAdapter) y JClouds (JCloudsAetherFrameworkAdapter) los cuales extienden la funcionalidad de la primera y permiten llevar a cabo las traducciones mencionadas. Con respecto a los roles que cumple cada clase de la figura en relación al patrón "Adapter" podemos ver que AetherFrameworkAdapter corresponde al target u objetivo, CloudStore y BlobStore corresponden al adaptee o interface existente que se necesita adaptar y por último CloudLoopAetherAdapter y JCloudsAetherFrameworkAdapter corresponden a las clases adapters en sí.

Para comprender mejor la funcionalidad de un adapter y su interacción con Aether se presenta a continuación en la figura 4.8 un diagrama de secuencia en dónde se puede ver la inicialización del adapter y una solicitud de almacenamiento de un archivo por parte de la aplicación del usuario. Continuando con el ejemplo anterior se presenta el caso para la aplicación de usuario desarrollada utilizando JetS3t y Aether configurado para utilizar el almacenamiento de Google.

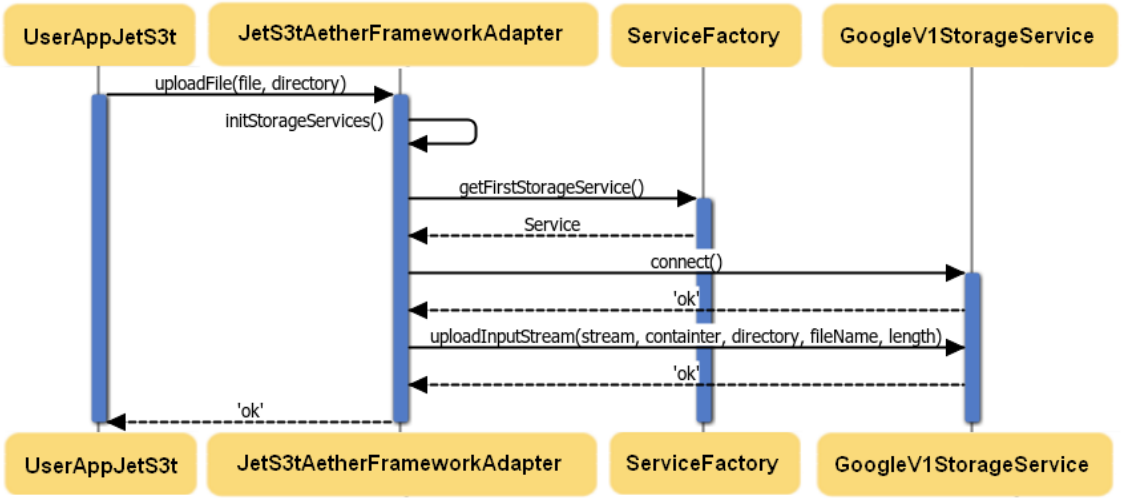

Figura 4.8 Secuencia de inicialización de un adapter y solicitud de almacenamiento.

En la figura anterior puede apreciarse la solicitud de almacenamiento de un archivo por parte de la aplicación del usuario (denominada "UserAppJetS3t") al adapter de JetS3t (JetS3tAetherFrameworkAdapter) por medio del método uploadFile. El adapter realiza el proceso de inicialización haciendo uso de su método initStorageServices desde el cual se solicita a ServiceFactory el servicio que tenga configurado por medio del método getFirstStorageService. Una vez que el adapter obtuvo el servicio, procede a realizar la conexión invocando al método connect de dicho servicio, en este caso correspondiente a GoogleV1StorageService. Cuando finaliza el proceso de conexión, el adapter realiza los llamados correspondientes a la solicitud de la aplicación del cliente. En este caso traduce la llamada a uploadFile de JetS3t por su equivalente en Aether uploadInputStream pasándole sus respectivos

parámetros. GoogleV1StorageService es en este punto el responsable de almacenar los datos según corresponda.

4.3 Módulo para el reemplazo dinámico de llamadas (Aether-loader)

Como se pudo ver en la figura 4.8 correspondiente a la sección 4.2, la aplicación del usuario es la que realiza la llamada al método del adapter. Para que estas llamadas se hagan de manera transparente para la aplicación del usuario, como fue mencionado en el capítulo de enfoque, se creó un módulo para lograr detectarlas y redireccionarlas al adapter correspondiente.

Luego de analizar diferentes alternativas para atacar el problema de interceptar las llamadas y redirigirlas al adapter correspondiente determinamos que la mejor solución es detectar la carga de clases mediante un classloader (Oracle, 2013) personalizado, modificar en tiempo de ejecución la clase del framework que utiliza la aplicación del cliente insertando las llamadas al adapter correspondiente y retornarla a la aplicación del usuario para que ésta no note la alteración y trabaje de manera normal.

Para lograr esto se recurrió a la utilización de una herramienta llamada Javassist (Javassist, 2013), la cual provee, entre otras utilidades de importancia, la posibilidad de modificar en tiempo de ejecución el código fuente de las clases ya "compiladas". El poder de esta herramienta se combinó con el uso de reflexión que provee java y la utilización de un classloader personalizado para Aether, el cual por medio de la lectura de datos desde un archivo de configuración xml reconoce las clases que debe modificar y compilar nuevamente. Siguiendo con el ejemplo de la sección anterior, éste módulo insertará en las clases del framework JetS3t que sean necesarias, las invocaciones al adapter JetS3tAetherFrameworkAdapter para que éste se encargue de las traducciones correspondientes a Aether y retorne el resultado esperado por la aplicación.

Debido a que la funcionalidad de este módulo es sencilla y concreta no fue necesario utilizar ningún patrón de diseño específico. A continuación en la figura 4.9 se presenta el diagrama de clases correspondiente que hace posible estas detecciones y modificaciones de código.

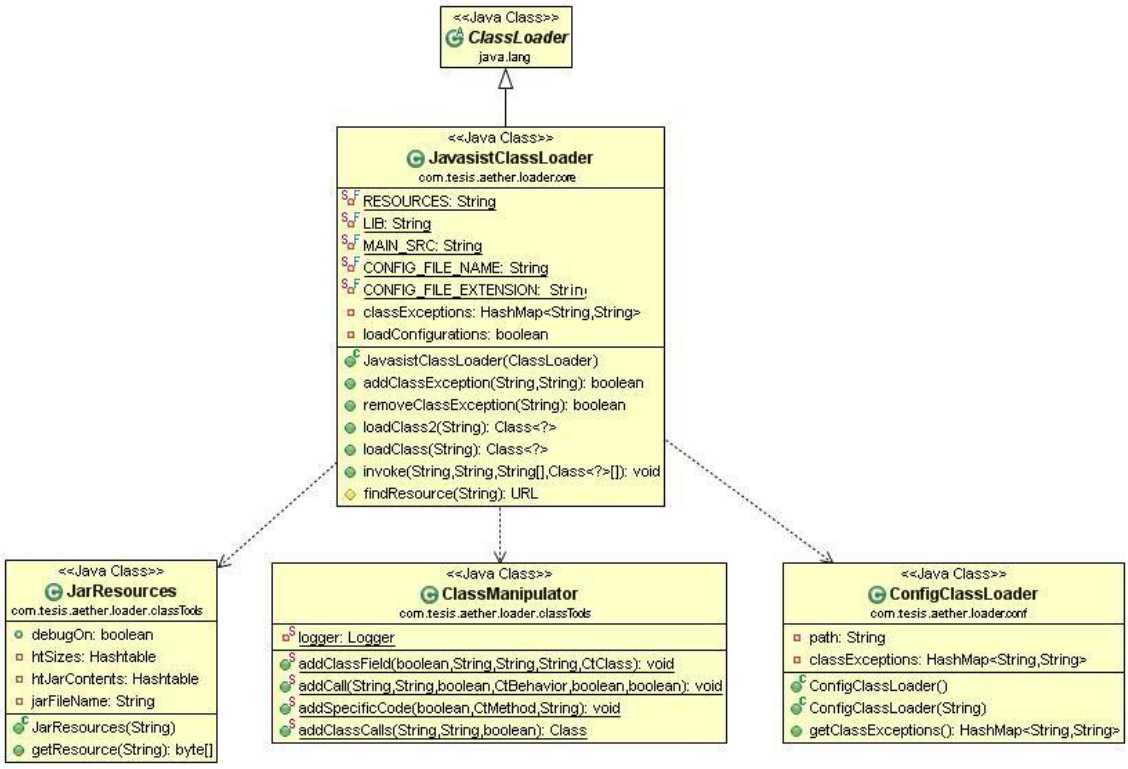

Figura 4.9 Diagrama de clases de Aether-loader.

Como puede verse en el diagrama anterior, Aether-loader consta de cuatro clases. La principal es "JavasistClassLoader" la cual provee las funcionalidades de carga de clases comunes de un classloader pero posee además una funcionalidad para detectar clases particulares y modificarlas antes de retornarlas. ConfigClassLoader se encarga de configurar el cargador de arranque con los datos especificados en un archivo de configuración XML. En este archivo se encuentran detallados los mapeos que se deben realizar entre las clases del framework de tercero y las clases del adapter. JarResources permite buscar y cargar las clases que se soliciten y se encuentren dentro de archivos jar. Por último, ClassManipulator es la encargada de modificar la clase que se indique agregando las llamadas al adapter correspondiente.

El archivo de configuración correspondiente a este módulo es muy simple y se debe encontrar presente en cada adapter desarrollado debido a que las clases que se deben detectar y modificar difieren de un framework a otro. A continuación en la figura 4.10 se presenta la estructura del archivo XML de configuración correspondiente al framework JetS3t mencionado para el caso de ejemplo.

```
<ClassLoaderConfig>
    <classException>
        <srcClass>org.jets3t.service.impl.rest.httpclient.RestS3Service</srcClass>
        <dstClass>com.tesis.aether.adapters.jets3t.JetS3tAetherFrameworkAdapter</dstClass>
    </classException>
    <classException>
        <srcClass>org.jets3t.service.impl.rest.httpclient.RestStorageService</srcClass>
        <dstClass>com.tesis.aether.adapters.jets3t.JetS3tAetherFrameworkAdapter</dstClass>
    </classException>
    <classException>
        <srcClass>org.jets3t.service.S3Service</srcClass>
        <dstClass>com.tesis.aether.adapters.jets3t.JetS3tAetherFrameworkAdapter</dstClass>
    </classException>
</ClassLoaderConfig>
```

Figura 4.10 Estructura del archivo XML de configuración de un adapter.

En la figura anterior pueden apreciarse cuatro elementos diferentes. ClassLoaderConfig determina la sección de configuración. ClassException contiene la información correspondiente a un mapeo de clases en el cual srcClass indica la clase original que deberá ser modificada al momento de cargarla mientras que dstClass hace referencia a la clase correspondiente al adapter que posee las implementaciones de los métodos de la original. Vale destacar que el archivo de configuración de un adapter puede contener más de un mapeo de clases como se puede apreciar en el ejemplo.

Cómo se indicó anteriormente, las clases especificadas en el archivo de configuración del classloader sufren modificaciones de su código en tiempo de ejecució. Estas modificaciones constan de agregar al comienzo de cada método de la clase a modificar, una llamada a uno con igual signatura presente en el adapter correspondiente y pasarle los parámetros recibidos. De esta forma, al momento de la nueva compilación en tiempo de ejecución, se deja sin efecto el resto de código presente en el método modificado. Las llamadas agregadas en este paso tienen la forma:

```
"return " + clase adapter + ".getInstance()." + nombre del método +
"(parametros);"
```

El método “getInstance()” del cual depende la inyección de código debe haber sido implementado por cada adapter concreto que se vaya a utilizar.

Llevaremos todo esto a un ejemplo sencillo en el cuál se describe la clase original, la clase correspondiente al adapter y el resultado final de la clase original modificada por el classloader. Supongamos la existencia de la clase original “Clase1” con un método “metodo1” el cuál recibe como parámetro un valor entero “param1” y retorna un valor de tipo String como se muestra a continuación:

```
public class Clase1 {
	public String metodo1(int param1) {
		return "Hola " + param1;
	}
}
```

El adapter correspondiente deberá poseer un método con la misma signatura que el declarado en la Clase1, por lo tanto el adapter quedaría como se muestra a continuación:

```
public class ClaseAdapter extends AetherFrameworkAdapter {
	private static ClaseAdapter INSTANCE = null;
	protected ClaseAdapter() {
		super();
	}
	public static ClaseAdapter getInstance() {
		if (INSTANCE == null) {
			INSTANCE = new ClaseAdapter();
		}
		return INSTANCE;
	}
	public String metodo1(int parametro) {
		return "Llamada modificada: Hola " + parametro;
	}
}
```

En el archivo xml de configuración para Aether-loader se debería indicar el mapeo de clases con la estructura y datos siguientes:

```
<ClassLoaderConfig>
      <classException>
            <srcClass>Clase1</srcClass>
            <dstClass>ClaseAdapter</dstClass>
      </classException>
</ClassLoaderConfig>
```

Al correr la aplicación con estos elementos, el classloader irá cargando las clases correspondientes a medida que se soliciten verificando que sean diferentes a la especificada en el archivo de configuración ("Clase1"). En caso de detectar que la clase que se solicitó coincide con la especificada, procederá a cargarla y modificarla resultando ésta en el código que se muestra a continuación:

```
public class Clase1 {
      public String metodo1(int param1) {
            return ClaseAdapter.getInstance().metodo1(param1);
      }
}
```

Gracias a este procedimiento de detección y modificación de código en tiempo de ejecución se libra al usuario en gran medida de tener que realizar cambios en la aplicación ya desarrollada. Las aplicaciones ya desarrolladas que suelen necesitar la intervención del usuario para adaptar su código a Aether son las que hagan uso de classloaders personalizados. En estos casos se debe modificar el código del classloader de la aplicación para delegar la carga de clases al cargador de Aether.

Es muy importante destacar que para activar el uso del classloader de Aether se debe utilizar un switch adicional para la máquina virtual de Java. Esto permite definirlo como classloader del sistema y de esta forma poder interceptar y redireccionar las llamadas hacia los adapters. El parámetro que debe especificarse para ejecutar la aplicación es el siguiente:

-Djava.system.class.loader=com.tesis.aether.loader.core.JavasistClassLoader

Como es de suponer, el cargador de adapters se encuentra entre la aplicación del usuario y el framework que ésta utiliza. Una vez iniciada la aplicación utilizando este módulo, Aether comienza a capturar las invocaciones de la aplicación del usuario a la herramienta objetivo (jClouds por ejemplo). Cuando se detecta un método de interés se reemplaza la llamada original por una llamada a su método homónimo en el adapter indicado en el archivo de configuración. Cabe destacar también que el flujo original de la aplicación se mantiene intacto con respecto a la nueva implementación, tal es el caso que desde el punto de vista del desarrollador el método que se ejecuta sigue siendo el original. Al trabajar de este modo, el usuario del framework sólo debe tener conocimiento del módulo cargador y cómo configurarlo, lo que se traduce en simpleza a la hora de utilizar la plataforma.

CAPÍTULO 5

5. RESULTADOS EXPERIMENTALES

Este capítulo analiza el desempeño de Aether frente a los frameworks más utilizados en la actualidad teniendo en cuenta características tales como consumo de recursos, facilidades de migración y cantidad de código requerido para realizar las tareas típicas de almacenamiento. Este análisis se realizó con el objetivo de validar el cumplimiento de las metas principales planteadas para el framework en el capítulo 3 de este trabajo. A lo largo del capítulo se presentarán las aplicaciones utilizadas para realizar los diferentes experimentos, su preparación y los resultados obtenidos. Paso a paso se analizarán las métricas que muestran los resultados de las diferentes implementaciones y el comportamiento de cada aplicación en tiempo de ejecución.

Para poder llevar a cabo el desarrollo de los experimentos y analizar el comportamiento de cada herramienta se determinó un conjunto de requerimientos mínimos que debe cumplir cada una de ellas. Estos requerimientos se detallan a continuación:

- Subir archivos.
- Descargar archivos.
- Buscar / listar archivos remotamente.
- Migrar de protocolo (para esto deberá poseer el soporte correspondiente).
- Poseer soporte para el lenguaje Java.

Una vez determinados los requerimientos mínimos con los que debe contar cada herramienta y basándonos en los resultados obtenidos para cada una de ellas en el capítulo 2 se seleccionaron las siguientes:

- libcloud
- jClouds
- Dasein
- CloudLoop
- JetS3t

5.1. Secuencia de experimentos y métricas

Para poder evaluar el comportamiento de Aether frente a las herramientas competidoras se llevaron a cabo sobre éstas una serie de experimentos. Estos experimentos se realizaron mediante el uso de tres aplicaciones con finalidades diferentes. Las aplicaciones debían proporcionar diferentes servicios, estar programadas en java, realizar diversas operaciones sobre archivos (por ejemplo guardar, borrar, leer), ser de diferentes tamaños (se buscó una aplicación chica –entre 3000 y 4000 líneas-, una aplicación mediana –entre 9000 y 10000 líneas- y una aplicación grande –de más de 50000 líneas-) y ser de dominio público. Siguiendo estos criterios la primera aplicación seleccionada fue el editor de texto ***Neoeedit*** (neoeedit, 2013). ***Neoeedit*** es un editor de texto sencillo el cual permite reconocer y resaltar sintaxis y trabaja solo con almacenamiento de archivos en un repositorio local, por lo tanto no presenta soporte alguno para almacenamiento en cloud. La segunda aplicación es el sincronizador de archivos ***JFileSync*** (JFileSync, 2013), el cual permite mantener sincronizados pares de directorios en el mismo o diferentes medios de almacenamiento. Cabe destacar que tampoco posee soporte para almacenamiento en cloud. La última aplicación es el administrador de archivos de doble ventana ***MuCommander*** (MuCommander, 2013). Esta aplicación permite la administración de archivos y directorios sobre diferentes protocolos y sistemas. Esta aplicación sí posee soporte para almacenamiento en *Cloud* en Amazon S3 utilizando una versión antigua de JetS3t.

Ya definidas las aplicaciones y las herramientas a utilizar para llevar a cabo los experimentos se seleccionó un desarrollador experto en el dominio, el cual utilizó tanto para el desarrollo de las soluciones como para las pruebas en tiempo de ejecución una máquina con las siguientes características:

- Procesador: Intel Core i5 750
- Memoria RAM: 8GB DDR3
- Windows 7 64 bits
- JAVA: JDK6

Para poder evaluar las características de cada framework y compararlas para determinar las virtudes y deficiencias de cada uno se definió una serie de métricas, las cuales son detalladas a continuación:

- **Cantidad de código accedido/consultado.** Esta métrica cuenta la cantidad de código que debió consultar el desarrollador para poder llevar a cabo la implementación correspondiente. Estará indicado por líneas de código, contabilizando una por cada línea presente en el método de la clase

que se acceda. Esta métrica se capturará utilizando la herramienta Mylyn (The Eclipse Foundation, 2013) durante el desarrollo. Mylyn es un framework para la gestión de tareas y ciclos de vida de aplicaciones. Además permite obtener una variedad de métricas para cada tarea que se realice, por ejemplo la cantidad de código accedido para completarla.

- **Líneas de código adicionales / alteradas.** Las líneas de código adicionales permitirán conocer las modificaciones necesarias para implementar cada solución. Se contabilizarán todas las líneas modificadas, agregadas y eliminadas de los archivos fuente (Java). Esta métrica será capturada utilizando la herramienta Araxis Merge (Araxis, 2013). Araxis Merge es una aplicación que permite comparar y fusionar documentos de manera muy sencilla. Además provee funcionalidades para generar reportes informando los resultados de las comparaciones realizadas.

- **Líneas de configuración adicionales / alteradas.** Al igual que la métrica descrita anteriormente, ésta indicará la cantidad de líneas de configuración que deberán ser modificadas para permitir el correcto funcionamiento de la aplicación para una implementación en particular. Se contabilizarán todas las líneas modificadas, agregadas y eliminadas de los archivos de configuración (properties y xml). Esta métrica será capturada utilizando Araxis Merge.

- **Uso de memoria.** Esta métrica indicará la cantidad máxima de memoria en MB utilizada durante cada ejecución en particular. Este valor será capturado utilizando la herramienta YourKit Java Profiler (YourKit, 2013).

- **Uso de CPU.** El uso de CPU permitirá conocer la cantidad de segundos de procesador al 100% necesario para correr la aplicación el tiempo que requiera cada ejecución. Esta métrica será también capturada con YourKit Java Profiler.

Para tomar las dos últimas métricas fue necesario ejecutar cada aplicación una cierta cantidad de veces. De esta forma se pudieron tomar los valores de cada una de ellas y calcularon los promedios correspondientes. Este valor se estableció en 10 (diez) ejecuciones.

A su vez, para poder tomar las métricas correspondientes a cada framework, se diseñó cada experimento en tres etapas las cuales permiten determinar las diferencias existentes en aspectos tales como facilidad de migración entre protocolos, facilidad de migración entre frameworks y el comportamiento de cada

herramienta en tiempo de ejecución. Las tres etapas de los experimentos y el objetivo de cada una de ellas se detallan a continuación:

***Etapa 1: Implementación de los casos de estudio con cada framework a analizar*.** En esta etapa se agregará soporte a cada una de las aplicaciones base para el almacenamiento en *Cloud* haciendo uso de las facilidades que presenta cada framework. Las métricas y resultados de esta etapa permitirán evaluar que tan complejo resulta implementar cada uno de los casos de estudio con cada herramienta y el comportamiento que presenta cada una en tiempo de ejecución con respecto a la utilización de recursos.
Las métricas utilizadas en esta etapa para la codificación de la solución son las siguientes:

- **Cantidad de código accedido/consultado.**
- **Líneas de código adicionales / alteradas**
- **Líneas de configuración adicionales / alteradas**

Una vez finalizado el desarrollo y captura de métricas anterior se determinará la eficiencia en términos de uso de recursos en tiempo de ejecución para cada framework. Para esto se definieron una serie de experimentos que serán llevados a cabo con cada aplicación (estos experimentos se describirán más adelante). Las métricas que se tomarán para analizar el comportamiento en tiempo de ejecución son las siguientes:

- **Uso de memoria.**
- **Uso de CPU.**

***Etapa 2: Evaluación de las facilidades de migración entre protocolos dentro de cada framework*.** En ésta segunda etapa se trabajara sobre las aplicaciones resultantes de la etapa anterior. Cada aplicación será migrada a otro protocolo soportado por el framework sobre el cual se implementó. Esto nos permitirá evaluar la magnitud de los cambios que son necesarios para realizar las migraciones, tanto en cantidad de líneas de código como en cantidad de líneas de configuración. Para poder determinar estos valores se tomarán las métricas nombradas a continuación:

- **Líneas de código adicionales / alteradas.**
- **Líneas de configuración adicionales / alteradas.**

***Etapa 3: Migración de cada implementación al resto de los frameworks competidores*.** En esta última etapa también se trabajara con las aplicaciones resultantes de la primera. El objetivo será realizar la migración de cada aplicación a cada uno de los frameworks competidores. Esto servirá para determinar

qué tan complejo resulta realizar la migración de una aplicación desarrollada utilizando un framework particular a otro diferente. Una vez finalizadas las diferentes implementaciones se tomará el siguiente conjunto de métricas:

- **Cantidad de código accedido/consultado.**
- **Líneas de código adicionales / alteradas.**
- **Líneas de configuración adicionales / alteradas.**

Al igual que en la etapa uno, una vez finalizado el desarrollo y captura de métricas anterior se procederá a determinar la eficiencia en términos de uso de recursos en tiempo de ejecución para cada framework. Para esto se utilizarán los mismos experimentos usados en la primer etapa. Las métricas que se tomarán son las siguientes:

- **Uso de memoria.**
- **Uso de CPU.**

Una observación que cabe destacar sobre la captura de métricas en tiempo de ejecución es que para evitar valores erróneos en los resultados se procedió a realizar los pasos anteriores diez veces sobre cada implementación. Con esto se logra reducir el error que puede existir debido a condiciones especiales como por ejemplo la ejecución del garbage collector. Esto se llevó a cabo en cada uno de los casos de estudio analizados, tanto para la etapa 1 como para la etapa 3.

5.2. Descripción de los casos de estudio

En las secciones siguientes se presentan uno a uno los tres casos de estudio llevados adelante junto a los resultados y análisis de cada una de las métricas mencionadas anteriormente.

5.2.1 Caso de estudio 1: neoeedit

Este caso de estudio se concentra en el análisis del editor de texto “neoeedit”. Esta aplicación trabaja únicamente con archivos en el almacenamiento local del usuario. El objetivo será agregar soporte para almacenamiento remoto con cada framework. Esto incluye soporte para la funcionalidad de abrir y grabar archivos, buscar archivos que contengan cierto bloque de texto y demás funcionalidades provistas por la aplicación original. La figura figura 5.1 muestra a grandes rasgos la arquitectura de la aplicación.

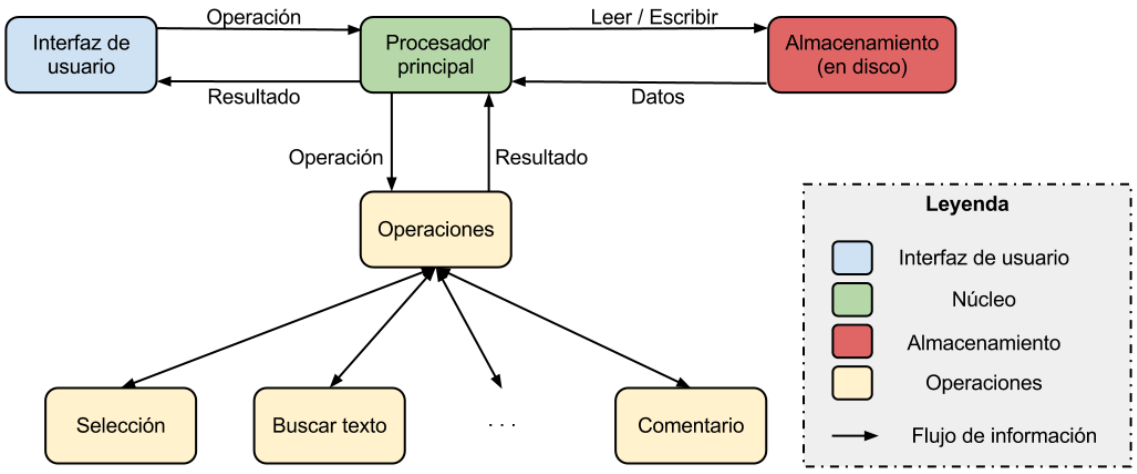

Figura 5.1 Diagrama abstracto del funcionamiento de neoeedit.

Como puede apreciarse el funcionamiento de la aplicación es muy simple. El usuario da órdenes a la aplicación por medio de la interfaz de usuario la cual inmediatamente envía la misma al procesador principal. Este procesador, dependiendo de la orden recibida, selecciona y ejecuta un conjunto de acciones (por ejemplo almacenar los cambios realizados sobre el archivo que está siendo editado). Las operaciones que necesitan acceder a los datos del archivo original lo hacen leyendo o escribiendo directamente en el disco local por medio de las facilidades que otorga java.

Antes de comenzar con el desarrollo de cada una de las tres etapas mencionadas anteriormente presentamos las métricas básicas (Líneas de código y Promedio de líneas de código por método) correspondientes a la aplicación original. Con estos datos tenemos una referencia del tamaño de la aplicación. Los valores de estas dos métricas son los siguientes:

Líneas de código totales: 3642

Promedio de líneas de código por método: 13,98

5.2.1.1. Etapa 1

En el análisis inicial de la herramienta se identificaron las clases que deben ser modificadas para cumplir con los objetivos planteados anteriormente para esta primera etapa. Lamentablemente, al tratarse de una aplicación de escritorio los desarrolladores nunca consideraron la posibilidad de trabajar con sistemas de

archivos remotos, esto significa que las operaciones sobre los archivos no se encontraban agrupadas en clases que manejaran la persistencia. Tomando en cuenta estas falencias, en cada implementación se reemplazaron las llamadas a disco originales por llamadas a una interface. Esta interface brinda el soporte necesario para que cada framework en particular realice las tareas de almacenamiento de manera más sencilla.

Una vez que se realizaron las implementaciones correspondientes utilizando cada uno de los frameworks seleccionados se tomaron las métricas correspondientes a esta etapa, las cuales se presentan en la figura 5.2.

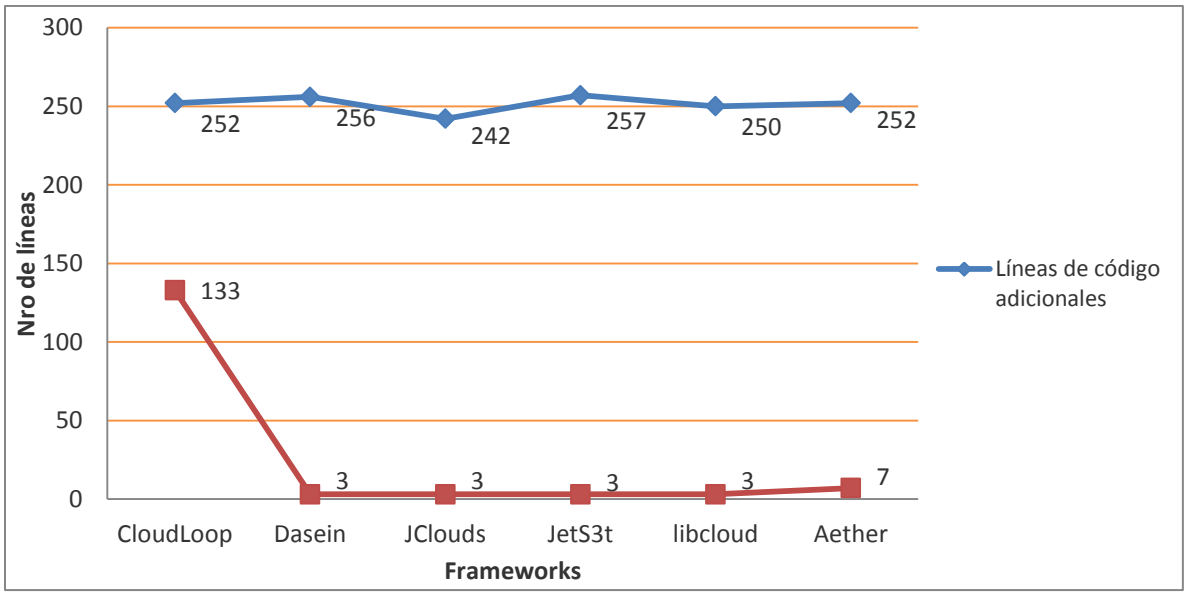

Figura 5.2 Líneas de código y configuración adicionales de cada implementación.

Como puede apreciarse en el gráfico todos los frameworks están equilibrados en cuanto a la cantidad de líneas de código requeridas por cada implementación ya que requirieron la incorporación de aproximadamente 250 líneas de código adicionales. Esta situación varía si consideramos las líneas de configuración adicionales requeridas por cada framework. En este sentido, la solución de CloudLoop requirió de 130 líneas más que sus competidores, mientras que el resto de los frameworks, incluido Aether, no requirieron de tanta carga de configuración manteniendo la métrica Líneas de configuración en un dígito.

Para poder llevar a cabo cada una de las implementaciones anteriores, el desarrollador debió consultar ciertos fragmentos de código pertenecientes a cada aplicación y a cada framework. Este esfuerzo se midió

en líneas de código consultado para cada desarrollo. Los datos correspondientes a esta métrica para el primer caso de estudio pueden apreciarse en la figura 5.3.

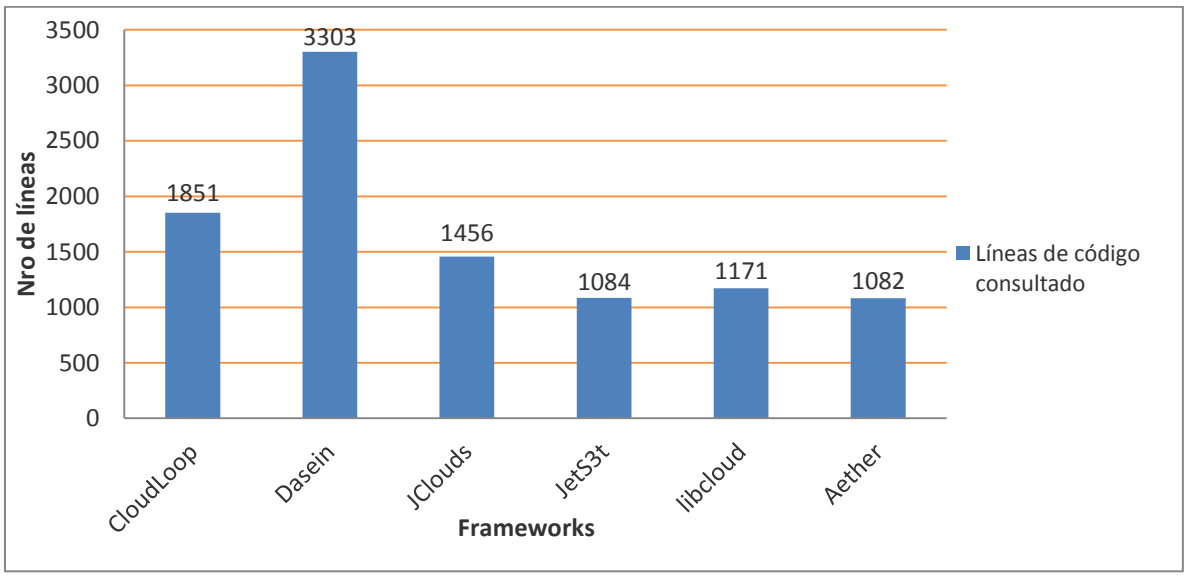

Figura 5.3 Líneas de código consultado para realizar cada implementación.

Podemos apreciar en el gráfico que Aether y JetS3t son los frameworks que más se destacan, seguidos muy de cerca por Libcloud. Por el contrario, el desarrollo con Dasein necesitó de la consulta de más del triple de código que las anteriores para poder realizar el mismo trabajo. Esto se debe en gran parte a la calidad de las interfaces de los frameworks ya que muchas veces resultan ser poco claras y con nombres de métodos y parámetros muy poco intuitivos.

Como es de suponerse, aquellas herramientas que necesiten menor cantidad de líneas de código consultado para su utilización benefician de gran forma al programador ya que permiten realizar un desarrollo más rápido y ágil que el resto. Estos valores resultan ser muy importantes a la hora de estimar las horas de desarrollo de un sistema ya que si por ejemplo se tiene poco conocimiento acerca de la herramienta a utilizar, el tiempo total se verá impactado directamente por la complejidad que posean las interfaces y la documentación correspondientes.

Una vez finalizada la primera parte de esta etapa se procedió a probar el comportamiento de cada implementación en tiempo de ejecución. Para esto se diseñó y ejecutó sobre cada desarrollo un caso de prueba, el cual para esta implementación en particular consta de cuatro pasos los cuales se detallan a continuación:

1. Abrir un archivo de texto remoto existente y vacío.
2. Agregar la palabra test.
3. Guardarlo remotamente.
4. Buscar remotamente todos los archivos que contengan la palabra test.

Los resultados de este experimento se muestran en la figura 5.4 en términos de Consumo de Memoria y en relación al uso de CPU en la figura 5.5.

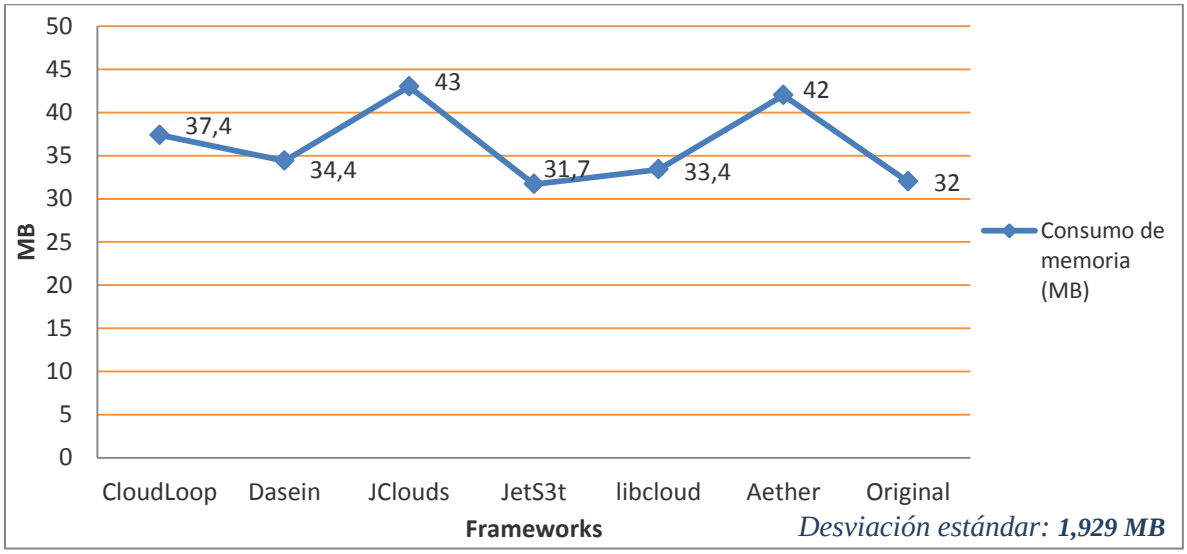

Figura 5.4 Consumo de memoria en tiempo de ejecución para cada framework.

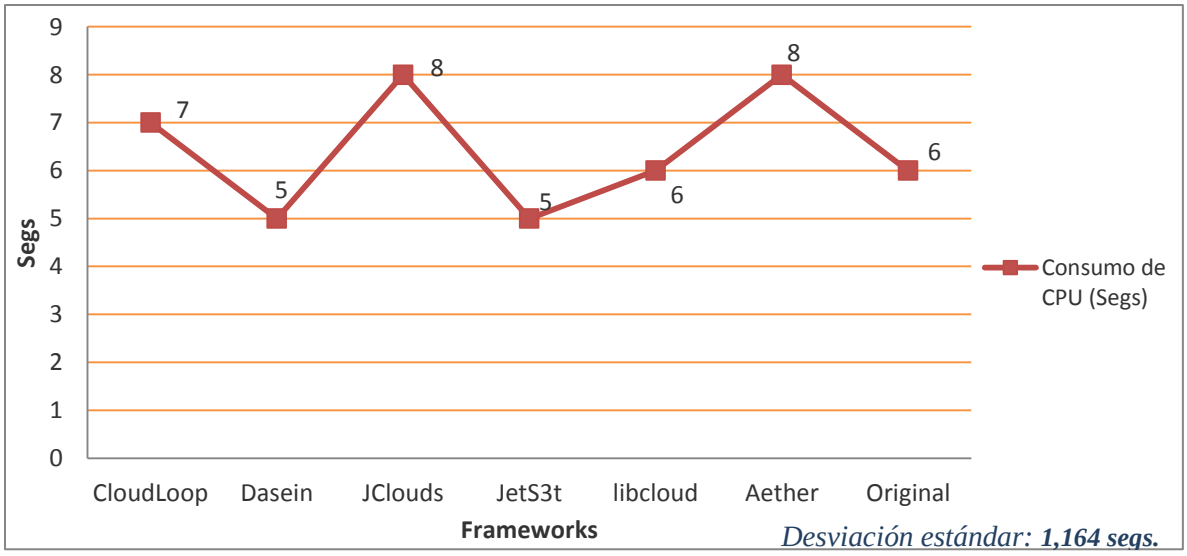

Figura 5.5 Consumo de CPU en tiempo de ejecución para cada framework.

Visualizando los resultados presentes en los dos gráficos se puede apreciar que las implementaciones que utilizan Aether y JClouds utilizaron mayor cantidad de recursos que la competencia. Estos dos frameworks consumieron unos 2 segundos más de CPU y 10MB más de memoria que la implementación original de neoeedit. Para el caso de Aether, este incremento en el uso de CPU y memoria se debe a que para permitir la modificabilidad del servicio de storage se realizan varias instrucciones intermedias que permiten la abstracción de cada operación. El resto de los frameworks resultan más conservadores en la utilización de recursos. Puede verse también que para este experimento en particular Dasein, libcloud y JetS3t son los más indicados en cuanto a eficiencia en tiempo de ejecución.

5.2.1.2. Etapa 2

Una vez finalizados los pasos correspondientes a la etapa anterior, se procedió a realizar las migraciones entre protocolos soportados por cada framework. Los datos obtenidos de estas implementaciones nos indicarán que tan complejo resulta llevar a cabo una migración de protocolos utilizando la misma herramienta. Luego de realizadas las migraciones correspondientes a esta etapa sobre los resultados de la anterior se obtuvo el conjunto de datos presentado en la figura 5.6 para la cantidad de líneas de código y configuración modificadas.

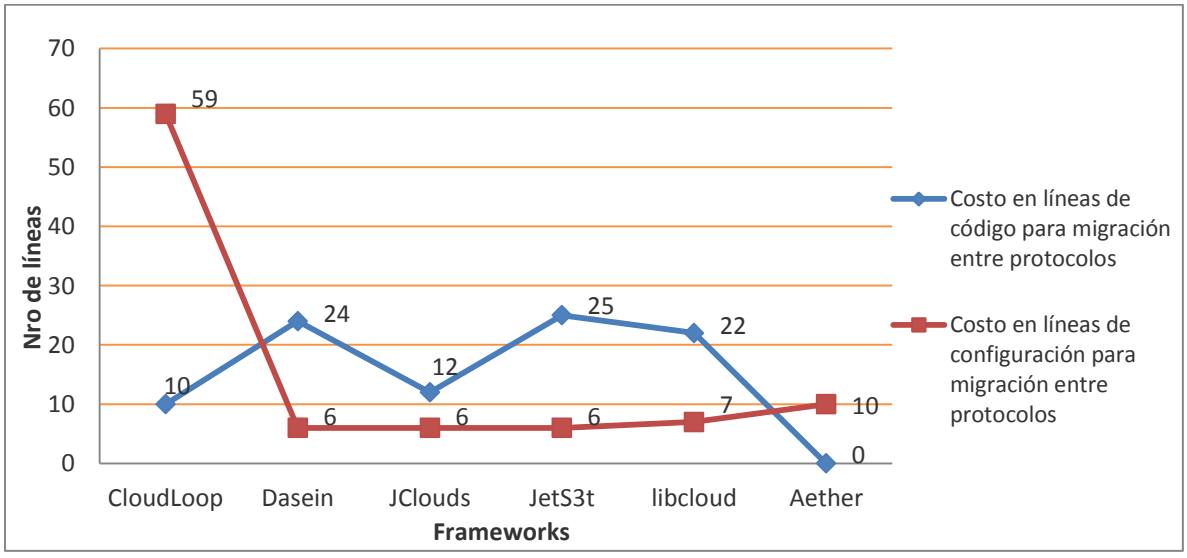

Figura 5.6 Líneas de código y configuración adicionales para migraciones entre protocolos.

En base a los resultados obtenidos podemos apreciar que Aether fue el único framework que no requirió cambios de código en la aplicación para realizar la migración. El resto de los frameworks requirieron

modificaciones que variaron entre las 10 y las 25 líneas de código, lo cual a pesar de ser neoeedit una aplicación pequeña resulta relativamente baja. La cantidad de líneas de código modificadas pueden ser despreciables en una aplicación pequeña como neoeedit, pero en una aplicación de mayor tamaño pueden resultar costosas. En las aplicaciones siguientes se podrá ver el impacto sobre esta variable ya que son considerablemente más grandes que neoeedit.
En cuanto a las líneas de configuración podemos observar que nuevamente fue CloudLoop el que requirió mayor cantidad de cambios. El resto de los frameworks se mantuvieron estables con una cantidad de líneas de configuración que no sobrepasaron las 10, una medida relativamente baja.

5.2.1.3. Etapa 3

Una vez desarrolladas las etapas anteriores queda por delante evaluar que tan "portable" resulta cada una de las herramientas, es decir, qué tan complejo resulta migrar una aplicación desarrollada utilizando una herramienta específica a otra diferente. Para esto nos basamos en las implementaciones resultantes de la etapa uno, adaptando cada solución obtenida a cada uno de los frameworks competidores. Finalizadas las implementaciones correspondientes se obtuvo un conjunto de resultados para las métricas propuestas los cuales se irán detallando a continuación.

Al realizar la migración de una implementación específica a otra utilizando una herramienta diferente es necesario realizar cambios tanto a nivel de código fuente como así también en la configuración de cada desarrollo. Estas modificaciones pueden variar desde unas pocas líneas a unos cientos, e incluso miles dependiendo de las características de cada implementación y framework utilizado. Como es de imaginar, estos cambios tienen como contra principal el tiempo que requiere el desarrollo para la adaptación a la nueva versión y la tediosa tarea de probar cada una de las modificaciones realizadas. La figura 5.7 presenta la cantidad de líneas de código que fueron necesarias para realizar cada una de las migraciones correspondientes. El gráfico indica con cada color el framework original sobre el cual se realizó la migración al indicado sobre el eje de "frameworks destino".

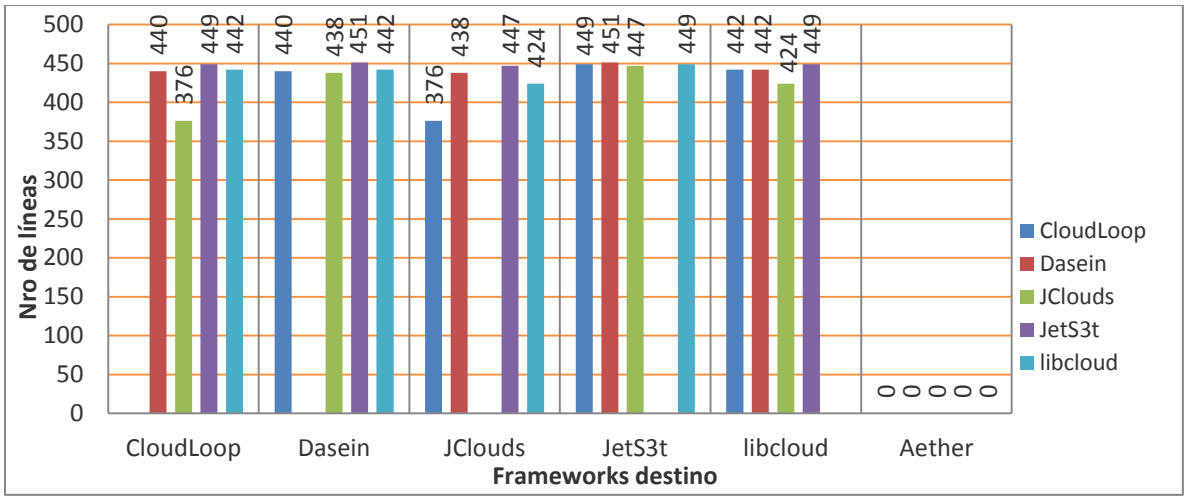

Figura 5.7 Líneas de código adicionales para migraciones entre frameworks.

En base a los resultados obtenidos se puede ver que la mayoría de los frameworks no contemplan este tipo de migración. Esto deriva en que el desarrollador tenga que implementar cada aplicación nuevamente con el framework deseado. Visualizando los valores podemos apreciar que no ocurre esto para el framework Aether, ya que para este caso en particular, la tarea de migración de cada uno de los frameworks restantes a éste resultó en un costo de cero líneas de código fuente modificado. Debido a esto Aether es muy superior a cualquiera de los otros frameworks en cuanto a las migraciones entre ellos. Este valor (cero líneas modificadas) se obtiene gracias a que Aether hace uso de su cargador de clases y adapter correspondiente para poder traducir cada instrucción de un framework particular en instrucciones equivalentes para él. Es útil destacar que al no requerir cambios en el código fuente, Aether hace una gran tarea facilitándole considerablemente la migración al programador.

Como se mencionó anteriormente una aplicación está compuesta, además de por el código fuente, por archivos de configuración. Estos archivos se utilizan para indicar a la aplicación ciertos parámetros variables sin necesidad de volver a regenerar la solución. Al migrar una aplicación es muy común que se modifiquen datos de estos archivos ya que se deben alterar ciertos comportamientos o valores específicos para el correcto funcionamiento. A continuación en la figura 5.8 se muestran los resultados correspondientes en cuanto a líneas de configuración modificadas por cada re-implementación.

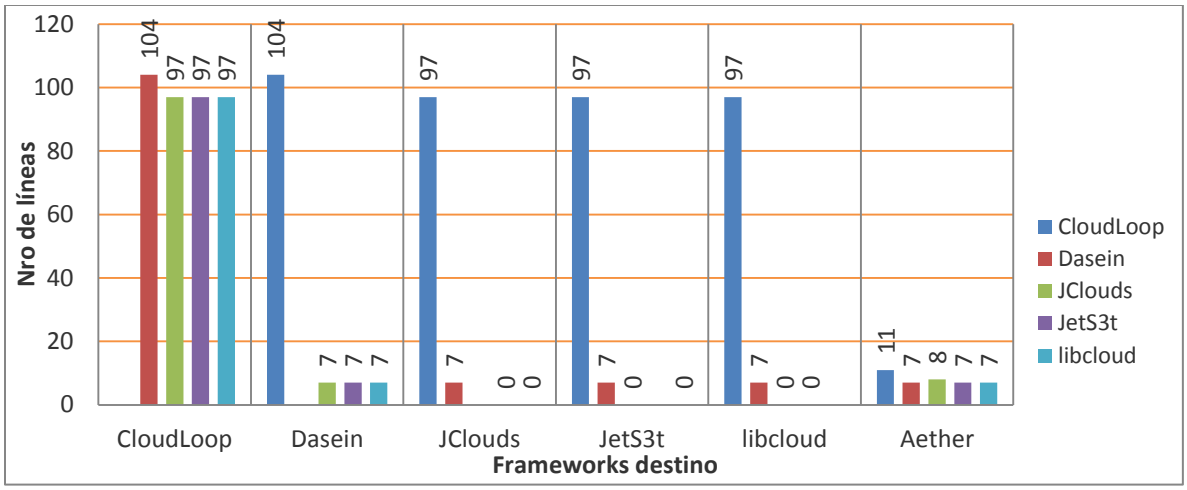

Figura 5.8 Líneas de configuración adicionales para migraciones entre frameworks.

Se puede apreciar que cada aplicación mantiene los niveles expuestos en las etapas previas del experimento. Como sucedió anteriormente, CloudLoop presenta la mayor cantidad de modificaciones sobre las líneas de configuración necesarias para su funcionamiento. El resto de las herramientas mantuvieron un nivel relativamente bajo y constante en cuanto a líneas de configuración modificadas. Luego de ver los resultados de este gráfico podemos notar que si sumamos los esfuerzos en líneas de código y configuración modificadas Aether saca una gran ventaja sobre el resto de los frameworks ya que sólo fue necesario para cada migración modificar unas pocas líneas de configuración.

Con respecto al último punto importante a tener en cuenta sobre la fase de desarrollo describiremos, al igual que se realizó en la primera fase, la complejidad de realizar cada re-implementación en términos de líneas de código consultado. Para esto se capturaron cada uno de los métodos a los que ingresó el programador para determinar su comportamiento. Los datos obtenidos para ésta métrica en particular una vez finalizadas las implementaciones se presentan a continuación en la figura 5.9.

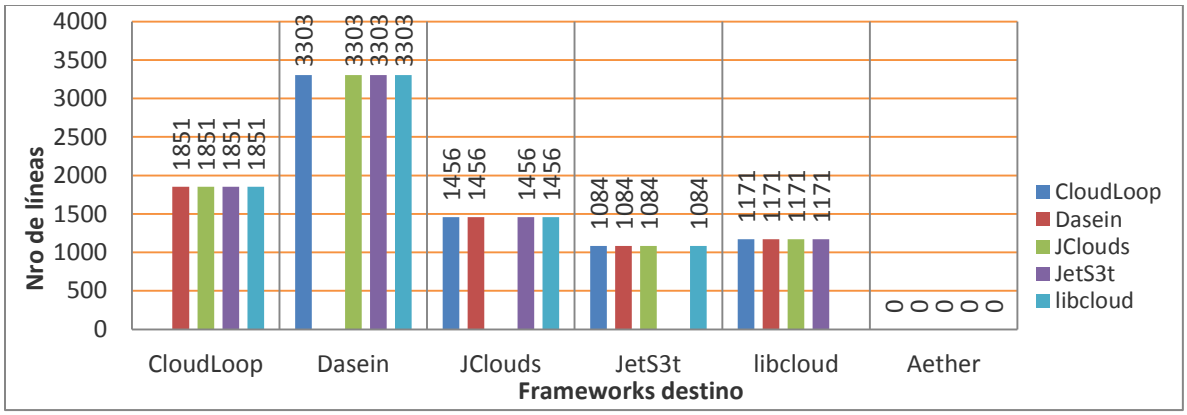

Figura 5.9 Líneas de código consultado para migraciones entre frameworks.

** Para Aether sólo se miraron líneas de configuración y no código fuente.*

En el gráfico anterior podemos ver claramente que Aether no requirió consultas de código para realizar la migración, mientras que el resto de los frameworks se correspondieron con los niveles idénticos a los de la etapa 1.

Habiendo determinado anteriormente el comportamiento de cada uno de las herramientas en tiempo de ejecución sería de gran importancia verificar como impacta sobre el uso de CPU y memoria cada migración realizada por Aether. Para lograr esto se volvió a ejecutar sobre cada una de las migraciones la secuencia de pasos descrita en la etapa 1. Los gráficos siguientes (figuras 5.10 y 5.11) presentan los resultados correspondientes para cada una de estas nuevas ejecuciones.

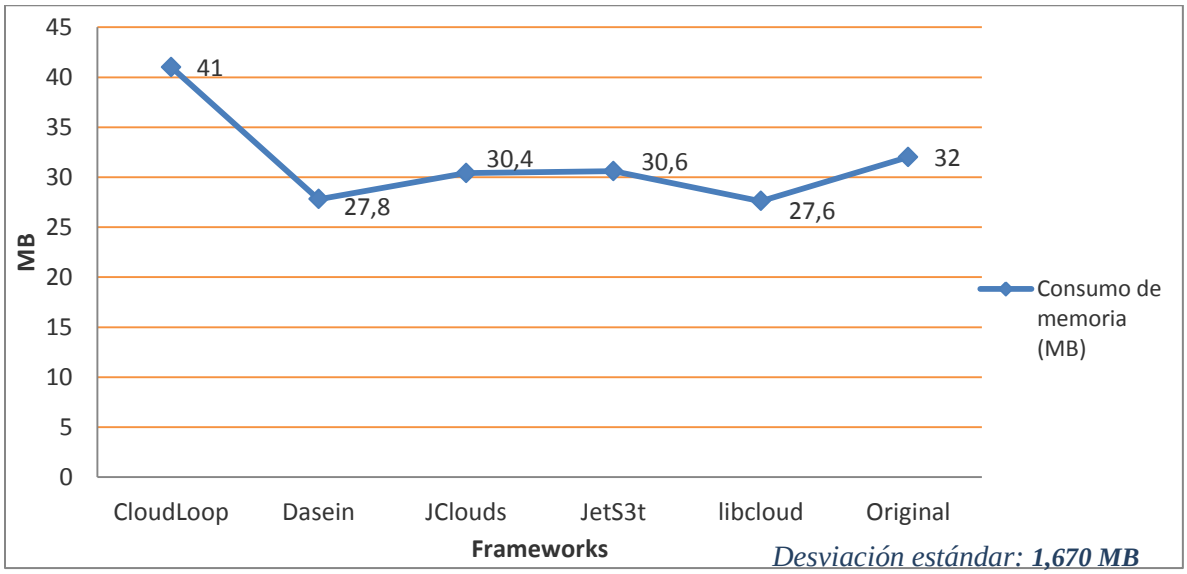

Figura 5.10 Consumo de memoria en tiempo de ejecución para migraciones a Aether.

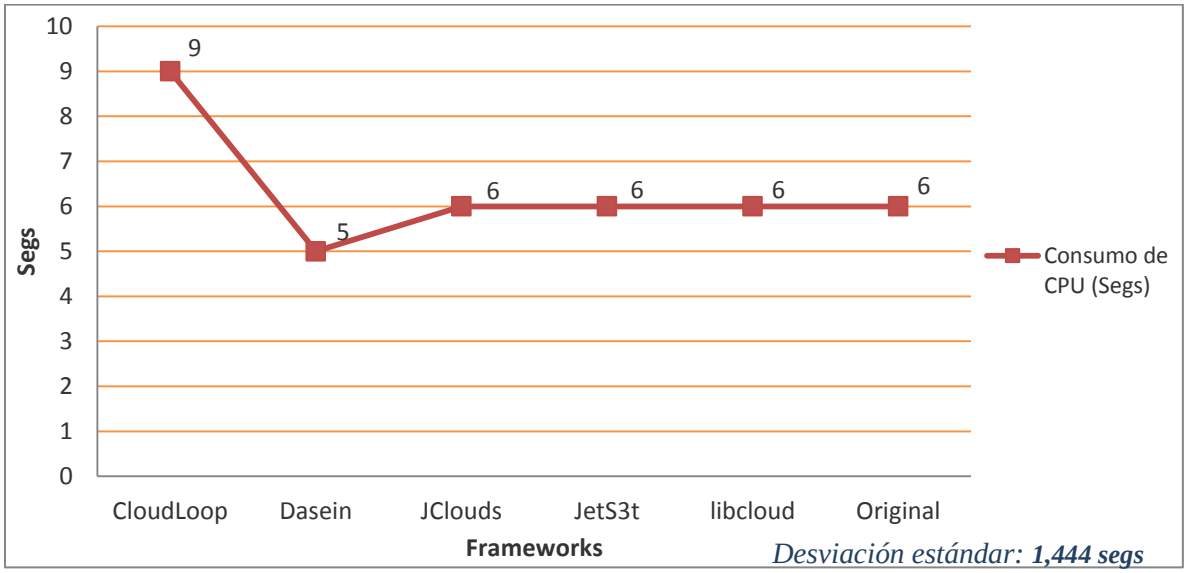

Figura 5.11 Consumo de CPU en tiempo de ejecución para migraciones a Aether.

Como puede apreciarse en los gráficos anteriores no se presentan variaciones significativas entre las migraciones realizadas por Aether y la aplicación original. Podemos ver que la única excepción corresponde al caso de CloudLoop en el cual se nota un pequeño incremento en el uso de ambos recursos (unos 9MB extra de memoria y 3 segundos más de CPU). Por medio de estos resultados podemos notar que el reemplazo dinámico de llamadas que debe hacer Aether para utilizar los Adapters, no produce un impacto negativo significativo en cuanto a performance.

5.2.2. Caso de estudio 2: JFileSync

El segundo caso de estudio se realizará sobre el sincronizador de archivos "JFileSync". La aplicación se utiliza para sincronizar pares de directorios presentes en el mismo o en diferentes dispositivos de almacenamiento. El objetivo será implementar el soporte para almacenamiento remoto utilizando cada uno de los frameworks a evaluar. Para este fin se utilizaran diversas capacidades de cada framework como las de obtener metadatos y subir o descargar archivos. A continuación la figura 5.12 muestra la arquitectura de la aplicación a grandes rasgos.

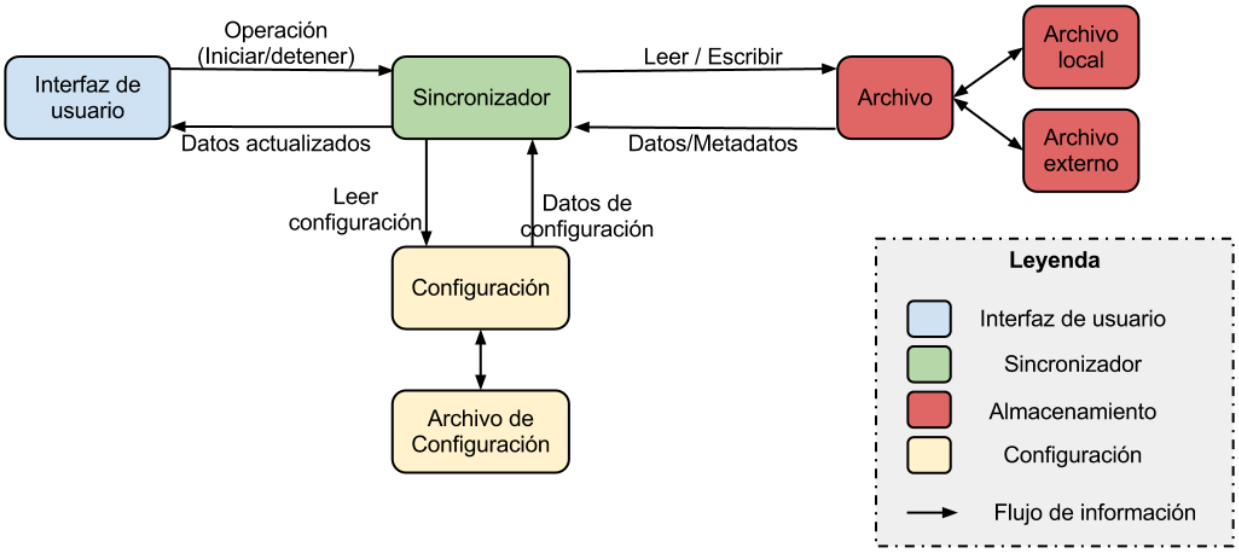

Figura 5.12 Diagrama abstracto del funcionamiento de JFileSync.

Como puede apreciarse, el funcionamiento de la aplicación es muy sencillo. Al iniciar la aplicación carga los datos de configuración (carpetas a sincronizar y otras propiedades) los cuales puede modificar el usuario en tiempo de ejecución y mantiene sincronizados los archivos/carpetas que se indiquen. Estos elementos se pueden encontrar tanto en un repositorio local (en el disco de la máquina) como en uno remoto.

Al igual que para el caso de estudio 1, antes de comenzar a desarrollar cada etapa procederemos a presentar las métricas básicas correspondientes a la aplicación JFileSync original como son la Cantidad de líneas de código y el Promedio de líneas de código por método. Nuevamente, con estos datos podremos determinar el tamaño de la aplicación; estos valores son:

Líneas de código totales: 9623
Promedio de líneas de código por método: 10,53

5.2.2.1. Etapa 1

Previamente a la implementación de esta primera etapa se procedió a realizar un análisis inicial de la herramienta identificando las clases que se debieron modificar para cumplir con los objetivos planteados anteriormente. Gracias a que JFileSync soporta varios protocolos (uno para archivos locales y otro

propietario para archivos remotos) la inclusión de uno nuevo basado en algún framework particular no resulta tan complicada. Las interfaces con las que se debe trabajar están bien definidas y los métodos son claros.

Una vez realizadas las implementaciones correspondientes utilizando cada uno de los frameworks seleccionados se tomaron las métricas correspondientes a esta etapa. En la figura 5.13 se presentan las métricas resultado de ésta primera etapa para el caso de estudio actual.

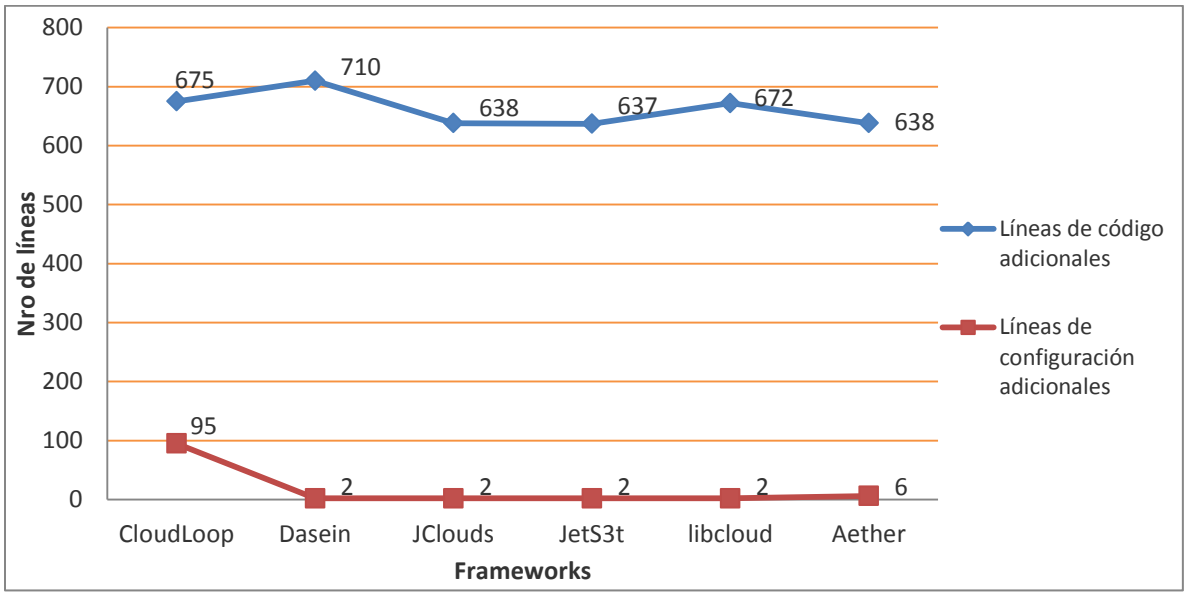

Figura 5.13 Líneas de código y configuración adicionales de cada implementación.

Como puede apreciarse en el gráfico anterior, los frameworks están relativamente equilibrados en cuanto a Líneas de código adicionales destacándose Dasein como el framework que más modificaciones necesita para realizar el mismo trabajo que JClouds, Jets3t y Aether logran con 70 líneas menos. Esto indica que la implementación de la solución con cada uno de estos últimos frameworks resulta menos costosa (en líneas de código) que el resto de las implementaciones. Si nuevamente consideramos las líneas de configuración junto con las líneas de código modificadas vemos que CloudLoop, al igual que en el caso de estudio 1, es el que más modificaciones necesitó para realizar la misma tarea que los frameworks restantes (unas 62 líneas más que Daseín siendo este el segundo con más modificaciones necesarias). El resto de los frameworks se mantuvieron estables con una cantidad de entre 640 y 674 líneas modificadas en total.

Para poder llevar a cabo cada una de las implementaciones anteriores, el desarrollador debió consultar ciertos fragmentos de código pertenecientes a JFileSync y a cada framework utilizado. Este esfuerzo se midió en líneas de código consultado para cada desarrollo. Los datos correspondientes a esta métrica para el segundo caso de estudio pueden apreciarse a continuación en la figura 5.14.

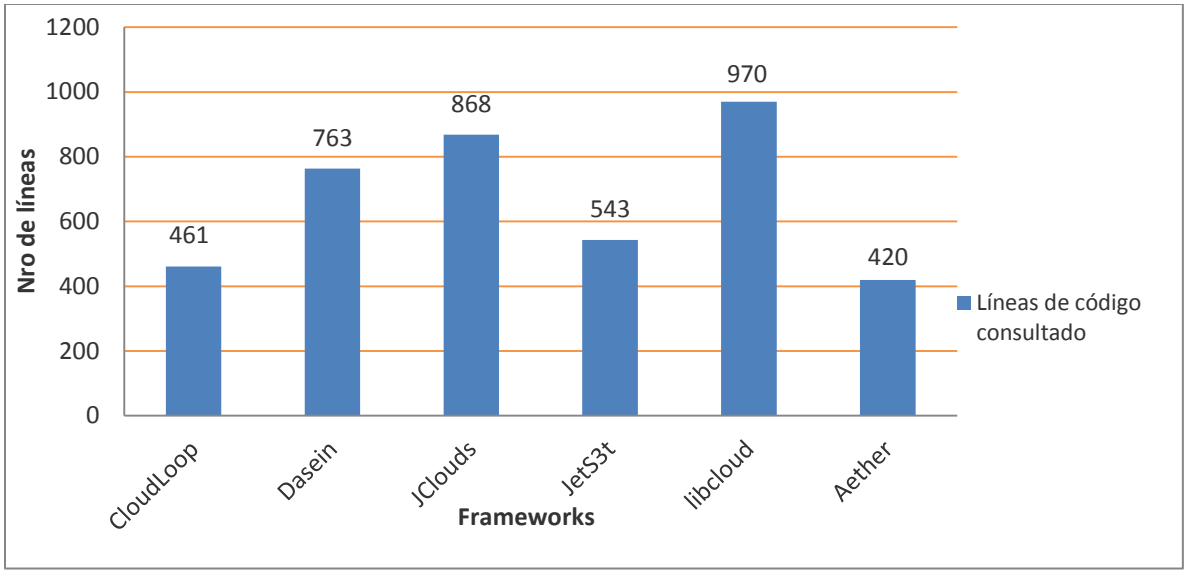

Figura 5.14 Líneas de código consultado para realizar cada implementación.

Como puede verse en el gráfico, las herramientas más eficientes resultaron ser Aether y CloudLoop, seguidos muy de cerca por JetS3t mientras que Dasein, JClouds y libcloud requirieron la consulta de entre 300 y 500 líneas de código adicionales llegando a un máximo de 970. Como se indicó en el caso de estudio 1, la diferencia de líneas de código consultado se debe en gran parte a la calidad de las interfaces ya que algunas (sobre las que más código se debió consultar) no poseen nombres de métodos ni parámetros descriptivos con lo cual se debe visualizar el código fuente para poder determinar el comportamiento.

Para pasar ahora a testear otra de las características importantes de los frameworks, el desempeño en tiempo de ejecución, se definió una secuencia de pasos a realizar sobre cada implementación. A continuación se detalla esta secuencia de pasos para jFileSync:

1. Crear una comparación entre un directorio local con dos archivos de texto (1.txt y 2.txt) y un directorio remoto con dos archivos de texto (2.txt y 3.txt)

2. Sincronizar los directorios hacia la izquierda. Esto significa que el archivo 2.txt se mantiene localmente y se descarga el archivo 3.txt. El archivo restante, 1.txt, es eliminado localmente para replicar la estructura remota.

Los resultados de este experimento se muestran en la figura 5.15 en términos de Consumo de Memoria y en relación al uso de CPU en la figura 5.16 ambas contrastadas contra los datos obtenidos para la aplicación original.

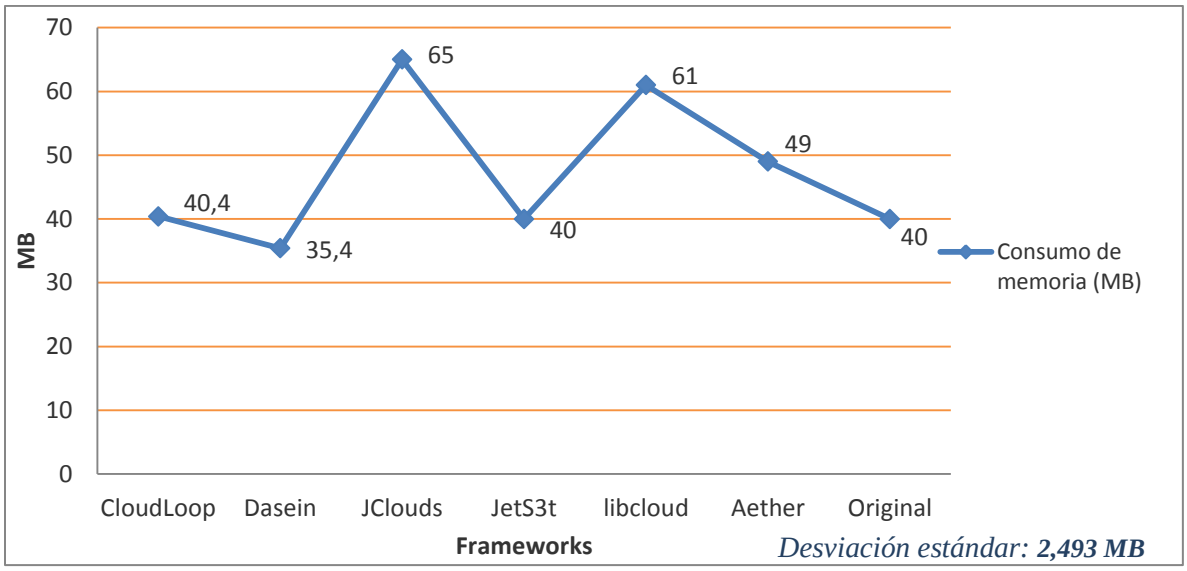

Figura 5.15 Consumo de memoria en tiempo de ejecución para cada framework.

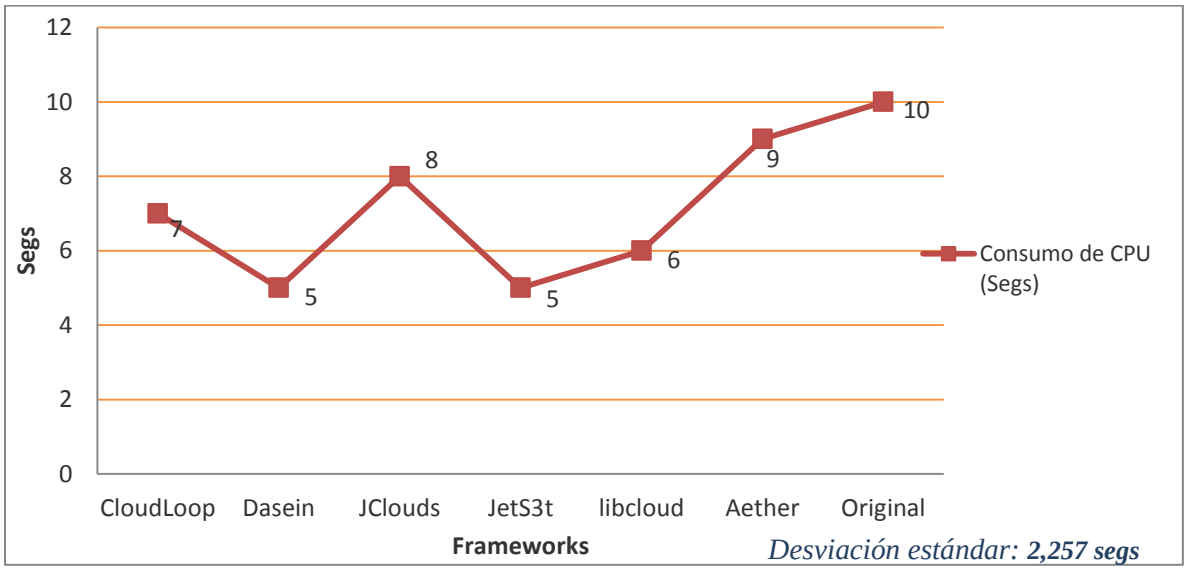

Figura 5.16 Consumo de CPU en tiempo de ejecución para cada framework.

Visualizando los datos obtenidos podemos notar que para ésta aplicación en particular los registros de uso de memoria difieren para el mismo framework con respecto al uso de CPU lo cual no ocurrió para neoeedit. Podemos ver entonces que JClouds es el framework más demandante en cuanto a uso de memoria, seguido muy de cerca por libcloud. Se desprende también que el mejor manejo de memoria lo posee Dasein ya que consumió tan solo 35.5MB para realizar la misma tarea. Aether en estas pruebas se ubicó en poco más de la mitad de los resultados.

En cuanto al uso de CPU se puede notar que nuevamente Dasein resultó ser el más eficiente, esta vez junto con JetS3t y ambos con 5 segundos de CPU. Aether fue el framework que necesito más tiempo de CPU con 9 segundos utilizados (4 segundos más que los frameworks más veloces).

5.2.2.2. Etapa 2

Una vez implementadas las migraciones, al igual que en la etapa dos del caso de estudio anterior, se procedió a realizar las migraciones entre protocolos soportados por cada framework. Luego de realizadas las migraciones correspondientes a esta etapa sobre los resultados de la anterior se obtuvo el conjunto de datos presentado en la figura 5.17 para la cantidad de líneas de código y configuración modificadas.

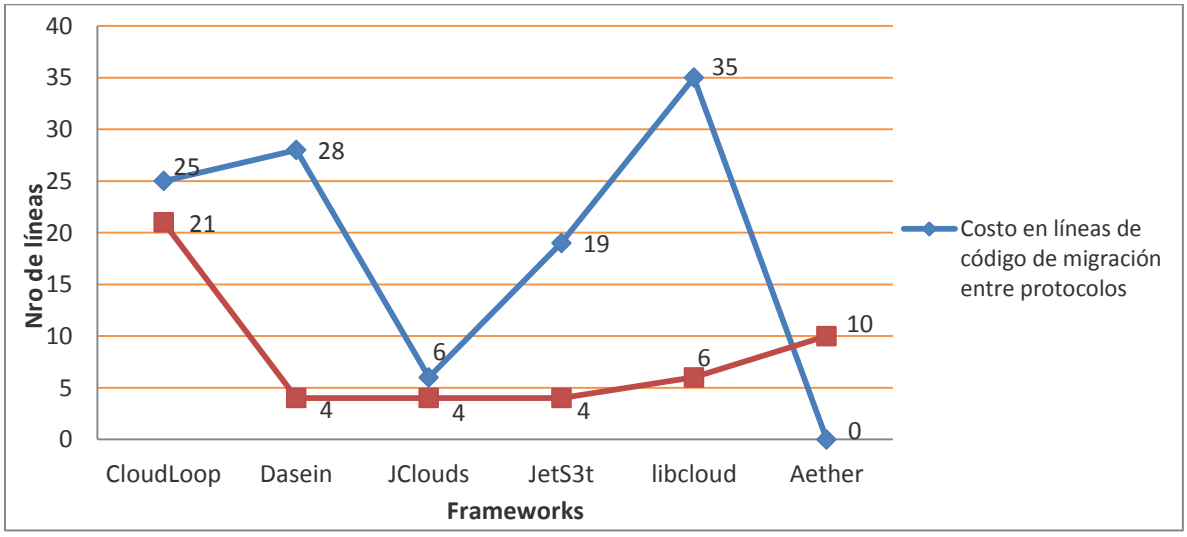

Figura 5.17 Líneas de código y configuración adicionales para migraciones entre protocolos.

Analizando los datos presentes en el gráfico se puede notar que para implementar la migración con Aether no fue necesario realizar ningún cambio en el código fuente. Por el contrario, todo el resto de los

frameworks vieron su base de código alterada entre 6 (JClouds) y 35 (libcloud) líneas. Esto provoca que Aether se destaque significativamente sobre el resto de los frameworks ya que nuevamente fue el único que no requirió de cambios sobre el código de la aplicación.

Al analizar los cambios de configuración, lo único destacable es que nuevamente CloudLoop fue el framework que más cambios requirió llevándolo a necesitar en total la modificación de 46 líneas. El resto de los frameworks tuvo niveles bajos, similares a los de la etapa anterior. En el caso puntual de Aether, se requirieron 10 líneas de configuración alteradas para migrar entre protocolos lo cual lo ubica segundo entre los que mayor cantidad de líneas de configuración debieron modificar y primero junto a JClouds en cantidad de líneas totales modificadas.

5.2.2.3. Etapa 3

En la etapa 3 del primer caso de estudio se mencionaron las desventajas que implica realizar las migraciones. Las mismas también aplican para este caso de estudio sin consideraciones particulares. Los resultados en cuanto a las líneas de código y configuración modificadas para cada implementación pueden apreciarse a continuación en la figura 5.18.

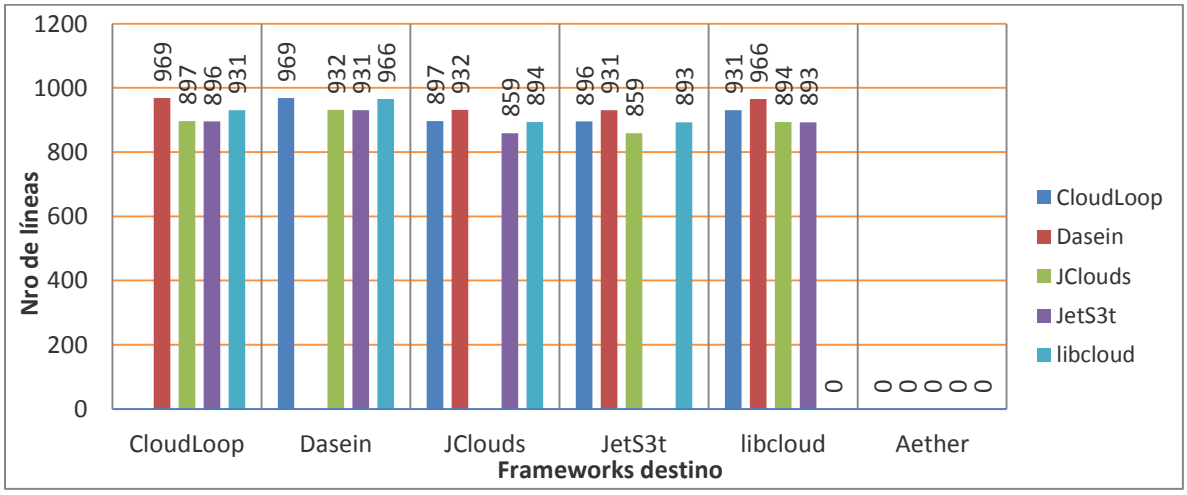

Figura 5.18 Líneas de código adicionales para migraciones entre frameworks.

En éste gráfico se ve claramente que ningún framework fue diseñado considerando ésta posibilidad de migración. Como resultado, el desarrollador se ve obligado a re-implementar cada solución utilizando el framework deseado y posiblemente empezando la codificación del almacenamiento desde cero. La única

excepción a esto es el framework Aether ya que no fue necesario realizar cambio alguno (sobre el código fuente) para llevar a cabo las migraciones.

Al migrar una aplicación es muy común que se modifiquen los datos de los archivos de configuración ya que se deben alterar ciertos comportamientos o valores específicos para el correcto funcionamiento. En la figura 5.19 se muestran los resultados correspondientes en cuanto a líneas de configuración modificadas por cada re-implementación.

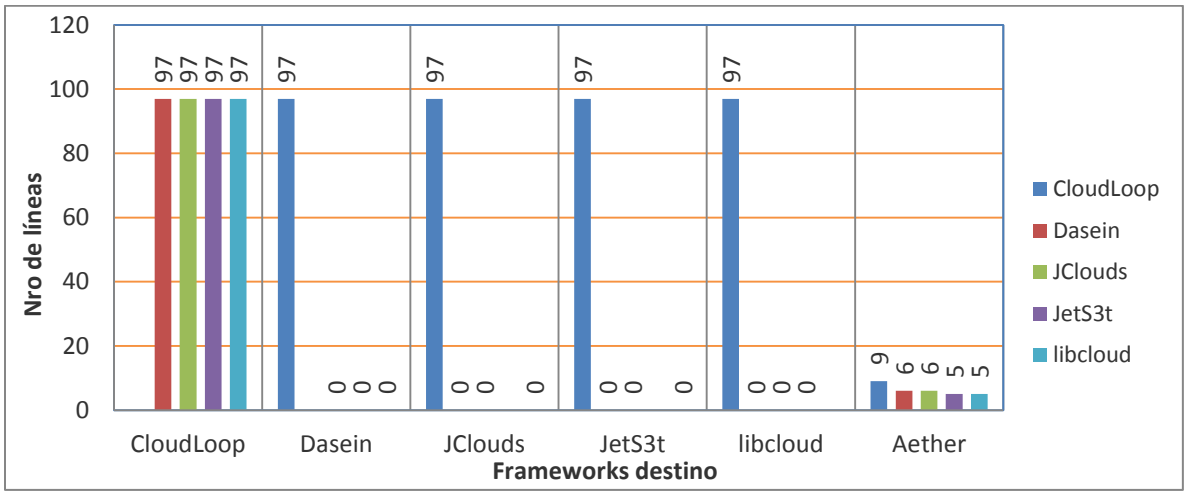

Figura 5.19 Líneas de configuración adicionales para migraciones entre frameworks.

Podemos apreciar que cada aplicación mantiene los niveles expuestos en las etapas previas del experimento. Al igual que en las etapas anteriores, fue CloudLoop el único que se destacó requiriendo la mayor cantidad de modificaciones sobre las líneas de configuración necesarias para su funcionamiento. El resto de las herramientas mantuvieron un nivel relativamente bajo y constante en cuanto a líneas de configuración modificadas.

Con respecto al último punto importante a tener en cuenta sobre la fase de desarrollo describiremos, al igual que se realizó en la primera fase, la complejidad de realizar cada re-implementación en términos de líneas de código consultado. Los datos obtenidos para ésta métrica en particular una vez finalizadas las implementaciones se presentan a continuación en la figura 5.20.

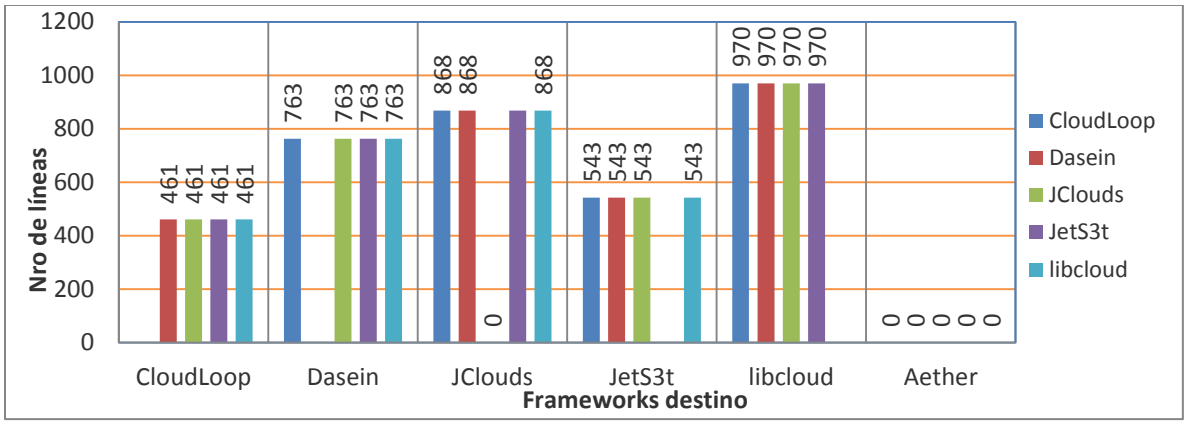

Figura 5.20 Líneas de código consultado para migraciones entre frameworks.

En cuanto a la cantidad de código que se debió consultar para realizar la tarea podemos notar que Aether fue el framework más eficiente ya que no fue necesario consultar el código fuente para la implementación. El resto de los frameworks poseen niveles idénticos a los de la etapa 1.

Como último punto resta analizar las migraciones a Aether en cuanto a uso de CPU y memoria. Para lograr esto se realizaron las ejecuciones del experimento descrito en la etapa 1 sobre cada una de las implementaciones. En las figuras 5.21 y 5.22 se pueden apreciar los resultados correspondientes.

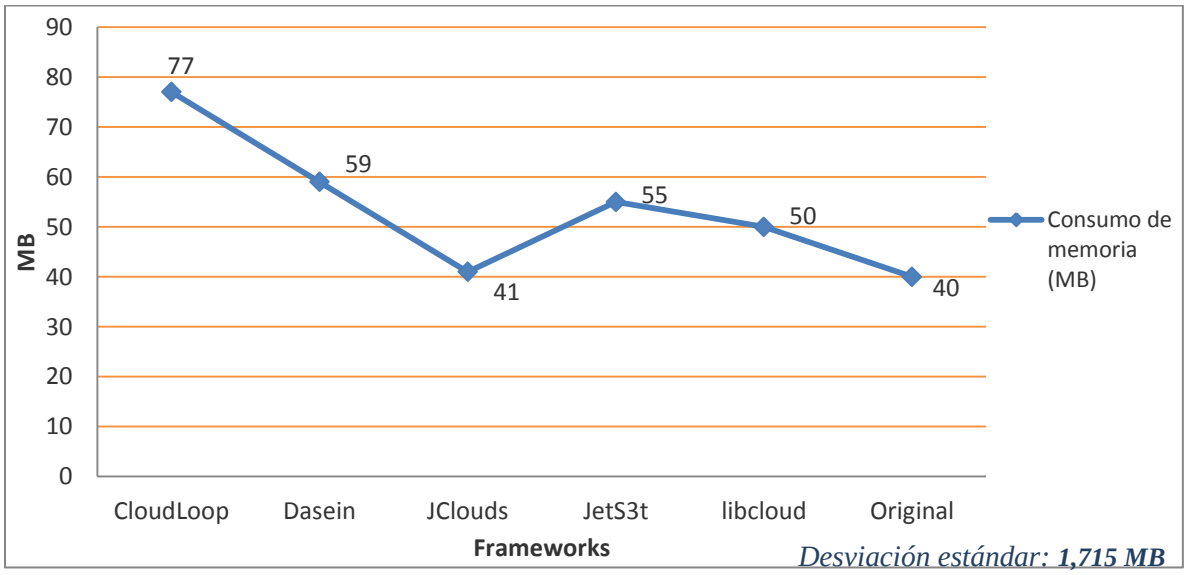

Figura 5.21 Consumo de memoria en tiempo de ejecución para migraciones a Aether.

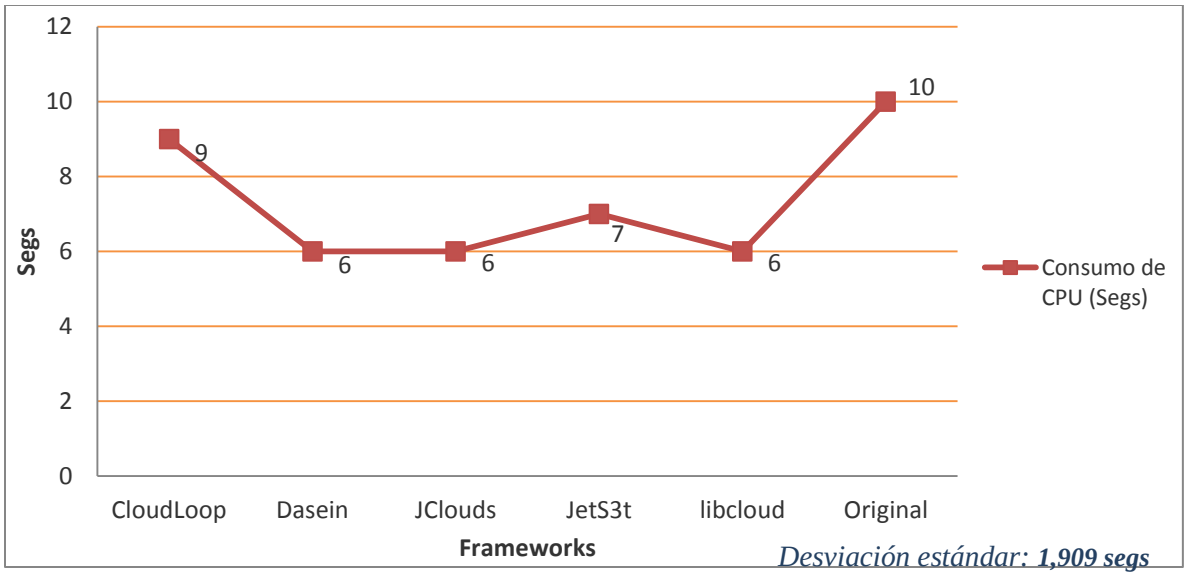

Figura 5.22 Consumo de CPU en tiempo de ejecución para migraciones a Aether.

Podemos observar que en general se mantienen los niveles de utilización de recursos observados en la primera etapa. El único dato notable es que la migración de CloudLoop a Aether necesito la mayor cantidad de memoria, consumiendo casi el doble que la aplicación original bajo la misma prueba (40MB contra 77MB).

5.2.3. Caso de estudio 3: MuCommander

Este tercer y último caso de estudio analiza el administrador de archivos "MuCommander". Esta aplicación permite la administración de archivos de manera sencilla sobre diferentes protocolos y sistemas operativos. El objetivo será agregar soporte para el almacenamiento remoto con cada framework a analizar. Para implementar las diferentes funcionalidades habrá que utilizar las facilidades de cada uno de estos frameworks tales como obtener metadatos, subir o bajar archivos, listarlos, etc. A continuación en la figura 5.23 se presenta la arquitectura simplificada de la aplicación.

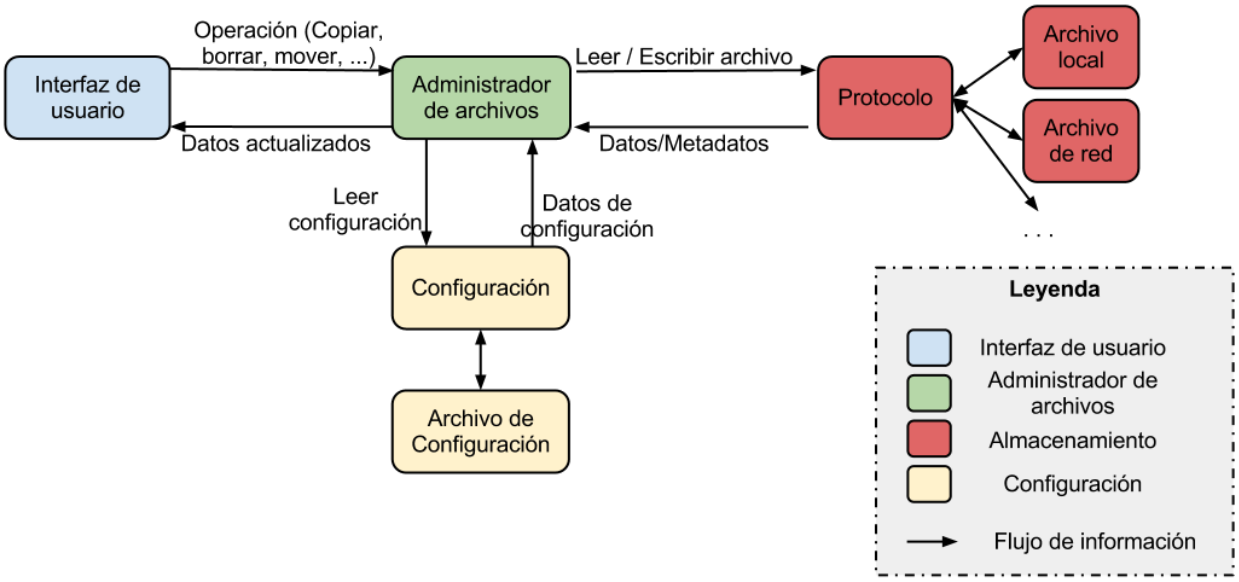

Figura 5.23 Diagrama abstracto del funcionamiento de MuCommander.

Como puede verse en la imagen anterior, el funcionamiento de la aplicación es un poco más complejo que el de las dos aplicaciones presentadas anteriormente. Al iniciar la aplicación se cargan las preferencias del usuario desde un archivo de configuración. Posterior a esto se permite la conexión a diferentes dispositivos de almacenamiento por medio de diversos protocolos. Esto permite al usuario hacer uso de una sola aplicación para administrar (copiar, borrar, renombrar, etc.) todos sus archivos de diversas fuentes (FTP, disco, etc).

Al igual que se realizó en los casos de estudio anteriores, antes de comenzar con el desarrollo de cada una de las tres etapas presentamos las métricas básicas (Líneas de código y Promedio de líneas de código por método) correspondientes a la aplicación original. Con estos datos tenemos una referencia del tamaño de la aplicación. Los valores de estas dos métricas se describen a continuación:

Líneas de código totales: 75814

Promedio de líneas de código por método: 7,07

5.2.3.1. Etapa 1

Previamente a la implementación de esta primera etapa se realizó un análisis inicial de la herramienta identificando las clases a modificar para cumplir con los objetivos planteados anteriormente. Gracias a que MuCommander soporta varios protocolos (archivos locales, ftp, smb, etc.) la inclusión de uno nuevo

basado en algún framework particular no resulta tan complicado. Las interfaces con las que se debe trabajar están bien definidas y los métodos y nombres de paquetes son claros.

Luego de implementadas las modificaciones para adaptar la aplicación a cada framework se obtuvieron las métricas correspondientes a esta etapa. A continuación en la figura 5.24 se presentan las líneas de código y configuración adicionales necesarias para cada implementación.

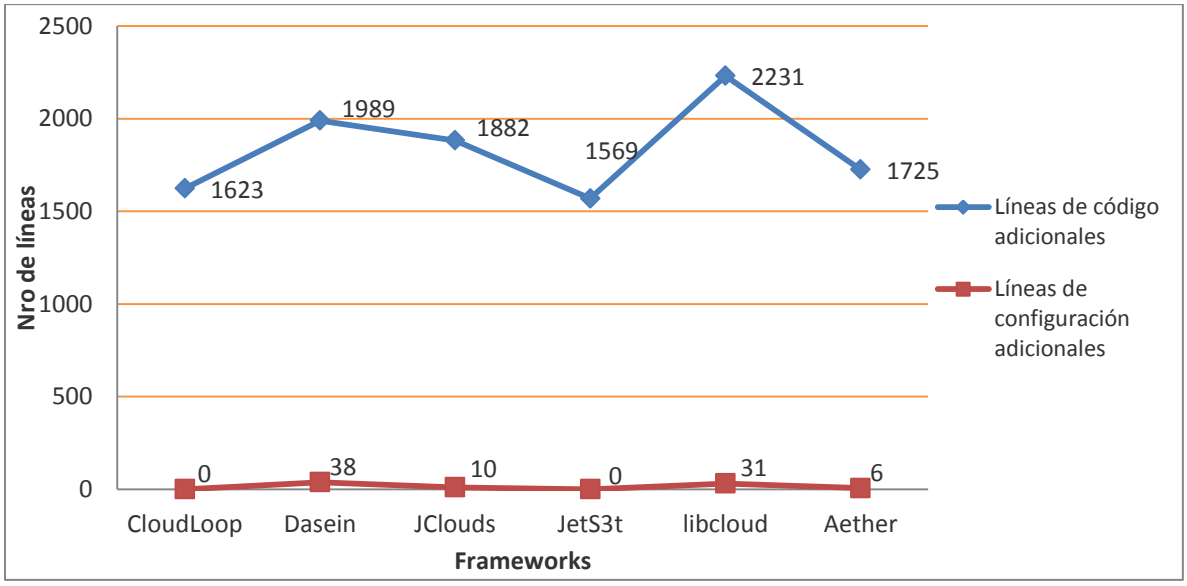

Figura 5.24 Líneas de código y configuración adicionales de cada implementación.

Analizando los datos presentes en el gráfico podemos apreciar que libcloud se destaca por sobre todos los demás haciendo uso de la mayor cantidad de líneas de código modificadas (2231 líneas) para su adaptación a la aplicación original. Como contrapunto de éste se encuentra JetS3t, el cual necesitó la menor cantidad de líneas modificadas. El resto de los frameworks tienen valores intermedios, destacando a Dasein que presenta también un costo relativamente elevado de codificación. Aether, por su parte, fue uno de los frameworks que menor cantidad de modificaciones en el código fuente necesitó con lo que se ubica entre los tres frameworks más eficientes en este sentido.

En cuanto a líneas de configuración podemos notar que tanto Dasein como Libcloud presentan valores elevados con respecto al resto, pero que igualmente no son alarmantes ya que se trata de unas pocas decenas de líneas. Los restantes frameworks obtuvieron valores bajos y hasta nulos para ésta medición.

Para poder llevar a cabo cada una de las implementaciones anteriores, el desarrollador debió consultar ciertos fragmentos de código pertenecientes a la aplicación y a cada framework. Este esfuerzo se midió en líneas de código consultado para cada desarrollo las cuales se presentan a continuación en la figura 5.25.

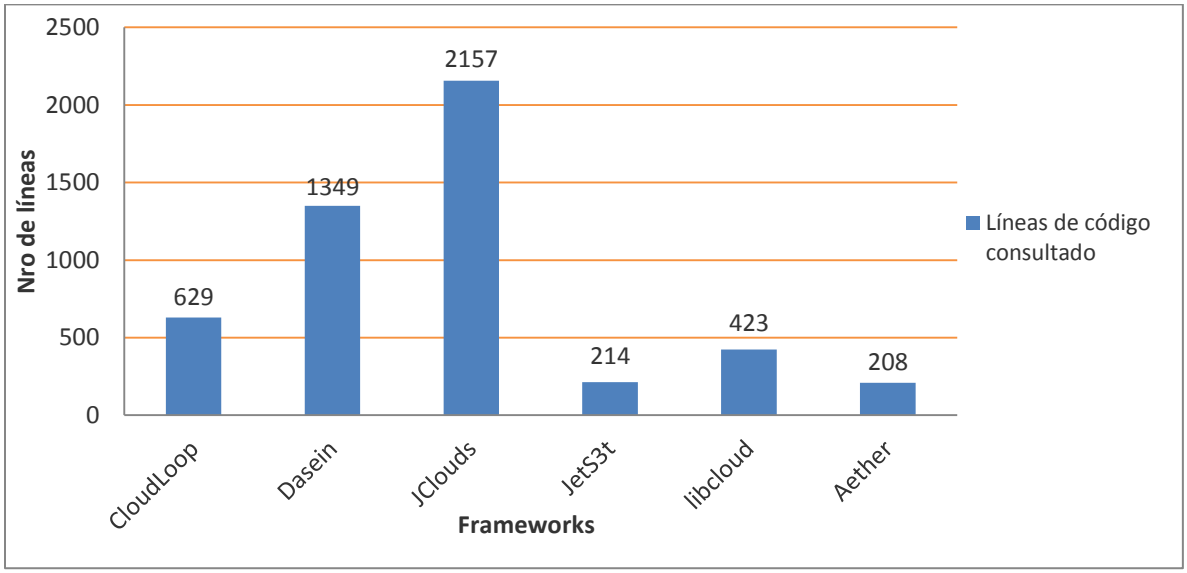

Figura 5.25 Líneas de código consultado para realizar cada implementación.

Como puede apreciarse en la figura anterior, los dos frameworks que más se destacan por sobre el resto con valores muy elevados para la métrica Líneas de código consultado son JClouds y Dasein, mientras que para el resto de los frameworks se mantuvo estable necesitando realizar menos de la mitad de las consultas al código. Entre los frameworks que menos líneas se debió consultar se encuentran Aether y JetS3t lo cual indica que la complejidad de los mismos para esta implementación fue baja en comparación al resto.

Una vez realizadas las diferentes implementaciones nos encontramos en el punto de determinar el comportamiento de cada framework en tiempo de ejecución con respecto al Consumo de memoria y Uso de CPU, para esto se definió la siguiente secuencia de pasos a seguir:

1. Copiar todos los archivos a un directorio temporal menos la carpeta “MOVER”.
2. Renombrar el archivo “tabla1.jpg” a “tabla.jpg”.
3. Mover la carpeta “MOVER” al directorio temporal.
4. Eliminar todos los archivos del repositorio remoto.
5. Mover todos los archivos de la carpeta temporal al repositorio remoto.

NOTA: Esta prueba se realizará utilizando una estructura de carpetas y archivos con las siguientes características:

49 archivos, 10 carpetas/subcarpetas y 2.83 Mb en total.

Luego de ejecutado el caso de prueba sobre cada implementación se obtuvieron los resultados detallados en las figuras 5.26 en términos de consumo de memoria y 5.27 en relación al uso de CPU. Se indican además en éstos gráficos los consumos para la misma prueba por parte de la aplicación original**.**

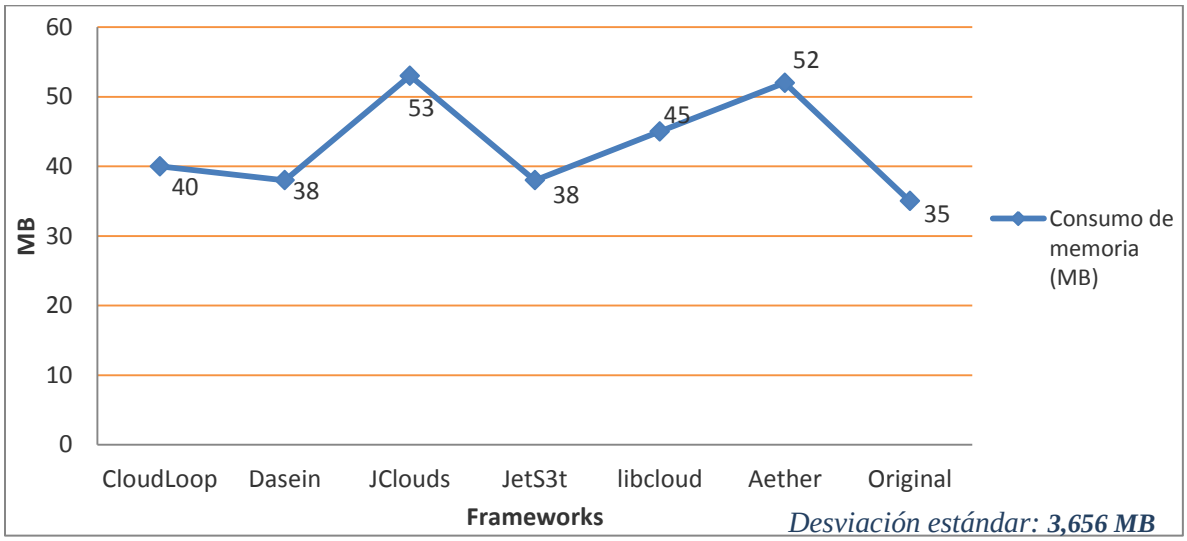

Figura 5.26 Consumo de memoria en tiempo de ejecución para cada framework.

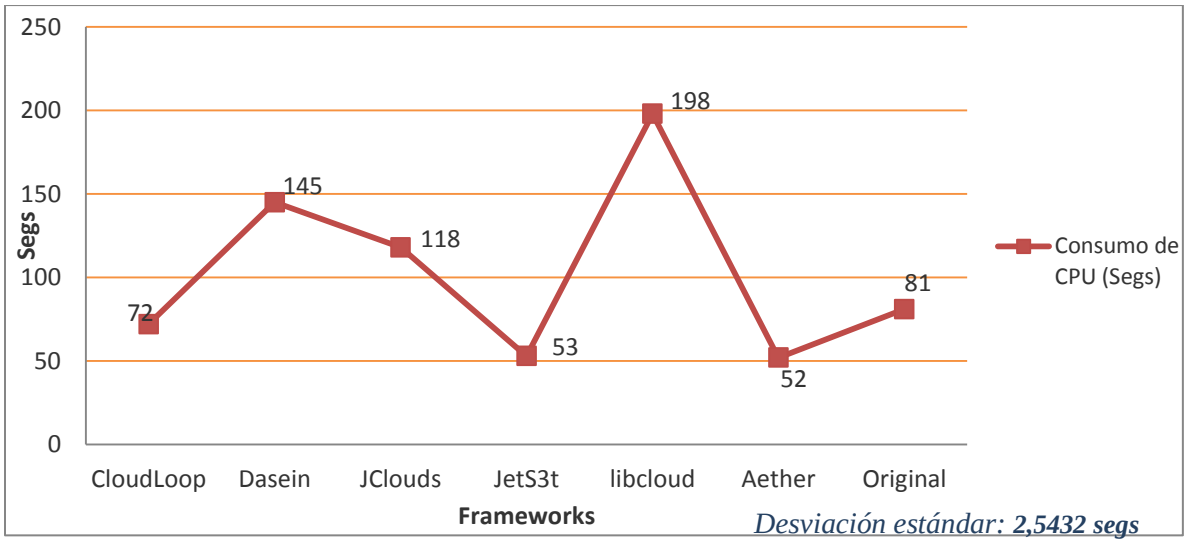

Figura 5.27 Consumo de CPU en tiempo de ejecución para cada framework.

Analizando estos datos podemos resaltar que JClouds y Aether resultan los frameworks menos eficientes en cuanto al uso de memoria. Por el contrario, Dasein y JetS3t resultan ser los más eficientes en este apartado. La situación cambia al tomar en cuenta la utilización de CPU donde Aether pasa a ser el framework con mejor eficiencia consumiendo sólo 52 segundos de CPU mientras que libcloud es el más ineficiente necesitando para realizar la misma tarea casi el cuádruple de tiempo, una medida bastante superior a la del resto de los frameworks competidores.

5.2.3.2. Etapa 2

Una vez finalizados los pasos correspondientes a la etapa anterior, se procedió a realizar las migraciones entre protocolos soportados por cada framework. Luego de realizadas las migraciones correspondientes a esta etapa sobre los resultados de la anterior se obtuvo el conjunto de datos presentado en la figura 5.28 para la cantidad de líneas de código y configuración modificadas.

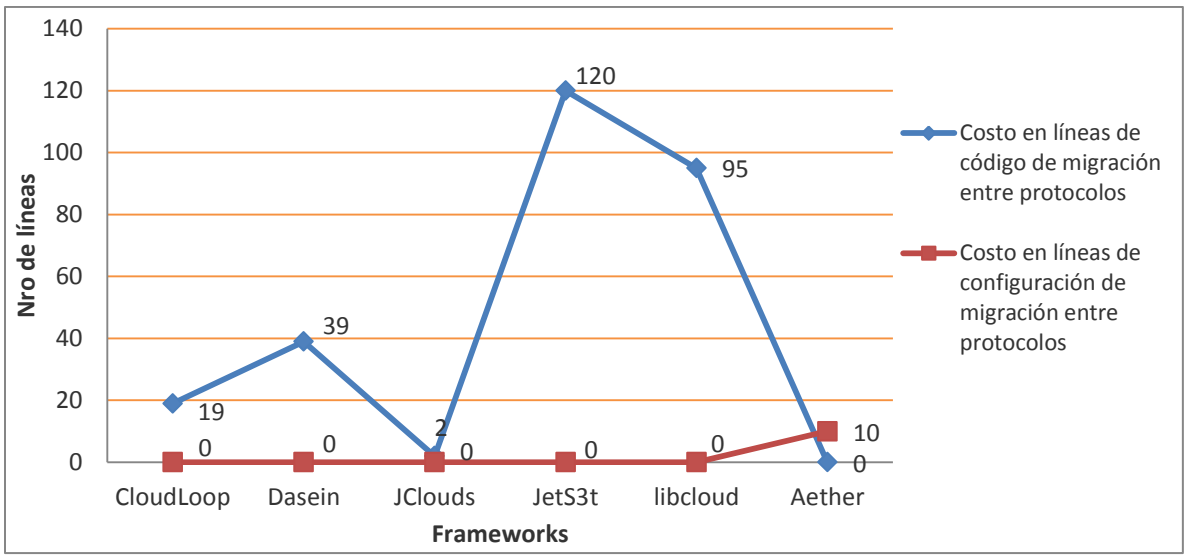

Figura 5.28 Líneas de código y configuración adicionales para migraciones entre protocolos.

Como se puede apreciar en el gráfico anterior, no fue necesario introducir cambios en el código fuente de la aplicación para implementar la migración de protocolo con Aether, el cual fue seguido muy de cerca por JClouds con sólo 2 líneas modificadas. Por el contrario, libcloud y JetS3t fueron los frameworks que más recodificación demandaron alcanzando las 95 y 120 líneas respectivamente. Si visualizamos la cantidad de líneas de configuración alteradas podemos ver que el único framework que se destaca por

haber necesitado modificaciones es Aether, aunque sólo fue necesario alterar 10 líneas. En términos generales podemos ver que los dos frameworks con menor cantidad de cambios necesarios para realizar la modificación de protocolos son tanto Aether como JClouds.

5.2.3.3. Etapa 3

En la etapa 3 del primer caso de estudio se mencionaron las desventajas que implica realizar las migraciones por lo que pasamos a presentar a continuación los resultados de cada una de estas implementaciones. Los datos correspondientes a las líneas de código y configuración modificadas para cada implementación pueden apreciarse a continuación en la figura 5.29.

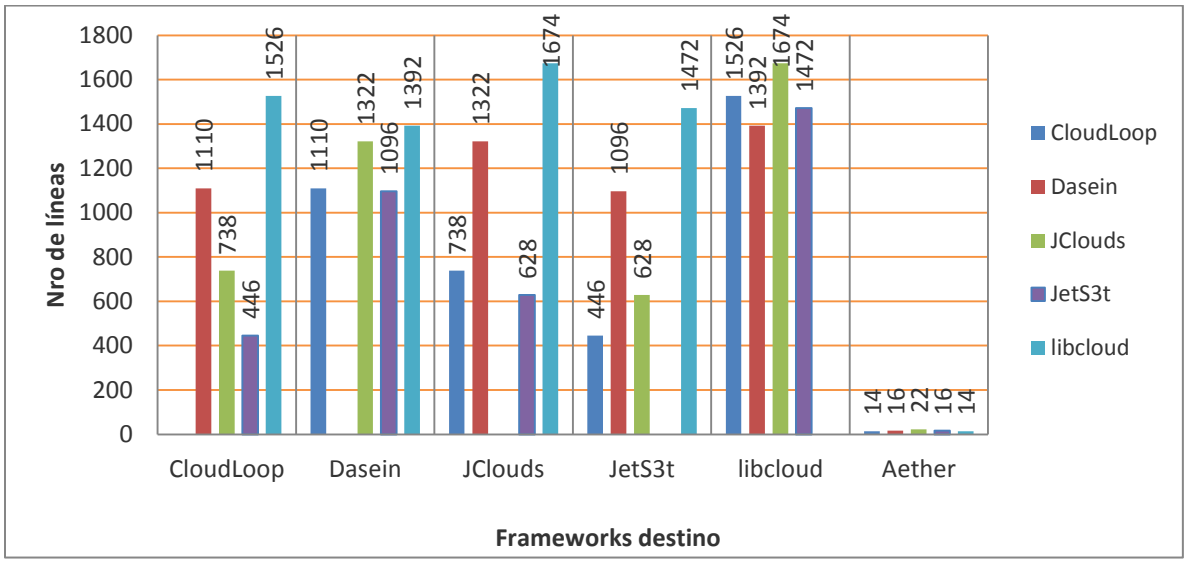

Figura 5.29 Líneas de código adicionales para migraciones entre frameworks.

Como puede verse en el gráfico, la mayoría de los frameworks presentan una cantidad elevada de líneas de código fuente modificadas variando desde 446 a 1674 líneas de código fuente modificadas. Como se mencionó también en los experimentos realizados sobre las dos aplicaciones anteriores, esto se debe a que los frameworks no poseen mecanismos para facilitar este tipo de migración. El único caso sobresaliente es el de Aether ya que presento una cantidad muy baja de líneas de código fuente modificadas (entre 14 y 22). Esto lo convierte en la herramienta ideal para esta situación ya que sólo es necesario un pequeño esfuerzo por parte del programador de la aplicación para realizar la adaptación.

Como se ha visto anteriormente, al migrar una aplicación es muy probable que se modifiquen a su vez los archivos de configuración ya que se deben indicar en ciertas situaciones nuevos valores o atributos para el correcto funcionamiento. El gráfico de la figura 5.30 detalla estos resultados.

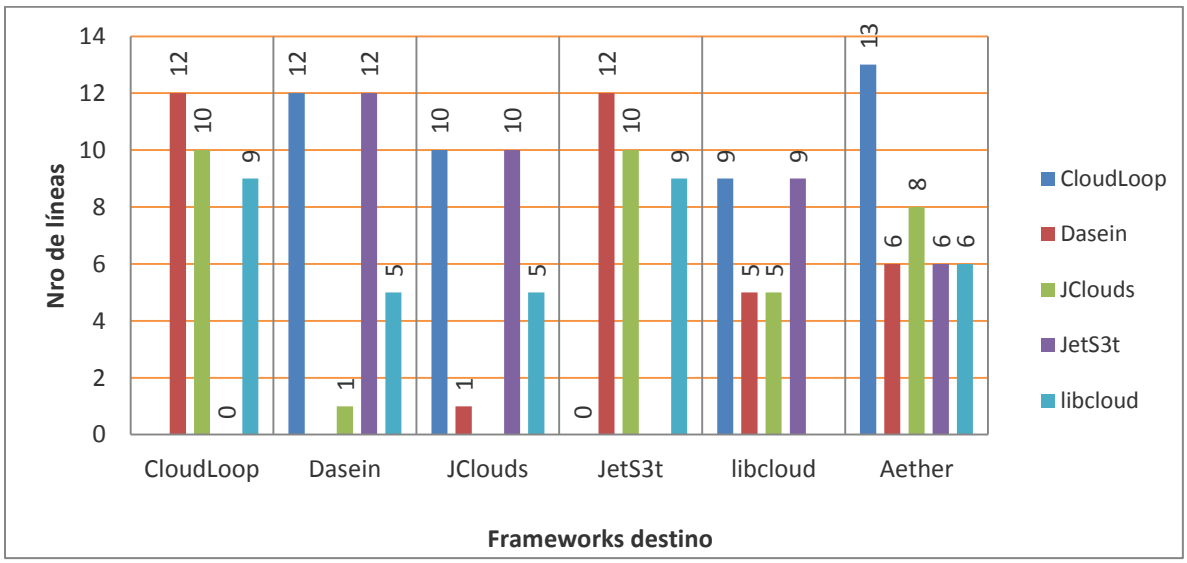

Figura 5.30 Líneas de configuración adicionales para migraciones entre frameworks.

Puede apreciarse que la cantidad de líneas de configuración modificadas para cada implementación se mantiene relativamente baja, no pasando de las 13 líneas. Si comparamos estas con las líneas de código fuente modificadas podemos ver que la cantidad resulta insignificante.

Como último punto a analizar correspondiente a la etapa de desarrollo describiremos la complejidad de realizar cada re-implementación en términos de líneas de código consultado. La figura 5.31 presenta los datos obtenidos para ésta métrica particular luego de finalizadas las implementaciones.

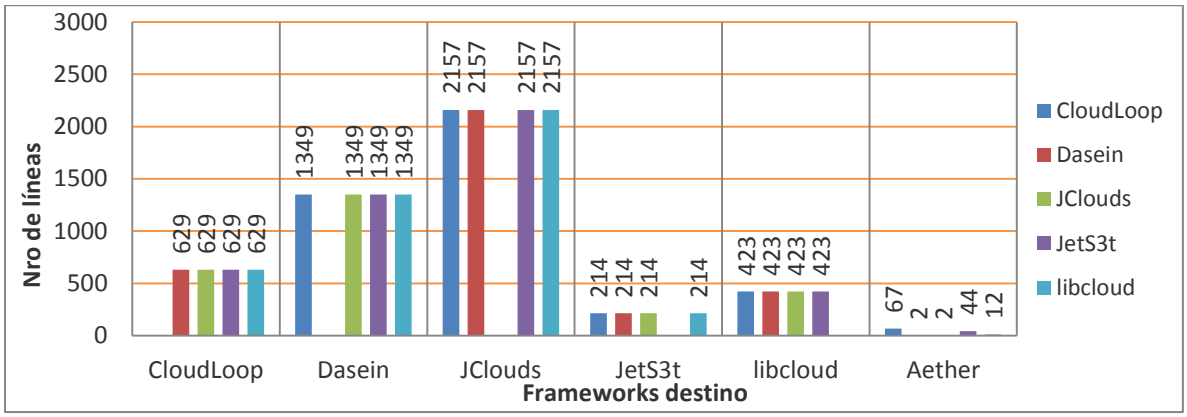

Figura 5.31 Líneas de código consultado para migraciones entre frameworks.

El hecho de que Aether requiera pocos cambios de código ayuda en reducir la cantidad de código que el usuario debe consultar para realizar la migración, debido a esto Aether es el framework más destacado en cuanto a Cantidad de código consultado para realizar la implementación. Los valores de ésta métrica para el resto de los frameworks analizados varió bastante encontrándose a JetS3t en el segundo lugar de los más sencillos y a JClouds como el más complejo, para el cuál se debieron consultar más de 2100 líneas de código.

Para concluir los experimentos realizados queda por delante analizar el comportamiento en tiempo de ejecución de cada migración realizada a Aether. Nuevamente para lograr esto se procedió a ejecutar el caso de test descrito al finalizar la etapa 1 sobre cada implementación. Las figuras 5.32 y 5.33 presentan los resultados para cada migración correspondiente en cuanto a Consumo de memoria y Uso de CPU.

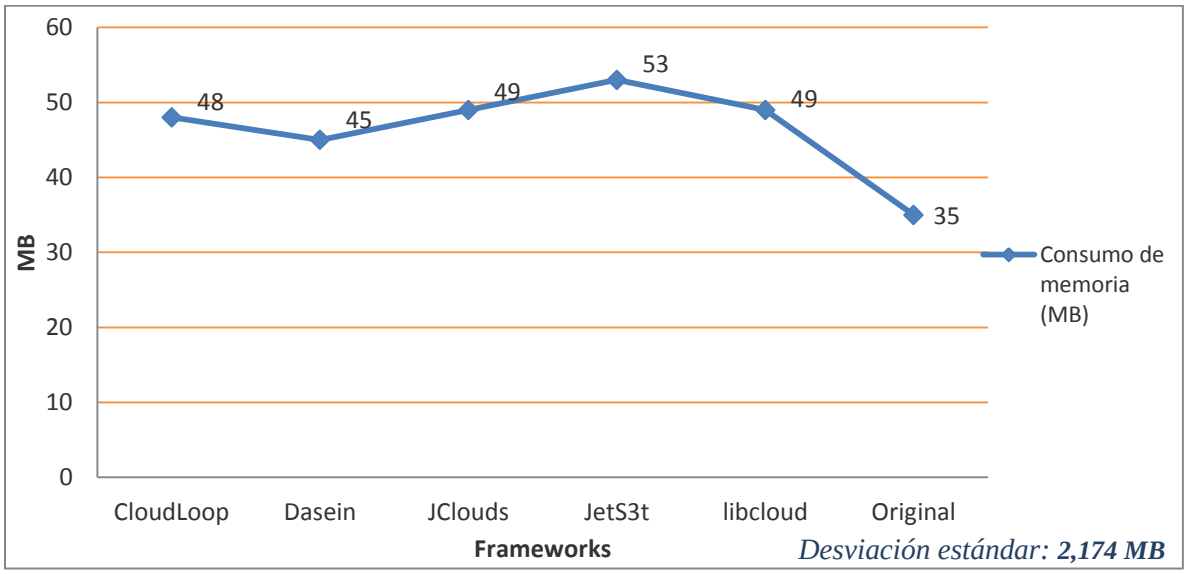

Figura 5.32 Consumo de memoria en tiempo de ejecución para migraciones a Aether.

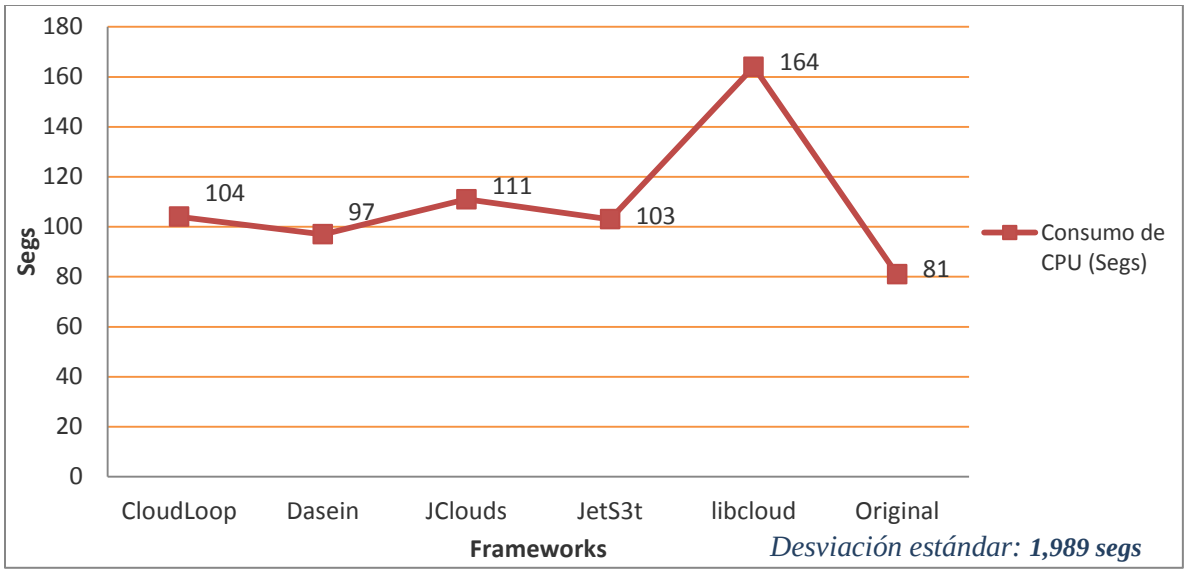

Figura 5.33 Consumo de CPU en tiempo de ejecución para migraciones a Aether.

Se puede apreciar que los Consumos de memoria para cada migración se mantienen estables y similares a los obtenidos en la primera etapa del experimento. No ocurre lo mismo para el Consumo de CPU ya que se ve que al utilizar Aether los valores tienden a estabilizarse lo cual provoca ciertas variaciones en los resultados dependiendo del framework en cuestión. Para las migraciones a Aether desde Dasein, JClouds y libcloud puede apreciarse que disminuye el Uso de CPU mientras que para CloudLoop y JetS3t el consumo de CPU aumenta.

En líneas generales, se presenta disparidad en el uso de CPU, basada en el adapter de Aether utilizado y la estructura original de cada framework.

5.3 Conclusiones generales

Analizando en conjunto los resultados de los casos de estudio se pueden extraer las siguientes conclusiones:

- En todos los casos de estudio la cantidad de código requerida para implementar la “etapa 1” fue similar al producido por la competencia. Analizando los valores obtenidos en las pruebas podemos ver que se destacan libCloud con la mayor cantidad de líneas modificadas (1063) y JetS3t con la menor cantidad, tan sólo 823. A continuación se presenta el gráfico que resume los

resultados obtenidos para ésta métrica. Cabe destacar que Aether requirió sólo 55 líneas más que la mejor solución.

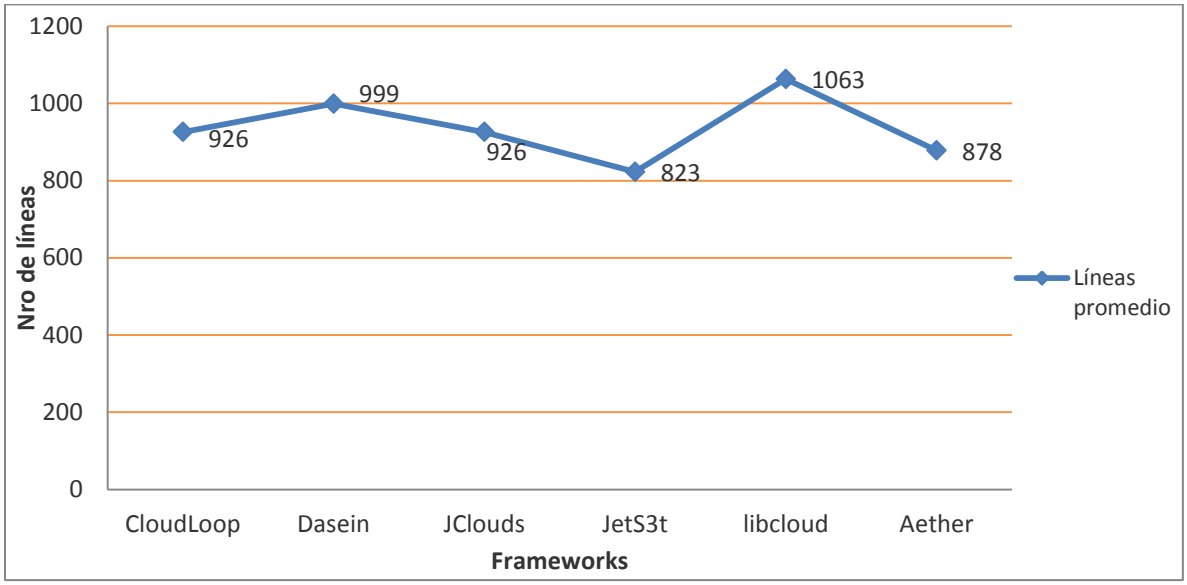

Figura 5.34 Líneas de código promedio adicionales para cada implementación.

- La cantidad de código que debió consultarse para lograr implementar la "etapa 1" con Aether se mantuvo muy baja en todos los casos, resultando el mejor framework en la categoría y seguido muy de cerca por JetS3t. Puede verse también que tanto Dasein como JClouds fueron los peores en este sentido. El gráfico siguiente muestra los resultados de los experimentos realizados.

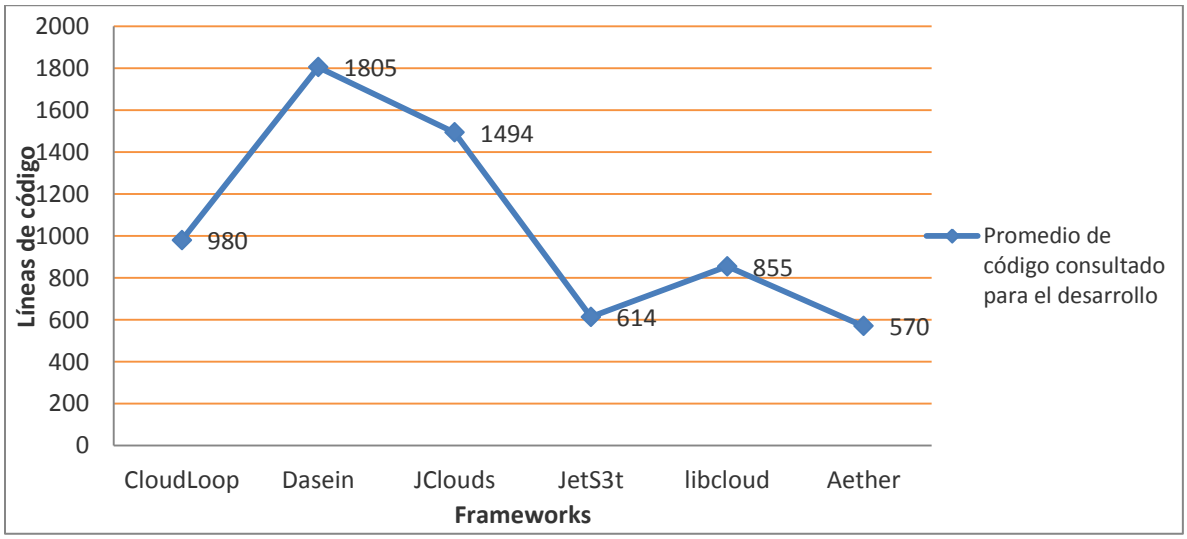

Figura 5.35 Promedio de código consultado para el desarrollo.

- En cuanto al consumo de memoria en la “etapa 1”, se puede apreciar que la implementación con cada framework tiende a seguir una línea con respecto a las ejecuciones de las aplicaciones restantes que utilizan el mismo framework. JClouds tiende a ser uno de los frameworks que más consumo de memoria requiere en la ejecución, seguido de cerca por Aether. Mientras tanto, Dasein y JetS3T se corresponden con los que menos recursos de este tipo utilizan.

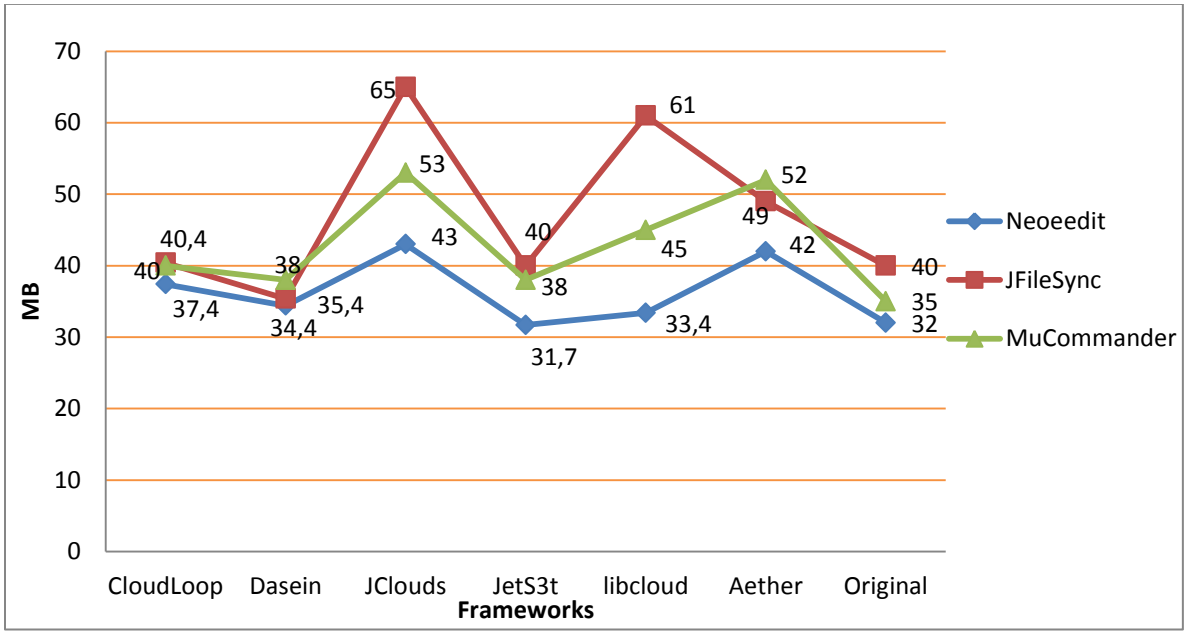

Figura 5.36 Consumo de memoria para cada implementación.

- No hay conclusiones claras en cuanto al uso de CPU en la “etapa 1” ya que los resultados obtenidos varían para las diferentes aplicaciones analizadas. Esto puede deberse a la naturaleza de cada aplicación ya que el objetivo de cada una es diferente al de las demás, lo cual en ciertas ocasiones puede demandar diferente nivel de uso de CPU. De todas maneras, se puede observar cierta estabilidad quitando de la comparación a libcloud (ya que en las pruebas realizadas sobre MuCommander generó un pico de 198 segundos de CPU).Cabe destacar que el uso de este tipo de recursos por parte de Aether se mantuvo dentro de límites razonables y semejante al del resto de los frameworks.

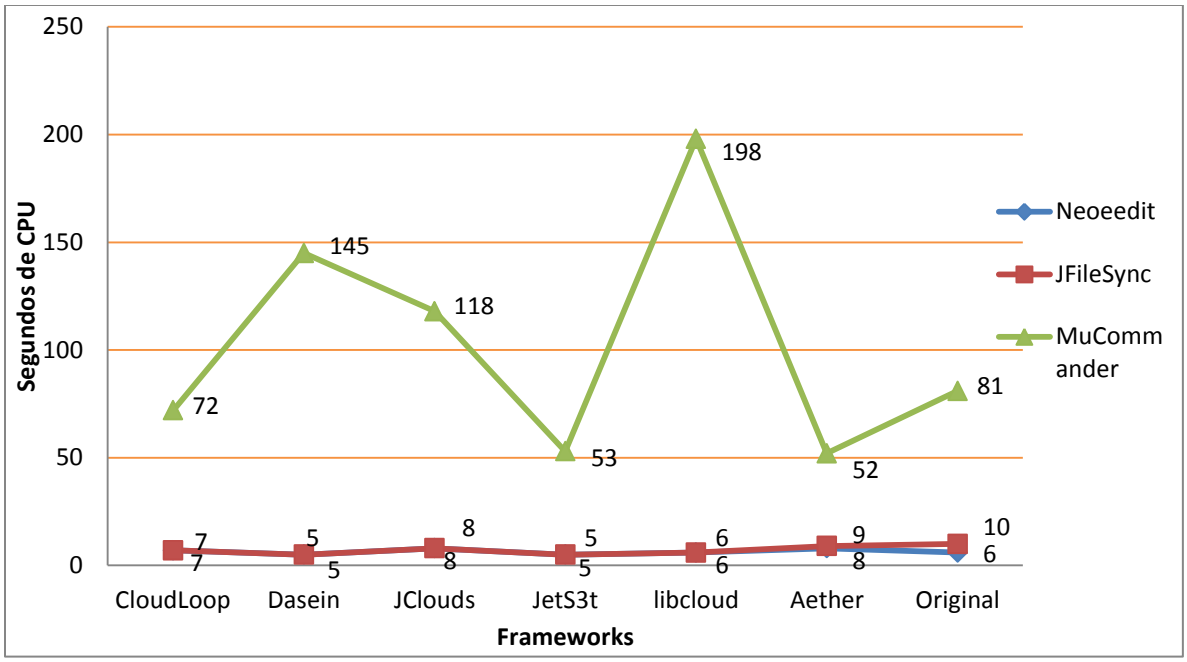

Figura 5.37 Uso de CPU para cada implementación.

- Analizando en éste punto la cantidad de líneas modificadas (de código fuente y configuración) para lograr la migración entre protocolos, puede notarse que Aether junto a JClouds fueron los frameworks que menos modificaciones debieron sufrir para tal fin. Un dato importante para resaltar en favor de Aether es que las 10 líneas que figuran como modificadas en promedio correspondieron sólo a líneas de configuración y no código fuente de la aplicación. Como caso opuesto nos encontramos con JetS3t y libcloud los cuales necesitaron realizar la mayor cantidad de cambios sobre las aplicaciones. A continuación se resume en promedio la cantidad de líneas modificadas totales por cada framework para realizar la migración.

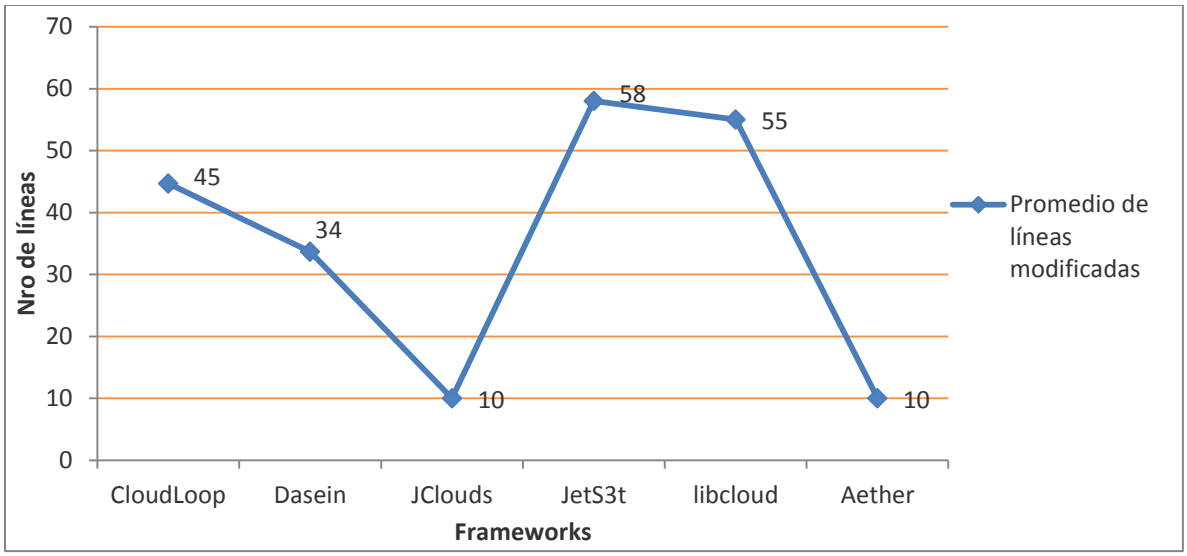

Figura 5.38 Promedio de líneas modificadas para migración entre protocolos.

- Al realizar las migraciones de cada aplicación ya codificada a otro framework podemos destacar que Aether fue ampliamente superior al resto de los frameworks. En todos los casos se pudo notar que Aether requirió una cantidad muy baja de cambios tanto en líneas de configuración como de código presentando variaciones entre 10 y 20 líneas. Por el contrario, el resto de los frameworks necesitó de unos cuantos cientos de líneas, incluso algunos superando las mil. A continuación se presenta el gráfico que resume los datos obtenidos.

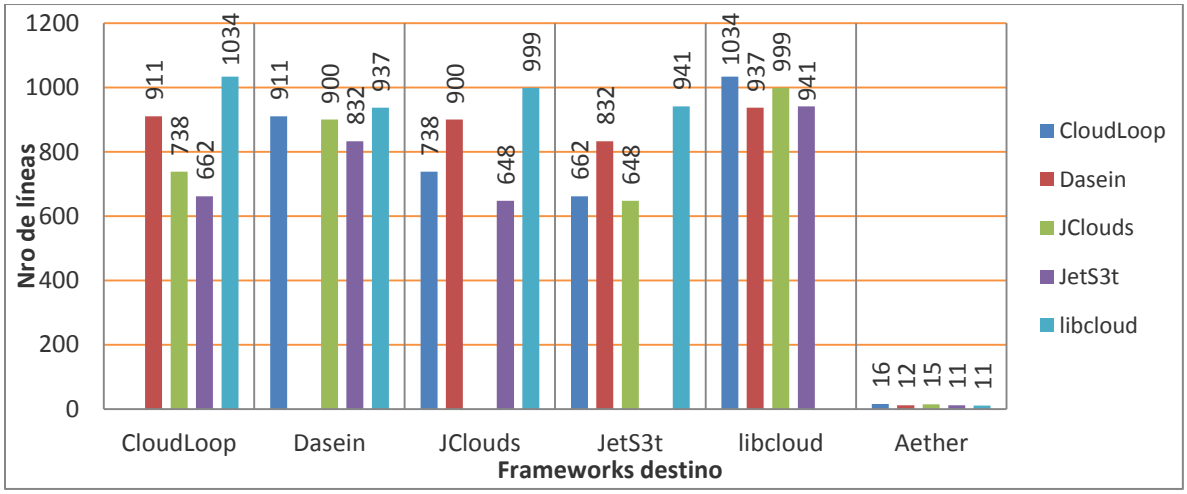

Figura 5.39 Líneas promedio para migraciones entre frameworks.

- Como último punto a analizar, en la "etapa 3" se obtuvieron resultados mixtos en cuanto al uso de recursos sobre las migraciones a Aether. En general tanto el uso de CPU como el consumo de

memoria se vieron afectados luego de la migración a Aether. Este último se vio afectado de diferentes maneras ya que para los frameworks como JClouds, JetS3t y libcloud el uso de memoria en promedio tendió a disminuir, mientras que para Cloudloop y Daseín ésta métrica en promedio tendió a aumentar. Con respecto al uso de CPU se puede notar que para casi todas las implementaciones los tiempos tienden a aumentar levemente. Este aumento es casi insignificante ya que se trata de unos pocos segundos en los peores casos. Este incremento puede deberse a que Aether traduce cada invocación realizada sobre los frameworks particulares a un conjunto de llamadas a sus métodos para luego procesar los datos y brindar la respuesta con su formato correspondiente. Puede apreciarse también que para el caso de libcloud el tiempo se redujo considerablemente.

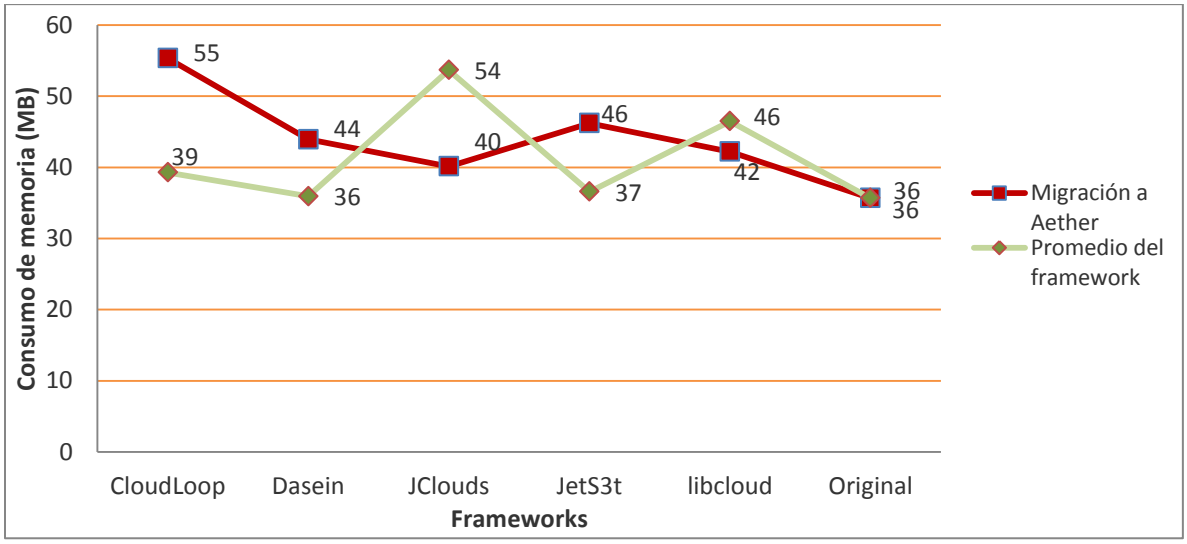

Figura 5.40 Consumo de memoria promedio para cada migración a Aether en contraste con el promedio para cada framework.

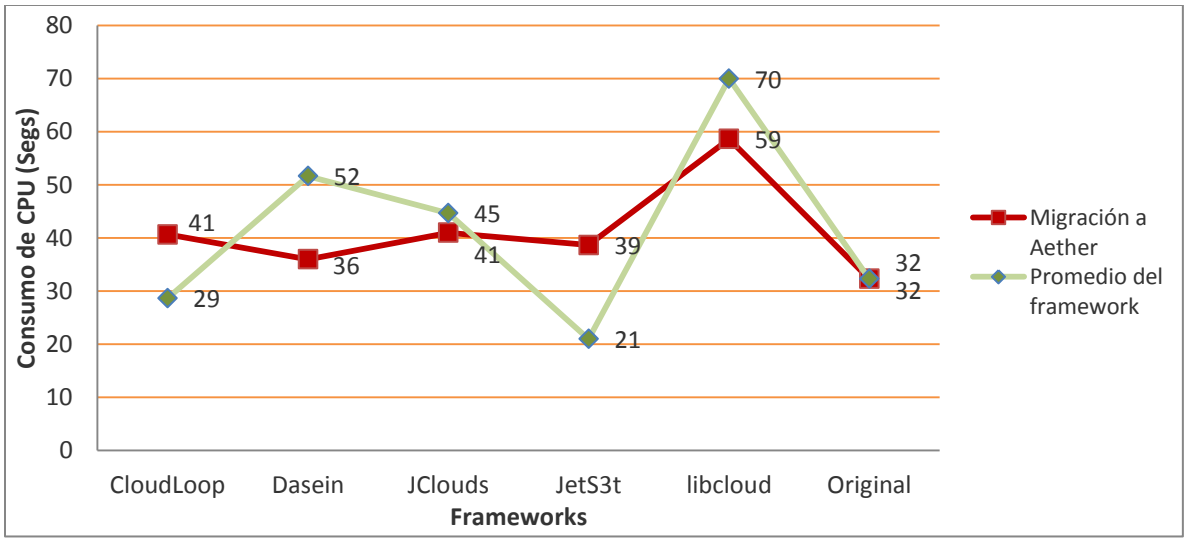

Figura 5.41 Uso de CPU promedio para cada migración a Aether en contraste con el promedio para cada framework.

En resumen, Aether ha mejorado en aspectos como cantidad de líneas de código consultadas, cantidad de líneas modificadas para migración entre protocolos y es muy superior al resto de los frameworks en cuanto a la cantidad de líneas de código necesarias para migrar las aplicaciones entre frameworks. Los valores de cantidad de líneas de código adicionales para la implementación se mantuvieron dentro de los valores de los mejores frameworks ubicándolo segundo. Por último, las principales limitaciones fueron el consumo de memoria y en menor medida el uso de CPU tanto para el uso normal como para las migraciones a Aether.

CAPÍTULO 6

6. CONCLUSIONES

En este trabajo de tesis se presentó un framework capaz de abstraer los diferentes servicios que se brindan sobre *Cloud* y soportar migraciones entre diversos proveedores de servicios y herramientas. Ambos ejes fueron tratados en pos de hacer frente a los problemas con que se encuentran los desarrolladores y empresas al momento de decidir un cambio de proveedor de servicios para sus aplicaciones.

El primer paso de este trabajo fue recolectar datos de las herramientas disponibles actualmente para trabajar con servicios en *Cloud*. Estos datos fueron analizardos con el objetivo de detectar los puntos fuertes y débiles de cada una de las herramientas. El resultado de este análisis puso en evidencia una serie de problemas comunes a todas las herramientas que se relacionan fundamentalmente con las capacidades de migración. Por un lado, se distinguió la migración de aplicaciones entre servicios, por ejemplo de Amazon S3 a Google Storage. Este tipo de migración suele ser tenida en cuenta por los diseñadores de las herramientas para *Cloud Computing*, pero no se encuentra disponible un mecanismo que realmente facilite la tarea al usuario ya que en muchos casos debe volver a codificar para adaptarse a los objetos del nuevo servicio. Por otro lado se analizó la migración de aplicaciones entre herramientas o frameworks para *Cloud Computing*. Un ejemplo de esto sería tener una aplicación codificada con el SDK de Amazon S3 y querer comenzar a utilizar Google Storage provisto por jClouds. Como pudimos ver, ninguna de las herramientas disponibles en la actualidad considera este tipo de migración y esto deriva en que el desarrollador deba recodificar la aplicación para utilizar la nueva plataforma.

De acuerdo a la problemática planteada anteriormente, el framework desarrollado permite reducir considerablemente el esfuerzo de codificación al realizar estas migraciones. Para lograr esto se trabajó con un modelo basado en reflexión y reemplazo en tiempo de ejecución de código fuente. Los resultados fueron muy positivos en todos los escenarios de migración estudiados. En los casos más extremos, Aether logro reducir el costo de codificación y configuración del orden de las mil líneas a tan solo una o dos decenas. Es importante destacar que en la mayoría de los casos alcanzó con modificar archivos de configuración para indicar los nuevos datos de la conexión al servicio y el adaptador que se debe utilizar para interpretar las invocaciones que se hacen a los servicios. Las aplicaciones que requirieron modificaciones en el código fuente corresponden a aquellas que poseen cargadores de clases personalizados. En estos casos se debieron modificar los cargadores de clases para delegar esta carga a Aether.

6.1 Limitaciones de la herramienta

La principal limitación de la herramienta radica en que está desarrollada para utilizarse solamente en aplicaciones escritas en lenguaje Java. Esto reduce el rango de aplicabilidad a desarrollos ya existentes escritos en un lenguaje diferente.

Otra limitación que encontramos se corresponde con la administración de las dependencias. Debido a que se utiliza MAVEN para administrar las dependencias utilizadas por el framework, es posible caer en la situación en que un proveedor de alguna biblioteca modifique el código fuente de la versión que se utiliza y cree algún tipo de conflicto con las clases que la utilizan. Esta limitación es sencilla de solventar. En caso de que ocurra algún problema de este tipo se podría recurrir a descargar al repositorio de MAVEN la versión de la biblioteca que se estaba utilizando de manera correcta o bien agregarla al proyecto como una bibioteca estática e indicar esto en el archivo de configuración de MAVEN (pom.xml).

Con respecto a la usabilidad de la herramienta, como se mencionó en capítulos anteriores, nos encontramos con dificultades al momento de utilizar este framework en aplicaciones ya desarrolladas utilizando otro framework y que poseen un cargador de clases (classloader) personalizado. En estos casos el cargador de clases de Aether no funcionará de manera correcta debido a que las clases son cargadas por el classloader de la aplicación. Para solventar este problema el programador deberá modificar el código de la aplicación desarrollada para delegar la carga de clases al cargador de Aether permitiendo de esta forma que este cargue y modifique las clases que sean necesarias.

6.2. Trabajos futuros

Existen distintos aspectos de la herramienta que pueden ser mejorados como trabajos futuros e incluso extendidos para lograr tener un mejor provecho de su utilización:

- Usabilidad: La usabilidad es un atributo muy importante en este tipo de sistemas. Los experimentos que se llevaron a cabo a lo largo del documento fueron realizados por usuarios expertos en el dominio. Debido a esto sería deseable poder probar la utilización de la herramienta por parte de otros usuarios con diferentes conocimientos.

- Versatilidad de servicios: sería bastante útil extender el framework para abarcar otros tipos de servicios además del servicio de almacenamiento ya implementado. Este framework fue diseñado

para poder extender los servicios e implementar el que fuera necesario de manera sencilla. Debido a que nos basamos en servicios de *Cloud* para desarrollar el trabajo sería importante agregarle los servicios por ejemplo de colas (queue) y cómputo (compute).

- Soporte de métricas: Otro punto interesante sería agregar al framework soporte para obtener métricas sobre las ejecuciones de los servicios. Debido a que es muy sencillo intercambiar los proveedores de servicios y no es necesario modificar el código fuente de la aplicación se podrían tomar métricas fácilmente para los diferentes proveedores y poder comparar de esta manera el rendimiento de cada uno. Por ejemplo comparando el tiempo que tarda en ejecutar un determinado servicio, o realizar algún cálculo específico.

- Soporte de backup: Aunque la gran mayoría de los proveedores de servicios tienen métodos para realizar backups para solventar ciertos errores y caídas de sistemas, puede extenderse el framework para proveer al usuario una alternativa a los mecanismos utilizados por cada proveedor. Esto podría ser utilizado para mantener de manera transparente al usuario copias de los datos vitales de una aplicación en otro servidor o proveedor alternativo y, de esta manera, tener acceso a esos datos en casos de fallos en el servicio principal.

BIBLIOGRAFÍA

Alegsa. (2013). *Diccionario de informática*. Recuperado el 25 de Marzo de 2013, de http://www.alegsa.com.ar/Dic/tecnologias%20de%20la%20informacion.php

Amazon. (2013). *Amazon AWS SDK para Java*. Recuperado el 5 de Marzo de 2013, de http://aws.amazon.com/sdkforjava/

Amazon. (2013). *Amazon EC2*. Recuperado el 3 de Mayo de 2013, de http://aws.amazon.com/es/ec2/

Amazon. (2013). *Amazon EC2 Pricing*. Recuperado el 3 de Mayo de 2013, de http://aws.amazon.com/es/ec2/pricing/

Amazon. (3 de Mayo de 2013). *Amazon Elastic Compute Cloud (Amazon EC2)*. Obtenido de Amazon Elastic Compute Cloud (Amazon EC2): http://aws.amazon.com/es/ec2/

Amazon. (3 de Mayo de 2013). *Amazon S3 Multipart Upload*. Obtenido de Amazon S3 Multipart Upload: http://aws.typepad.com/aws/2010/11/amazon-s3-multipart-upload.html

Amazon. (3 de Mayo de 2013). *Amazon Simple Queue Service (Amazon SQS)*. Obtenido de Amazon Simple Queue Service (Amazon SQS): http://aws.amazon.com/es/sqs/

Apache. (2013). *Apache libCloud*. Recuperado el 5 de Marzo de 2013, de http://libcloud.apache.org/

Apache. (3 de Mayo de 2013). *Apache Maven Project*. Obtenido de Apache Maven Project: http://maven.apache.org/

Apache Software Foundation. (3 de Mayo de 2013). *Deltacloud API*. Obtenido de Deltacloud API: http://deltacloud.apache.org/

Araxis. (5 de Mayo de 2013). *Araxis Merge*. Obtenido de http://www.araxis.com/merge/

Armbrust, M., Fox, A., Griffith, R., Joseph, A., Katz, R., Konwinski, A., & Lee, G. (2010). A view of Cloud Computing. *Communications of the ACM, 53*(4), 50-58.

Briscoe, G., & Marinos, A. (2009). Digital ecosystems in the Clouds: towards community Cloud Computing. *In Digital Ecosystems and Technologies. 3rd IEEE International Conference.*, (págs. 103-108).

Clark, J. (3 de Mayo de 2013). *How Amazon exposed its guts: The History of AWS's EC2*. Obtenido de How Amazon exposed its guts: The History of AWS's EC2: http://www.zdnet.com/how-amazon-exposed-its-guts-the-history-of-awss-ec2-3040155310/

Cloudloop. (3 de Mayo de 2013). *Cloudloop*. Obtenido de Cloudloop: https://java.net/projects/cloudloop

Computer World. (3 de Mayo de 2013). *Cloud storage specification gets ISO approval*. Obtenido de Cloud storage specification gets ISO approval: http://www.computerworld.com/s/article/9232598/Cloud_storage_specification_gets_ISO_approval

Dasein. (2013). *Dasein Cloud API*. Recuperado el 5 de Marzo de 2013, de http://dasein-cloud.sourceforge.net/

Dropbox. (3 de Mayo de 2013). *Dropbox*. Obtenido de Dropbox: https://www.dropbox.com/

Eclipse Foundation, Inc. (3 de Mayo de 2013). *The Eclipse Foundation open source community website*. Obtenido de The Eclipse Foundation open source community website: http://www.eclipse.org/

Eucalyptus Systems. (3 de Mayo de 2013). *Eucalyptus*. Obtenido de Eucalyptus: http://www.eucalyptus.com/

Forman, I., & Forman, N. (2005). *Java Reflection in Action.*

Franco. (2013). *El cloud multiplica la competitividad de las empresas*. Recuperado el 12 de Abril de 2013, de http://cloud.ticbeat.com/cloud-multiplica-competitividad-empresas-entrevista/

Google. (2013). *Google App Engine*. Recuperado el 3 de Mayo de 2013, de https://developers.google.com/appengine/

Google. (2013). *Google Cloud Storage*. Recuperado el 3 de Mayo de 2013, de https://developers.google.com/storage/

Heroku. (2013). *Heroku Cloud Application Platform*. Recuperado el 3 de Mayo de 2013, de http://www.heroku.com/

Hewlett-Packard. (2013). *Hewlett-Packard Cloud Services*. Recuperado el 3 de Mayo de 2013, de https://www.hpcloud.com/

Hofmann, P., & Woods, D. (2010). Cloud Computing: The limits of public Clouds for business applications. *Internet Computing, IEEE, 14*(6), 90-93.

IBM. (2013). *IBM Cloud Computing*. Recuperado el 3 de Mayo de 2013, de http://www.ibm.com/cloud-computing/us/en/

InfoWorld. (3 de Mayo de 2013). *IEEE aims to drive cloud computing standards*. Obtenido de IEEE aims to drive cloud computing standards: http://www.infoworld.com/d/cloud-computing/ieee-aims-drive-cloud-computing-standards-712

Javassist. (3 de Mayo de 2013). *Javassist*. Obtenido de Javassist: http://www.csg.ci.i.u-tokyo.ac.jp/~chiba/javassist/

java-storage. (3 de Mayo de 2013). *java-storage*. Obtenido de java-storage: http://code.google.com/p/java-storage/

jClouds. (2013). *jClouds*. Recuperado el 5 de Marzo de 2013, de http://www.jclouds.org/

JetS3t. (3 de Mayo de 2013). *JetS3t*. Obtenido de JetS3t: http://www.jets3t.org/

JFileSync. (5 de Mayo de 2013). *JFileSync*. Obtenido de http://jfilesync.sourceforge.net/

Leavitt, N. (2009). Is cloud computing really ready for prime time. *Growth, 27*(5).

Mandel, M. (2001). *La Depresión de Internet.* Madrid: Pearson Educación.

Mell, P., & Grance, T. (2011). The NIST Definition of Cloud Computing. *NIST Special Publication 800-145*. Obtenido de , , NIST Special Publication 800-145 (September 2011)..

Microsoft. (2013). *Windows Azure*. Recuperado el 3 de Mayo de 2013, de http://www.windowsazure.com/es-es/

Microsoft. (3 de Mayo de 2013). *Windows Azure: Plataforma en la nube de Microsoft*. Obtenido de Windows Azure: Plataforma en la nube de Microsoft: http://www.windowsazure.com/es-es/

MuCommander. (5 de Mayo de 2013). *MuCommander*. Obtenido de http://www.mucommander.com/

neoeedit. (5 de Mayo de 2013). *neoeedit*. Obtenido de https://code.google.com/p/neoeedit/

NIST. (3 de Mayo de 2013). *Two New Publications Provide a Cloud Computing Standards Roadmap and Reference Architecture*. Obtenido de Two New Publications Provide a Cloud Computing Standards Roadmap and Reference Architecture: http://www.nist.gov/itl/csd/cloud-091311.cfm

OpenNebula Project. (3 de Mayo de 2013). *OpenNebula*. Obtenido de OpenNebula: http://opennebula.org/

Oracle. (3 de Mayo de 2013). *ClassLoader (Java Platform SE 7)*. Obtenido de ClassLoader (Java Platform SE 7): http://docs.oracle.com/javase/7/docs/api/java/lang/ClassLoader.html

Oracle. (28 de Septiembre de 2013). *The Reflection API*. Obtenido de The Reflection API: http://docs.oracle.com/javase/tutorial/reflect/index.html

Rackspace. (2013). *Rackspace*. Recuperado el 3 de Mayo de 2013, de http://www.rackspace.com/

Sobel, J., & Friedman, D. (28 de Septiembre de 2013). *An Introduction to Reflection-Oriented Programming.* Obtenido de An Introduction to Reflection-Oriented Programming: http://www.cs.indiana.edu/~jsobel/rop.html

The Eclipse Foundation. (5 de Mayo de 2013). *Mylyn*. Obtenido de http://eclipse.org/mylyn/

TrainSignal. (3 de Mayo de 2013). *CCSK Overview: Certificate of Cloud Security Knowledge*. Obtenido de CCSK Overview: Certificate of Cloud Security Knowledge: http://www.trainsignal.com/blog/ccsk-certificate-of-cloud-security-knowledge

VMWare. (2013). *VMWare Cloud Computing Solutions*. Recuperado el 3 de Mayo de 2013, de http://www.vmware.com/solutions/cloud-computing/index.html

W3C. (3 de Mayo de 2013). *Extensible Markup Language (XML)*. Obtenido de Extensible Markup Language (XML): http://www.w3.org/XML/

YourKit. (5 de Mayo de 2013). *YourKit Java Profiler*. Obtenido de http://www.yourkit.com/