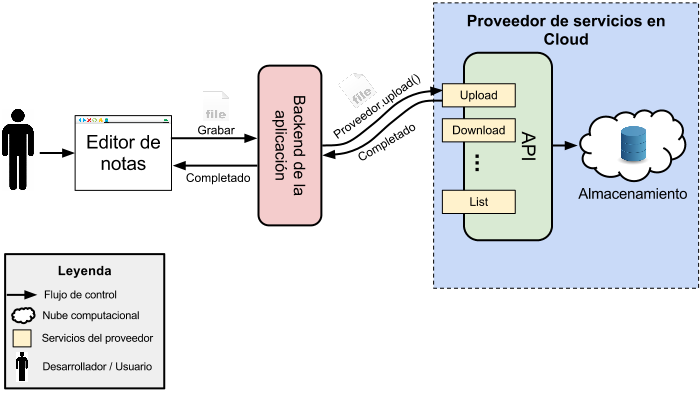
## 3. Enfoque

Durante el análisis de las herramientas actuales se logró detectar una serie de problemas o puntos no tenidos en cuenta que son de sumo interés para usuarios de este tipo de plataformas. Los problemas principales que se encontraron se relacionan fundamentalmente con las capacidades de migración de las herramientas. 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 el otro lado se nos presenta 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. 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.

Llevaremos esto a un caso concreto suponiendo la existencia de una aplicación denominada “Cloud Recorder” que permite elaborar y almacenar notas periodísticas. Esta aplicación presenta una interfaz gráfica sencilla en la que un usuario puede cargar archivos de audio o texto, añadir comentarios y persistirlos de manera segura. Al momento de persistir las notas, la interfaz gráfica hace uso de un método “Grabar” provisto por el backend de la aplicación. Para mantener un backup off-site de las notas los diseñadores de la aplicación decidieron utilizar un servicio de almacenamiento en cloud. Luego de realizar un relevamiento de la oferta de distintos proveedores, se selecciona a uno de ellos y se procede a integrar su API resultando en un flujo similar al presentado en la Figura 3.1.



​Figura 3.1Vista general de Cloud Recorder utilizando un proveedor especifico.

Este enfoque presenta un problema claro, el backend de la aplicación queda atado a los métodos de la API del proveedor que haya elegido. Si por alguna razón se requiere cambiar el proveedor de servicios de almacenamiento (costos, performance, disponibilidad del servicio, etc.) los desarrolladores se verían obligados a re codificar el backend de la aplicación.

Para darle solución a la problemática planteada se introduce el concepto de “reflexión” o “reflexión computacional”. 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.

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 es posible detectar la utilización de un método en particular y, en tiempo de ejecución, modificar su comportamiento para que ejecute una versión modificada de su código. Una consecuencia no tan visible de utilizar este tipo de mecanismo es que se permite generar un mecanismo que afecte muy poco, o incluso nada, al código del usuario.

En nuestro contexto, este tipo de mecanismos es muy útil para plantear una plataforma que permita migrar entre proveedores del mismo tipo de servicio manteniendo bajos los cambios en el código. Supongamos tener una aplicación ya codificada contra el SDK de un proveedor en particular X. Utilizando mecanismos de reflexión Aether puede capturar la ejecución de los métodos útiles (subir, bajar, agregar metadatos, etc) y modificar su código en tiempo de ejecución para que invoque métodos propios. Un esquema conceptual de este enfoque puede observarse en la Figura 3.2**.**

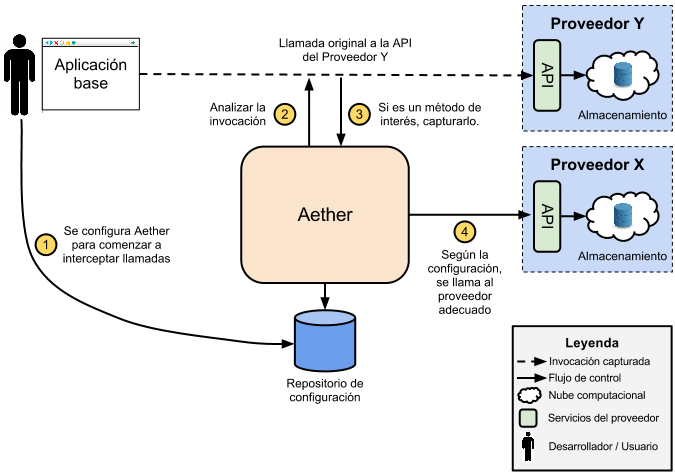


Figura 3.2 Vista general del uso de Aether.

Como se puede apreciar, en un primer momento la aplicación base utiliza directamente el API de un proveedor específico. Inmediatamente surge un problema ya que existe un gran número de proveedores de servicios y migraciones entre estos no son triviales. Factores tales como costos, performance o disponibilidad pueden hacer que se decida migrar la aplicación, agregar otro proveedor como backup, balancearlos según ubicación geográfica, etc. En este contexto el usuario puede tomar provecho de Aether para hacer que su código funcione con otros proveedores sin mayor esfuerzo.

El primer paso para utilizar Aether consiste en introducir la configuración necesaria. Esta configuración incluye los siguientes ítems:

* Datos de configuración del servicio al que se quieren traducir las llamadas. Esto pueden ser simples credenciales para algunos proveedores o llegar a datos más complejos como claves para encriptación.
* El protocolo de traducción de métodos a utilizar. Por esta vía se definen los métodos a interceptar y como traducirlos. El concepto de protocolo en el contexto de Aether se desarrolla en el capítulo 3.2.

Una vez configurado, Aether comienza a inspeccionar las invocaciones generadas por la aplicación al proveedor objetivo. En caso de encontrar un método de interés para Aether (definido en el protocolo configurado), se captura y se procesa para transformar su invocación según se requiera. En el caso de encontrar una invocación que Aether no sepa o no deba manejar se seguirá el curso normal de la aplicación.

Internamente Aether puede verse como muestra la figura 3.3.

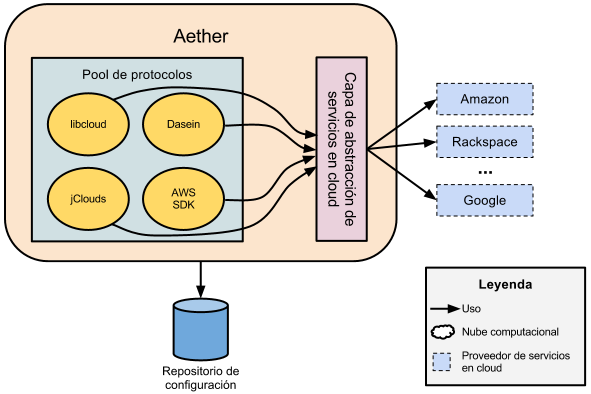


Figura 3.3 Vista interna de Aether.

Como se puede apreciar, el modelo de Aether consta de una serie de protocolos, cada uno de los cuales maneja las traducciones desde un modelo particular, y 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 y 3.2.

### 3.1. Capa de abstracción de servicios en cloud

Esta unidad de Aether tiene como función proveer de un modo de acceso uniforme a servicios de un mismo tipo de distintos proveedores. La idea es que cada servicio 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 encargara de la invocación final al proveedor deseado haciendo las traducciones que sean pertinentes. Esto puede verse en la figura 3.3.

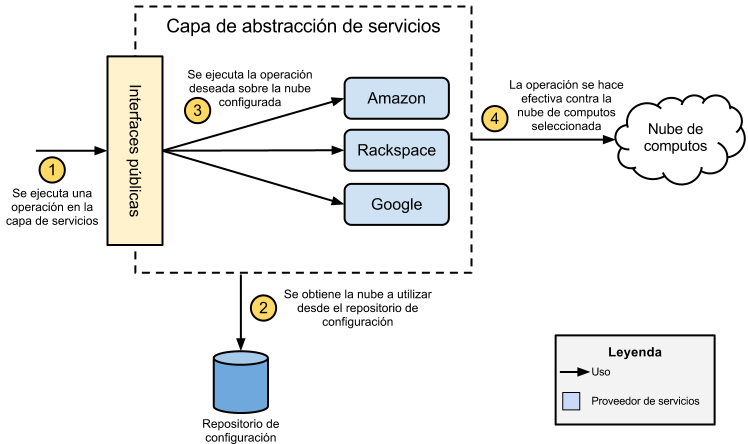


Figura 3.3 Funcionamiento de la capa de abstracción de servicios.

Como se puede apreciar, descomponemos a la capa de abstracción de servicios en dos unidades, las interfaces públicas y las implementaciones 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 de proveedor seleccionada. 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.

### 3.2. 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.

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 3.4.

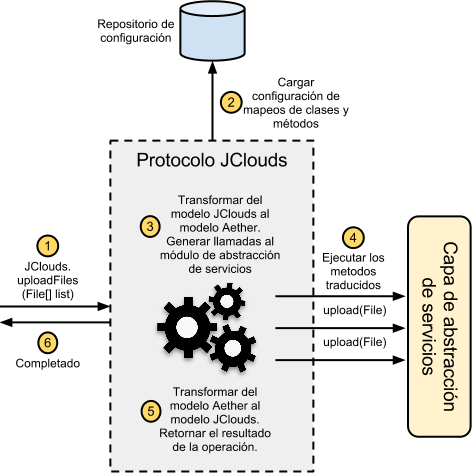


Figura 3.4 Funcionamiento de un protocolo para jClouds.

El funcionamiento de este protocolo es claro. Una vez detectada una invocación al método de interés (uploadFiles en el ejemplo) se debe transformar la invocación original en un set equivalente de invocaciones a la capa de abstracción de servicios. Esto no solo incluye mapear métodos uno a uno sino que también puede incluir transformaciones de parámetros y otras operaciones más complejas. Cabe destacar que todo esto se realiza por medio de mecanismos de reflexión que permiten modificar estructuras de bajo nivel (métodos, objetos, etc) en tiempo de ejecución. Siguiendo la línea de ejecución, ya contamos con el método traducido por lo que solo resta ejecutarlo contra la capa de abstracción de servicios. 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 para luego retornar el control al llamador original. 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.

### 3.3 Conclusiones

En este capítulo se describió el esquema propuesto con el que Aether logra mitigar los problemas de migración existentes en las herramientas actuales. La idea consiste en generar un framework que permita la migración entre proveedores de servicios en cloud de aplicaciones ya codificadas. Para este fin se planteó el uso de dos unidades funcionales dentro de Aether, 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, que no se vio modificado.