Escuela Técnica Superior de Ingeniería Informática.

Ingeniero en Informática.

Framework para la creación de infraestructura para provisión de almacenamiento en la nube como un servicio.

Realizado por:
José Ramón Muñoz Pekkarinen.

Dirigido por:
José Antonio Pérez Castellanos.

Departamento de lenguajes y sistemas Informáticos.

Sevilla, 20 de Noviembre de 2012.

Índice

1 Introducción.

Swiftness es el comienzo del desarrollo de una nueva infraestructura libre, con objeto de proporcionar almacenamiento en la nube como un servicio. Este sistema de almacenamiento puede ser alojado en una nube privada de almacenamiento. El usuario podrá decidir el coste que desea invertir en almacenamiento, pudiendo construir desde nubes de bajo coste, para pequeñas empresas, a nubes de alto coste, disponibilidad y redundancia para grandes corporaciones. Al tratarse de una infraestructura libre, es posible elegir el método de implantación, pudiéndose optar por implantaciones internas, de bajo coste, o contratando los servicios de implantación externos, especialistas en la infraestructura. Estas opciones dotan de flexibilidad al proyecto desarrollado, lo que permite abarcar mercados muy dispares, mejorando la calidad del servicio ofrecido.

La infraestructura deberá estar conformada por una solución de almacenamiento en la nube, de entre las ya existentes, para lo que se ofrecerá un estudio de diversas soluciones, y una comparación por diferentes criterios que nos permita realizar una elección acorde al objetivo perseguido. Posteriormente se estudiarán las plataformas que conforman los clientes de la infraestructura de almacenamiento en la nube, para diseñar y desarrollar sistemas que faciliten la labor de implantación, administración, y uso de esta. Esto conllevará un amplio estudio de los sistemas operativos más comunes en los escritorios y estaciones de trabajo. En concreto, Microsoft Windows y GNU/Linux, con objeto de conocer qué infraestructuras básicas ofrecen, para que el soporte de la infraestructura de almacenamiento se ofrezca desde el mismo sistema operativo, permitiendo al cliente conocer los menos detalles posibles acerca del almacenamiento, sin perder la eficiencia de disponer de un cliente lo más liviano posible. En este sentido adquiere gran importancia la recopilación de información acerca de las infraestructuras ofrecidas por los sistemas operativos, puesto que supondrá una pieza básica para el posterior diseño y desarrollo de los drivers que ofrecerán el soporte deseado y delimitará los objetivos de este proyecto, sin dejar de ofrecer campos de estudio para futuros ciclos de desarrollo que permitan ampliar sus funcionalidades, y mejorar las capacidades ofrecidas en el ciclo inicial. Por tanto, se recogerán estudios, de las infraestructuras de los sistemas operativos, que ofrezcan servicios que pudieran ser no utilizados en este diseño inicial.

Complementaremos este estudio de plataformas con el desarrollo de un driver para el sistema operativo GNU/Linux que sirva como base y ejemplo de los objetivos planteado por el diseño de esta plataforma. Este driver ofrecerá un soporte básico para utilizar la plataforma de almacenamiento, permitiendo que ésta se observe en el cliente como si de un dispositivo local se tratara. Para ello se le dotará de capacidades de conexión con la plataforma, y resolución de operaciones básicas de entrada/salida, es decir, lecturas y escrituras, que el sistema de ficheros realizará al dispositivo. Por tanto, no es objetivo de este proyecto diseñar el sistema de ficheros, ofreciendo la flexibilidad al usuario de utilizar el que desee. Sin embargo, sí es uno de nuestros objetivos traducir las peticiones realizadas por el sistema de ficheros, a las comunicaciones de red requeridas por el servidor para que éste devuelva la respuesta requerida, y, posteriormente, traducir las respuestas recibidas para que el sistema de ficheros entienda qué es lo que el servidor ofrece.

2 Planificación.

Con el objetivo de garantizar la calidad del servicio ofrecido con el presente proyecto, realizaremos una estimación del tiempo y esfuerzo requerido para llevar a cabo las distintas tareas que lo componen. Cada tarea recibirá una estimación inicial, previa a la realización del objetivo, y una estimación final que se corresponde con el tiempo utilizado para realizar la tarea durante el ejercicio del proyecto. Se calculará el error relativo de dichas estimaciones para contrastar si dichas estimaciones se ajustan a lo que realmente se necesitó.

La distintas estimaciones se recogen en la siguiente tabla, en la que se estima que trabaja una persona, 8 horas al día:

Tarea	Estimación Inicial	Estimación Final	Error Relativo
Introducción.	1 día	1 día	2.12%
Planificación	1 día	1 día	2.12%
Documentación de nubes de almacenamiento.	7 días	10 días	39.83%
Documentación de S.O. GNU/Linux.	20 días	18 días	-11.91%
Documentación de S.O. Windows.	30 días	27 días	-11.91%
Documentación de requisitos.	7 días	7 días	2.12%
Documentación de análisis.	7 días	9 días	2.12%
Documentación de diseño.	7 días	6 días	-16.10%
Implementación.	7 días	10 días	39.81%
Pruebas.	5 días	5 días	2.12%
TOTAL	92	94	2.12%
Error relativo medio			5.03%

2.1 Estimación del coste.

Para determinar el coste del proyecto que nos concierne, acudiremos al uso de la conocida métrica del número de lineas de código (LDC). Ésta métrica propone el coste de un proyecto

basándose en su tamaño, y es de las métricas más utilizadas en la ingeniería del software. Dentro de ella se entienden por lineas de código, las lineas útiles, quedando excluidas las lineas en blanco y los comentarios. Su clasificación se recoge en la siguiente tabla:

Categoría	Programadores	Duración	LDC	Ejemplo
Trivial	1	0-4 semanas	< 1K	Utilidad de ordenación
Pequeño	1	1-6 meses	1K -3K	Biblioteca de funciones
Medio	2-5	0.5-2 años	3K-50K	Compilador de C
Grande	5-20	2-3 años	50K-100K	S.O. pequeño
Muy grande	100-1000	4-5 años	100K-1M	S.O. Grande
Gigante	1000-5000	5-10 años	>1M	Sistema de distribución

La aplicación que se provee con dicho proyecto se aproxima a las 1000 líneas de código, luego, se encuentra en la clasificación de un proyecto pequeño. Esto se debe a que, como comentábamos anteriormente, una parte importante dentro de éste es la realización del estudio detallado de las nubes de almacenamiento y los sistemas operativos en los que se ofrecerá el soporte de dicha nube.

2.2 Método COCOMO.

Para tener una estimación más elaborada del coste del proyecto, se usará el método COCOMO (Constructive Cost Model). Esta metodología de la ingeniería del software hace uso de 3 modelos distintos: básico, medio y avanzado. El modelo se selecciona en función del número de variables que definen nuestro problema, siendo el modelo básico el que menos variables utiliza, y el modelo avanzado el que más. Es posible aplicarlo a los siguientes tres tipos de software:

- **Orgánico:** son proyectos con un tamaño menor a 50000 líneas de código, en los que el equipo de desarrollo tiene experiencia, y se utiliza un entorno estable de desarrollo.
- **Semiacoplado:** proyectos intermedios en complejidad y tamaño, en los que la experiencia del equipo es variable, y sus restricciones son de complejidad intermedia.
- **Empotrado:** se tratan de proyectos complejos, en los que la experiencia aportada es poca o nula, en un entorno de innovación técnica. Sus requisitos son muy restrictivos y de gran volatilidad.

Método COCOMO.

Para nuestro modelo, ya que estimaremos el coste haciendo uso únicamente del tamaño en líneas de código, el modelo básico nos ofrecerá una buena aproximación del coste de nuestro proyecto.

El tipo de proyecto, dado que la variable en que nos apoyamos es el número de líneas de código, se corresponde con un sistema orgánico, y será considerado como tal. Sin embargo, conviene recordar que se trata de un proyecto con requisitos muy restrictivos, y en los que el equipo de desarrollo carece de experiencia alguna.

La ecuación del esfuerzo es la siguiente:

$$E = a\,KLDC^{b}$$ personas/mes.

KLDC se corresponde con el número de líneas de código en millares.

La ecuación del tiempo de desarrollo es la siguiente:

$$T = c\,E\,d$$

Los coeficientes a, b, c y d, se hallan de forma empírica, y sus valores quedan recogidos en la siguiente tabla:

Proyecto	A	B	C	D
Orgánico	2.4	1.05	2.5	0.38
Semiacoplado	3	1.12	2.5	0.35
Empotrado	3.6	1.2	2.5	0.32

El esfuerzo resultante es $E = 2.4 * 1^{1.05} = 2.4\ personas/mes$

Su tiempo sería: $T = 2.5 * 2.4^{0.38} = 3.486\ meses$

De estos cálculos puede deducirse el número de personas requerido para realizar el proyecto, siendo este $E/T = 2.4/3.486 = 0.68$. Luego el proyecto es realizable por una persona en un tiempo aproximado de tres meses y medio.

3 Documentación previa.

3.1 Cloud Storage.

El Cloud Storage, o almacenamiento en la nube, es un modelo de almacenamiento basado en redes en el que los datos son almacenados en piscinas virtuales de almacenamiento.

En las arquitecturas de almacenamiento en la nube, se persiguen cuatro objetivos, la agilidad, o capacidad de mover los datos al sitio adecuado para mejorar su disponibilidad, la escalabilidad, o capacidad de manejar crecientes cargas de trabajo, o de adaptarse para ser capaces de soportarlas, la elasticidad, o la capacidad de escalarse sin limites razonables, y la capacidad de ser multiusuario.

El Cloud Storage debe estar formado por servicios distribuidos que se comporten como un conjunto, debe ser tolerante a fallos a través de la redundancia de datos, ser duradero, sirviéndose de la creación de versiones de los documentos alojados, y, finalmente, debe ser consistente entre las diferentes versiones almacenadas.

Las principales preocupaciones que se plantean en estos modelos de almacenamiento son la seguridad ante posibles accesos indebidos, la estabilidad de la empresa proveedora, la accesibilidad de la información, y los costes hardware de la solución utilizada.

Actualmente, existen dos tipos de instalaciones de Cloud Storage, las realizadas en servidores externos, y las instalaciones locales que el administrador puede acomodar a sus necesidades.

En servidores externos, la compañías que ofrecen este servicio establecen una piscina de servidores virtuales en los que se aloja dicho almacenamiento y los pone a disposición del usuario. Este tipo de instalaciones goza de las mayores ventajas del Cloud Storage, que se apoyan en la disposición de grandes servidores de almacenamiento, y la delegación de la tarea de la administración y responsabilidades a la compañía encargada de ofrecer el servicio.

En las instalaciones locales, el usuario debe instalar la plataforma de almacenamiento en la nube, adaptarla a sus necesidades, y administrarla adecuadamente con el fin que persigue. En el proyecto que nos ocupa estudiaremos en mayor profundidad las soluciones locales.

3.2 Almacenamiento en soluciones libres y locales existentes.

3.2.1 Amazon S3.

La plataforma Amazon S3 no se trata expresamente de una plataforma libre, ni de instalación local, puesto que sólo se ofrecen las distintas interfaces de comunicación de esta. Aun así es necesario estudiarla, debido a que algunas soluciones basan su almacenamiento en esta plataforma.

La Amazon Simple Storage Service tiene como objetivo ser escalable, tener alta disponibilidad y baja latencia, todo ello a un bajo coste. Dispone de tres tipos de interfaces distintas,

REST (Representational State Transfer), SOAP (Simple Object Access Protocol) y BitTorrent. La primera se trata de una interfaz para sistemas distribuidos, la más adoptada en la World Wide Web. La segunda es una interfaz que cumple el mismo propósito que REST, pero basándose en tecnología XML. La tercera es una de las conocidas tecnologías P2P existentes.

Admite objetos de un tamaño inferior a 5 terabytes de información y los organiza en cubos. Para cada cubo, y objeto, dentro de éste, se proporciona una lista de control de acceso que permite al usuario disponer de su información. A cada objeto en un cubo puede accediderse directamente desde internet. Además, cada objeto es una semilla de la red BitTorrent que puede ser utilizada para reducir el ancho de banda utilizado durante la descarga.

Puesto que el principal uso de los cubos es almacenar páginas web, cada cubo puede ser configurado para almacenar los registros de acceso en un cubo gemelo, o réplica.

Esta solución es capaz de alojar maquinas virtuales que se pueden ejecutar a través de sus interfaces expuestas, y admite redundancia distribuida en su cluster de almacenamiento.

Debido a su condición de no tener instalación local, no es posible saber sobre que tipo de host funciona dicha solución.

3.2.2 OpenNebula.

Aunque la plataforma de Cloud Computing de OpenNebula es compatible con la interfaz Amazon EC2 Query Interface, lo que le ofrece a su vez compatibilidad con la plataforma de almacenamiento Amazon S3, dicha plataforma goza de un subsistema de almacenamiento propio.

El subsistema está ideado para almacenar máquinas virtuales únicamente. Dicha plataforma se compone de una serie de drivers divididos en dos tipos, data store drivers, o drivers de almacenamiento de datos, y transfer manager drivers, o drivers de gestión de transferencia. Los primeros se encargan de ofrecer las funcionalidades de creación, eliminación y modificación de las máquinas virtuales. Los segundos se encargan de realizar las transferencias a otras localizaciones, clonación y enlace.

Los drivers de transferencia proveen a esta solución de una interfaz REST, y no es posible separar los subsistemas de comunicación de los de gestión de almacenamiento y operación de esta nube. No se especifica que disponga de persistencia de datos replicándolos a lo largo del cluster formado por la nube. Los tipos de host donde pueden ser alojada esta solución deben ser servidores Linux.

3.2.3 Nimbus.

La plataforma Nimbus utiliza una implementación propia y libre de la API REST (Representational State Transfer) de Amazon S3, llamada Cumulus, para su almacenamiento. Las interfaces que ofrece dicha implementación son de tipo REST y SOAP, y no es posible separar la interfaz de comunicación del almacenamiento subyacente. Su almacenamiento, al igual que OpenNebula está basado en máquinas virtuales que gestionan sus propios discos, no siendo posible definir un almacenamiento sin sistema operativo. Esta solución permite, sin embargo, realizar replicación distribuida de datos para garantizar la persistencia. Los host que soportan esta solución deben ser servidores Linux.

3.2.4 OpenStack.

Dicha solución se divide en tres partes bien diferenciadas para ofrecer una nube personal. Éstas son Nova, para la ejecución de software, Swift, para el almacenamiento, y Glance, para la gestión de imágenes virtuales. Esto le ofrece la capacidad de poder instalar sólo el componente necesario para las necesidades del usuario. Swift es capaz de almacenar objetos o bloques indistintamente, permitiendo no sólo el almacenamiento de imágenes virtuales, sino de cualquier tipo de información.

Esta solución es escalable, distribuida, admite redundancia de datos, y propone al usuario una API de programación. Dicha API es compatible con las soluciones de almacenamiento masivo NetApp, Nexenta, SolidFire y Amazon S3. Debido a esto, admite comunicaciones a través de una interfaz REST o SOAP. El soporte de bloques permite que cualquier dispositivo de bloques(por ejemplo un disco duro, una cinta magnética, o una imagen virtual) pueda añadirse al sistema de almacenamiento distribuido de forma transparente.

Su organización se basa en clústeres de almacenamiento, y cada objeto se distribuye entre diferentes nodos de la red. El sistema garantiza la persistencia de bloques y capacidad de realizar instantáneas(snapshots) del almacenamiento para labores de salvaguardas.

Una vez más el tipo de servidor que es capaz de alojarlo es un servidor Linux.

3.2.5 OpenQRM.

En el caso de OpenQRM nos encontramos con una solución propia, basada en el sistema de plugins para la plataforma. Dichos plugins se encargan de dar soporte por separado a los diversos tipos de almacenamientos reales en las distintas máquinas. Para ello, proveen al usuario de una interfaz SOAP. Su tipo de almacenamiento es basado en máquinas virtuales y no admite replicación de datos a lo largo del cluster.

Entre ellos podemos encontrarnos los siguientes plugins, que nos dan una idea del tipo de almacenamiento tratado: netapp-storage, nfs-storage, local-storage, iscsi-storage, lvm-storage, xen-storage y kvm-storage.

Dicha solución debe ser instalada sobre un servidor Linux.

3.2.6 Eucalyptus.

Entre los distintos componentes de esta solución de Cloud Computing se encuentra el Storage Controller, o controlador de almacenamiento. Dicho componente implementa el sistema de almacenamiento Amazon EBS, Elastic Block Store, el cual provee de dispositivos de bloques brutos a la plataforma, siguiendo la estructura detallada en la plataforma Amazon S3. Esta implementación le permite utilizar las interfaces REST, SOAP y Bittorrent para manejar el almacenamiento, y en este caso es posible separar el subsistema de comunicación del de almacenamiento en servidores distintos.

Los hosts que soportan dicha solución deben ser servidores Linux.

3.2.7 CloudStack.

Esta plataforma se nutre de la solución de almacenamiento de OpenStack, Swift, estudiada anteriormente. Por tanto, ofrece interfaces REST y SOAP para su comunicación, cualquier tipo de almacenamiento, almacenamiento distribuido a lo largo del cluster, y, a diferencia de OpenStack, es capaz de instalarse sin sistema operativo servidor.

3.2.8 Abiquo Open Source Edition.

En esta plataforma nos encontramos con que el almacenamiento es independiente al resto de la plataforma, ofreciendo solo almacenamiento virtual de dos tipos: asistido, o genérico iSCSI. En el caso del almacenamiento asistido la plataforma ofrece almacenamiento como servicio directamente al usuario. En el caso del almacenamiento genérico iSCSI, el usuario dispone de un almacenamiento preconfigurado por el administrador del sistema, habitualmente una máquina virtual. Para ello se apoya en un conjunto de plugins, al igual que OpenQRM, que dan soporte a los siguientes tipos de almacenamiento: LVM y iSCSI Linux, Nexenta, NetApp. Por otra parte se ofrece una API de desarrollo para facilitar el soporte de otras plataformas como Dell Equallogic, IBM Volume Manager.

Esta solución solo ofrece una interfaz REST al usuario, y no ofrece replicación distribuida de forma automática. En este caso no se requiere de ningún sistema operativo para alojar la nube.

3.3 Comparación de soluciones.

La mayor parte de las soluciones estudiadas implementan al menos alguna de las interfaces de comunicación propuestas por la nube de Amazon, además de compartir su tipo de almacenamiento, y, al menos, ofrecen al usuario de almacenamiento basado en máquinas virtuales. Sin embargo, no todas ofrecen la oportunidad de realizar una instalación minimalista orientada al almacenamiento. Exceptuando la nube de Abiquo, y por razones obvias, la solución de Amazon, todas deben ser alojadas en servidores Linux, no soportándose la instalación nativa sobre un servidor, o la instalación sobre servidores Windows.

De este amplio abanico de soluciones es necesario destacar el soporte de las tres interfaces habituales por parte de Eucalyptus, y su soporte para distintos tipos de almacenamiento, admitiendo flexibilidad de instalación. Sin embargo, esta poderosa nube está orientada a ofrecer todos los servicios disponibles en cualquier nube habitual(computación, virtualización y almacenamiento), y no es especialmente cómoda para realizar una instalación minimalista, orientada al almacenamiento.

La solución Nimbus, a través de su servidor Cumulus, ofrece una imagen de sencillez si el objetivo perseguido es configurar una nube de almacenamiento, sin embargo su fuerte orientación a requerir máquinas virtuales con sistemas operativos propios hacen que la solución pierda atractivo. OpenNebula y OpenQRM, disponen de características similares, conformando las soluciones más

rígidas de todas, pero a la vez las más sencillas de administrar. Además, su implementación basada en drivers o plugins asegura facilidad de desarrollo de código y alta modularidad.

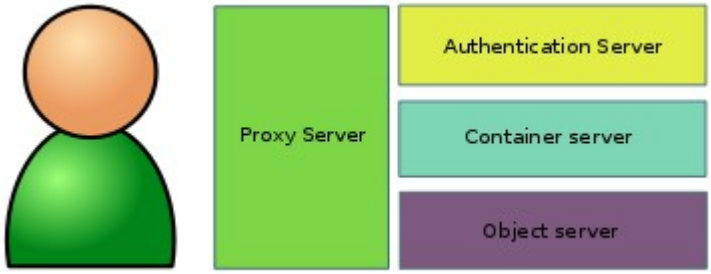
Las soluciones Openstack y Cloudstack ofrecen alta flexibilidad de instalación, una modularización aceptable, y por tanto, facilidad de mantenimiento de código, así como la posibilidad de hacer instalaciones minimalistas orientadas a cada uno de sus servicios por separado. Esta última característica las convierten en plataformas adecuadas para cualquier tipo de desarrollo sobre cualquiera de sus servicios.

Las características de las distintas plataformas se resumen en la siguiente tabla:

Característica / Nube		Amazon S3	OpenNebula	Nimbus	OpenStack	OpenQRM	Eucalyptus	Cloudstack	Abiquo O.S.E.
APIs soportadas	REST	Sí	Sí	Sí	Sí	No	Sí	Sí	Sí
	SOAP	Sí	No	Sí	Sí	Sí	Sí	Sí	No
	Bittorrent	Sí	No	No	No	No	Sí	No	No
Comunicación independiente de almacenamiento		No	Sí	No	Sí	No	Sí	Sí	Sí
Tipo almacenamiento	M.V.	Sí	Sí	Sí	Sí	Sí	Sí	Sí	Sí
	Puro	Sí	No	No	Sí	No	Sí	Sí	Sí
Cluster de replicación		Sí	No	Sí	Sí	No	Sí	Sí	No
Host	No requerido	?	No	No	No	No	No	Sí	Sí
	Linux	?	Sí	Sí	Sí	Sí	Sí	Sí	No
	Windows	?	No	No	No	No	No	No	No

3.4 Swift.

Este componente de la nube propuesta por Openstack está formado por cuatro servidores que se comunican para ofrecer el servicio propuesto. Éstos son, un servidor de cuentas, un servidor de contenedores, un servidor de objetos y un servidor proxy para distribuir las peticiones entre el resto de servidores. De esta forma, para realizar cualquier transacción, el cliente deberá solicitar, al servidor de cuentas, acceso a la plataforma. Validado su acceso, dispondrá de acceso al contenedor asociado a su cuenta, y a los objetos asociados a este contenedor, cada uno de ellos gestionado por su servidor correspondiente. Para realizar cualquier petición, el cliente debe comunicarse con el servidor proxy, que ejerce de fachada aislando al resto de servidores. La organización de éstos sería la siguiente:



Para el objetivo de este proyecto, todos los servidores pueden ser tratados como una caja negra que se dedica al almacenamiento y distribución de datos, excepto el servidor proxy, cuya API estudiaremos en mayor profundidad.

3.4.1 Swift Proxy Server.

La comunicación con este componente, se basa en el envío de peticiones HTTP. Dichas peticiones se gestionan mediante el uso de la operación GET, y deben contener unas cabeceras específicas para reconocer los distintos tipos de peticiones. Éstas disponen de una serie de restricciones detalladas a continuación:

- El numero máximo de cabeceras por petición es de 90.
- La máxima longitud de las cabeceras es 4096 bytes.
- Cada línea de petición no debe tener más de 8192 bytes.
- Una petición no debe exceder el tamaño de 5 gigabytes.
- El nombre de un contenedor no puede ser mayor de 256 bytes.
- El nombre de un objeto no puede ser mayor que 1024 bytes.

La operaciones posibles sobre el almacenamiento se dividen entre los tres servidores de gestión de éste. Detallamos a continuación las operaciones relevantes:

- **Servidor de cuentas.**
  - Listado de contenedores.
- **Servidor de contenedores.**
  - Listado de objetos en el contenedor.
  - Creación de contenedores.
  - Eliminación de contenedores.
- **Servidor de objetos.**
  - Recuperación de objetos.
  - Creación/actualización de objetos.
  - Copia de objetos.

- Eliminación de objetos.

3.4.2 Operaciones en la API.

Como ya adelantábamos anteriormente, para interactuar con el servidor proxy se utiliza una interfaz REST, es decir, el servidor espera peticiones HTTP para servir cualquier tipo de fichero disponible, y realizar el control de acceso al almacenamiento.

3.4.2.1 Identificación.

Esta operación es, por razones de seguridad, la primera a realizar de todas las posibles. Para ello hace falta proveer al servidor de un usuario válido y su contraseña a través de la operación GET de la versión 1.1 de HTTP. El servidor nos devolverá un identificador único necesario para realizar cualquier posterior operación. La solicitud es la siguiente:

```
GET /<api version> HTTP/1.1
Host: <Servidor proxy>
X-Auth-User: <usuario>
X-Auth-Key: <contraseña>
```

La respuesta tiene la siguiente forma:

```
HTTP/1.1 204 No Content

Date: <fecha>
Server: <servidor web>
X-Storage-Url: <dirección para posteriores accesos>
X-Auth-Token: <identificador para posteriores accesos>
Content-Length: 0
Content-Type: text/plain; charset=UTF-8
```

3.4.2.2 Listado de contenedores.

La operación de listado de contenedores no requiere ningún parámetro en especial, más que el identificador de sesión obtenido durante la identificación. Sin embargo, existen un par de variables de configuración de interés general para dicha operación. Las variables se concatenan a la dirección solicitada en la operación GET, separando el listado de variables de la dirección por un interrogante('?'). Dichas variables sirven para describir el formato en que se recibirá el listado(siendo válidos los valores format=json y format=xml), el límite de contenedores que se quiere recibir, configurando la variable limit=N, y la coincidencia desde la cual se quiere comenzar el listado con la variable marker.

```
GET /<api version>/<account>?var1=val1& … & varN=valN HTTP/1.1
Host: <dirección del servidor>
X-Auth-Token: <identificador de sesión>
```

Un ejemplo de respuesta en formato JSON es el siguiente:

```
HTTP/1.1 200 OK

Date: Tue, 25 Nov 2008 19:39:13 GMT
Server: Apache
Content-Type: application/json; charset=utf-8

[

  {"name":"test_container_1", "count":2, "bytes":78},
  {"name":"test_container_2", "count":1, "bytes":17}
]
```

En formato XML la respuesta sería de la siguiente forma:

```
  HTTP/1.1 200 OK

Date: Tue, 25 Nov 2008 19:42:35 GMT
Server: Apache
Content-Type: application/xml; charset=utf-8

<?xml version="1.0" encoding="UTF-8"?>


<account name="MichaelBarton">
  <container>
    <name>test_container_1</name>
    <count>2</count>
    <bytes>78</bytes>
  </container>
  <container>
    <name>test_container_2</name>
    <count>1</count>
    <bytes>17</bytes>
  </container>
</account>
```

3.4.2.3 Listado de objetos.

En el caso de los listados de objetos, se dispone de algunas variables de configuración adicionales, además de las ya descritas para los listados de contenedores. Estas nuevas variables son: end_marker, para devolver un listado terminado en el valor especificado, prefix, para ofrecer un prefijo que preceda al nombre del objeto y delimiter, para mostrar las ocurrencias listadas hasta el delimitador, desechando el resto de la dirección. Un ejemplo de uso de delimitador es aquél en el que sólo queremos listar los directorios existentes en la raíz del almacenamiento, para lo cual solamente requerimos utilizar el delimitador '/' para obtener el resultado deseado.

El formato de la petición es el siguiente:

```
GET /<api version>/<account>/<container>[?parm=value] HTTP/1.1
```

```
Host: <servidor>
X-Auth-Token: <identificador de sesión>
```

Un ejemplo de respuesta en formato JSON:

```
HTTP/1.1 200 OK
Date: Tue, 25 Nov 2008 19:39:13 GMT
Server: Apache
Content-Length: 387
Content-Type: application/json; charset=utf-8

[
  {"name":"test_obj_1",
   "hash":"4281c348eaf83e70ddce0e07221c3d28",
   "bytes":14,
   "content_type":"application\/octet-stream",
    "last_modified":"2009-02-03T05:26:32.612278"},
  {"name":"test_obj_2",
   "hash":"b039efe731ad111bc1b0ef221c3849d0",
   "bytes":64,
   "content_type":"application\/octet-stream",
   "last_modified":"2009-02-03T05:26:32.612278"},
]
```

En formato XML el servidor respondería lo siguiente:

```
HTTP/1.1 200 OK

Date: Tue, 25 Nov 2008 19:42:35 GMT
Server: Apache
Content-Length: 643
Content-Type: application/xml; charset=utf-8

<?xml version="1.0" encoding="UTF-8"?>


<container name="test_container_1">
  <object>
    <name>test_object_1</name>
    <hash>4281c348eaf83e70ddce0e07221c3d28</hash>
    <bytes>14</bytes>
    <content_type>application/octet-stream</content_type>
    <last_modified>2009-02-03T05:26:32.612278</last_modified>
  </object>
  <object>
    <name>test_object_2</name>
    <hash>b039efe731ad111bc1b0ef221c3849d0</hash>
    <bytes>64</bytes>
    <content_type>application/octet-stream</content_type>
    <last_modified>2009-02-03T05:26:32.612278</last_modified>
  </object>
</container>
```

3.4.2.4 Creación de contenedores.

Los contenedores, no son más que compartimentos en los que almacenar los objetos. Sin embargo, sus nombres deben cumplir las restricciones de no tener una longitud mayor de 256 caracteres, y no contener el carácter '/'. Un ejemplo de petición de creación de un contenedor es el siguiente:

```
PUT /<api version>/<account>/<container> HTTP/1.1
Host: <servidor>
X-Auth-Token: <identificador de sesión>
```

La respuesta en este caso:

```
HTTP/1.1 201 Created
Date: Thu, 07 Jun 2010 18:50:19 GMT
Server: Apache
Content-Type: text/plain; charset=UTF-8
```

3.4.2.5 Eliminación de contenedores.

Para la eliminación de un contenedor, es suficiente con la siguiente petición:

```
DELETE /<api version>/<account>/<container> HTTP/1.1
Host: <server>
X-Auth-Token: <identificador de sesión>
```

Sin embargo debe suceder que el contenedor esté vacío para que éste pueda ser eliminado. Su borrado es permanente. Su respuesta pudiera ser la siguiente:

```
HTTP/1.1 204 No Content

Date: Thu, 07 Jun 2010 18:57:07 GMT
Server: Apache
Content-Length: 0
Content-Type: text/plain; charset=UTF-8
```

3.4.2.6 Recuperación de objetos.

Aunque habitualmente se utiliza la operación GET para recuperar objetos del almacenamiento masivo, es posible utilizar las directivas If-Match, If-None-Match, If-Modified-Since e If-Unmodified-Since definidas en el protocolo RFC2616. Como sus nombres indican, su funcionalidad es hacer condicional la recuperación del objeto en cuestión, definiendo si se encuentra un objeto que coincida en nombre, si no lo hay, o si ha sido modificado desde una fecha determinada.

Existe también un soporte básico para recuperar rangos de memoria dentro del objeto, permitiendo recuperar sólo partes de éste si no se requiere el objeto completo.

Su petición es la siguiente:

```
GET /<api version>/<account>/<container>/<object> HTTP/1.1
Host: <servidor>
X-Auth-Token: <identificador de sesión>
```

Su posible respuesta correspondiente sería:

```
HTTP/1.1 200 Ok
Date: Wed, 11 Jul 2010 19:37:41 GMT
Server: Apache
Last-Modified: Fri, 12 Jun 2010 13:40:18 GMT
ETag: b0dffe8254d152d8fd28f3c5e0404a10
Content-type: text/html
Content-Length: 512000

[ ... ]
```

3.4.2.7 Creación, o actualización de objetos.

Es posible, y recomendable, que para la creación o actualización de objetos se utilicen sumas MD5 que ayuden a verificar la integridad de éste. Dicha funcionalidad está soportada a través del envío de una cabecera especial llamada Etag. Independientemente de si esta cabecera es incluida en la creación del objeto, al recuperarlo, siempre se devolverá el valor de la suma, para que el usuario pueda comprobar su integridad. Si se desea que el objeto expire en un tiempo determinado, o en una fecha, es posible añadir las cabeceras X-Delete-At y X-Delete-After. Su petición tiene la siguiente forma:

```
PUT /<api version>/<account>/<container>/<object> HTTP/1.1
Host: <servidor>
X-Auth-Token: <identificador de sesión>
ETag: <suma MD5>
Content-Length: <longitud en bytes>
X-Object-Meta-PIN: 1234

[ ... ]
```

Su posible respuesta sería:

```
HTTP/1.1 201 Created
Date: Thu, 07 Jun 2010 18:57:07 GMT
Server: Apache
ETag: d9f5eb4bba4e2f2f046e54611bc8196b
Content-Length: 0
Content-Type: text/plain; charset=UTF-8
```

3.4.2.8 Copia de objetos.

De cometerse un error con el nombre del objeto, al subirlo, o sencillamente querer cambiar

su nombre, el objeto debería eliminarse y subirse de nuevo, produciendo sobrecarga en las comunicaciones. Esto se evita mediante la copia en servidor de objetos. Esta copia puede realizarse de dos formas: a través de la cabecera X-Copy-From, donde se especificará el contenedor y el objeto que se desea copiar, o mediante la operación COPY. Detallamos sus formas a continuación:

```
PUT /<api version>/<account>/<container>/<destobject> HTTP/1.1
Host: <storage URL>
X-Auth-Token: <some-auth-token>
X-Copy-From: /<container>/<sourceobject>
Content-Length: 0

COPY /<api version>/<account>/<container>/<sourceobject> HTTP/1.1
Host: <storage URL>
X-Auth-Token: <some-auth-token>
Destination: /<container>/<destobject>
```

3.4.2.9 Eliminación de objetos.

La eliminación de objetos es inmediata y permanente, toda operación sobre el objeto después de la operación DELETE, devolverá el conocido error 404 de HTTP, objeto no encontrado. Es posible programar la eliminación mediante el uso de las cabeceras X-Delete-At y X-Delete-After. La petición debe tener la siguiente forma:

```
DELETE /<api version>/<account>/<container>/<object> HTTP/1.1
Host: <servidor>
X-Auth-Token: <identificador de sesión>
```

Su respuesta se asemejará a la detallada a continuación:

```
HTTP/1.1 204 No Content
Date: Thu, 07 Jun 2010 20:59:39 GMT
Server: Apache
Content-Type: text/plain; charset=UTF-8
```

3.4.3 Entorno de operación.

Para poder comprobar que nuestros drivers funcionan correctamente, es necesario preparar un servidor Openstack Swift donde poder realizar las conexiones pertinentes. Para este proyecto, hemos elegido usar una máquina virtual utilizando la tecnología de virtualización KVM, propia del núcleo de Linux. La máquina ha sido dotada de un sistema operativo Debian GNU/Linux en su versión de pruebas(actualmente Debian GNU/Linux Wheezy), y dispone de dirección IP propia dentro de la red gracias a la compatibilidad de KVM con los puentes virtuales que ofrece Linux. Además del software básico que se instala automáticamente con la distribución, se le ha añadido un servidor OpenSSH para realizar conexiones seguras a la máquina y los paquetes, incluidos en los

repositorios de la distribución, de Openstack Swift. De esta forma puede ejecutarse la máquina virtual como si de un servidor del sistema real se tratara, y podemos realizar cualquier prueba pertinente de nuestros drivers. Para la configuración del servidor Swift, hemos seguido las instrucciones detalladas en la documentación de Openstack SAIO(Swift All In One).

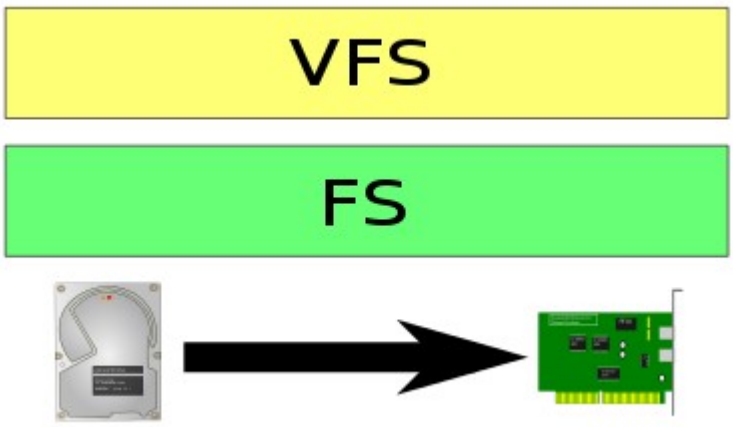
3.5 Linux.

En el conocido sistema operativo libre, creado por Linus Torvalds y publicado en el año 1991, los programas en espacio de usuario deben comunicarse con el núcleo a través de módulos que permitan la gestión de dispositivos en el sistema. Dichos módulos representan tres tipos de dispositivos: de caracteres, de bloques, y de red.

Los dispositivos de caracteres son aquellos que se comunican mediante flujos de caracteres secuenciales. Un ejemplo sencillo es el teclado, al que puede accederse, y la información transmitida es un flujo de caracteres secuencial: si tecleas una palabra, ésta se tratará en su estricto orden, y no en otro distinto. Los dispositivos de bloque suelen ser dispositivos de almacenamiento masivo y se estructuran en bloques o porciones a los que se accede en secuencias de operaciones, y a sus bloques no necesariamente se accede de manera secuencial. Por último, los dispositivos de red se encargan de cualquier tipo de comunicación con el exterior del sistema a través de sus interfaces de red, como las tarjetas Ethernet, modems, etc.

Los dispositivos representados no tienen la obligación de ser dispositivos reales, pudiendo permitirse la existencia de dispositivos virtuales que resuelvan un tipo de tarea determinada. Un ejemplo claro de dispositivo virtual es el dispositivo de generación de números aleatorios(al que se accede desde el fichero /dev/random), que no requiere de ningún hardware especifico para desempeñar su función.

La intención principal de Swiftness es proveer de un dispositivo de bloques que sea capaz de acceder al servidor Swift de Openstack, como si de un dispositivo local se tratara. Esto conlleva la clara diferencia de que, si detrás de un driver de dispositivo de bloques solemos encontrar dispositivos de almacenamiento físicos(como por ejemplo, un disco duro, una cinta magnética o un CD-ROM), en este caso nos encontraremos con una interfaz de red que debe saber comunicarse con el servidor.



Cabría esperar que la labor de Swiftness fuera la de la traducción de las peticiones de

bloques a directivas TCP. Sin embargo dicha labor ya se encuentra cubierta por el módulo NBD(Network Block Device) que estudiaremos más adelante. Con esto, la labor de Swiftness se reduce a implementar paquetes con las cabeceras adecuadas para que el servidor Swift pueda ser gobernado por el módulo NBD.

3.5.1 Dispositivos de bloques.

Los drivers dispositivos de bloques, dentro del código fuente de Linux, se alojan en el directorio “drivers/block”, y las interfaces que deben cumplir en el directorio “include/linux”. Como antes adelantábamos, dichos dispositivos disponen de grandes cantidades de información, dividida en bloques a los que no se accederá secuencialmente y es este hecho el que los diferencia de los dispositivos de caracteres. Debido a que tendremos que navegar a lo largo del dispositivo para acceder a la información, estos dispositivos ganan complejidad. La naturaleza de dichos dispositivos hacen al sistema altamente sensible a su disponibilidad.

El tamaño de un sector depende del dispositivo en cuestión, y es la unidad mínima fundamental de un dispositivo de bloques. Aunque muchos dispositivos de bloques dispongan de un tamaño de sector de 512 bytes, esto no quiere decir que sea un tamaño estándar. El sistema operativo debe cumplir con ciertas restricciones a la hora de acceder a un dispositivo de bloque. Debe acceder a la información en bloques de tamaño múltiplo del tamaño de sector, el tamaño de este bloque, deberá ser potencia de dos(restricción que suele aplicarse también al tamaño de sector), y el tamaño de bloque no puede ser mayor que el tamaño de una página de información en memoria principal. Los tamaños más habituales son 512 bytes, 1 kilobytes, y 4 kilobytes.

3.5.1.1 Proceso de una operación.

La jerarquía de subsistemas para acceder a un dispositivo de bloques, desde que una simple operación de lectura es enviada desde el espacio de usuario, hasta que llega al dispositivo es amplia. Ésta contempla desde el sistema de ficheros virtual, hasta el driver del dispositivo de bloques, pasando por las caches de discos, los distintos sistemas de ficheros, conformando la capa de direccionamiento del sistema, la capa genérica del dispositivo de bloques, y el planificador de entrada salida. Debido a esto, resulta interesante estudiar qué sucede, paso por paso, cuando una operación sobre el dispositivo está procesándose.

Suponemos que se ha llamado a la rutina de servicio read() para hacer un acceso de lectura. Este acceso de lectura provoca que el sistema active la función más adecuada dentro del sistema de ficheros virtual para procesarla. Esta función recibirá el descriptor del fichero abierto a leer y un desplazamiento dentro del fichero, para describir el bloque que se quiere leer. El sistema de ficheros virtual, debe determinar si los datos están ya disponibles en memoria principal, y cómo se realizará el acceso.

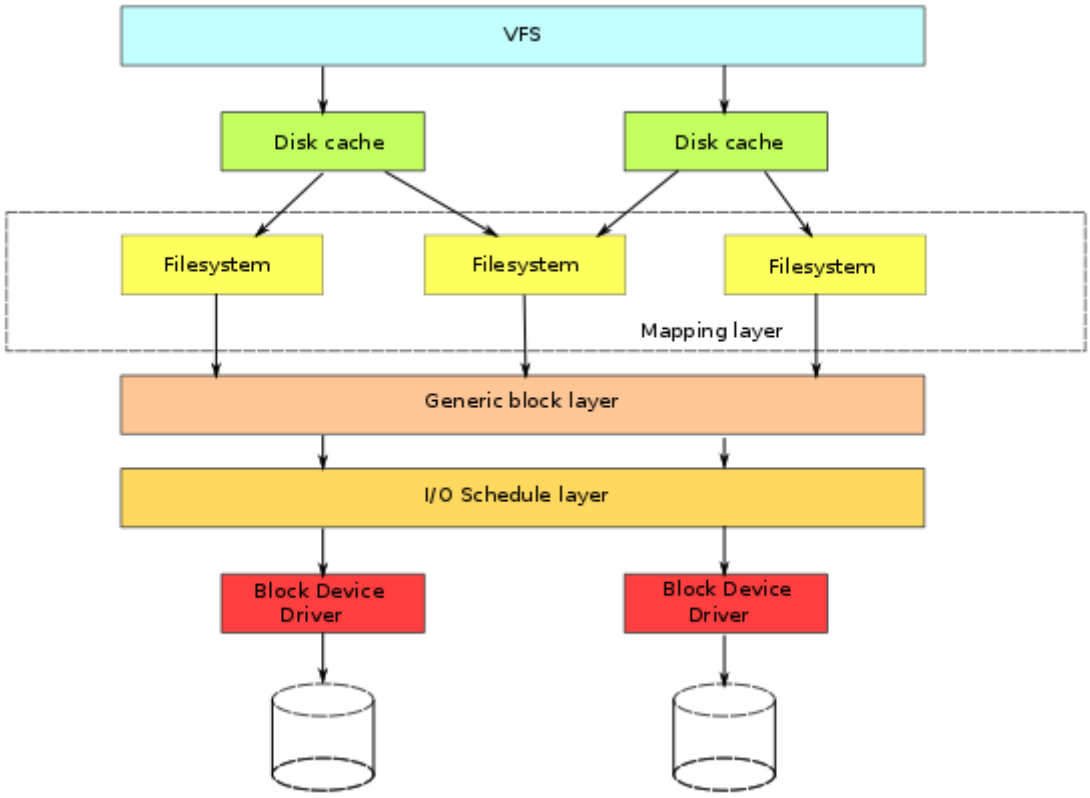
De requerir acceder al dispositivo de bloques, el sistema de ficheros virtual acudirá a la capa de direccionamiento. En esta capa se ejecutarán dos tareas principales. La primera será determinar el tamaño de bloque, para calcular el desplazamiento dentro del fichero en función de este tamaño. Posteriormente se invoca una función del sistema de ficheros real que determine el nodo del fichero y su posición dentro del disco.

Tras estas operaciones, el núcleo accederá a la capa genérica de dispositivos de bloque para

comenzar la operación de entrada/salida. Aquí se crearán e inicializarán las estructuras necesarias que se ofrecerán al planificador de accesos.

El planificador encola y ordena los accesos a los distintos bloques dentro del dispositivo con el fin de optimizar el acceso al dispositivo de bloques real, ya que, de una sola operación en él, accederemos a diversos bloques, y, debido a la lentitud del acceso a este tipo de dispositivos, es necesario asegurarse de que el número de accesos se reduzca todo lo posible.

Finalmente, las peticiones se ofrecen al driver del dispositivo de bloques, que ejecutará la operación real. Todo este largo proceso puede verse resumido en la siguiente figura, donde se observa la jerarquía completa de subsistemas.



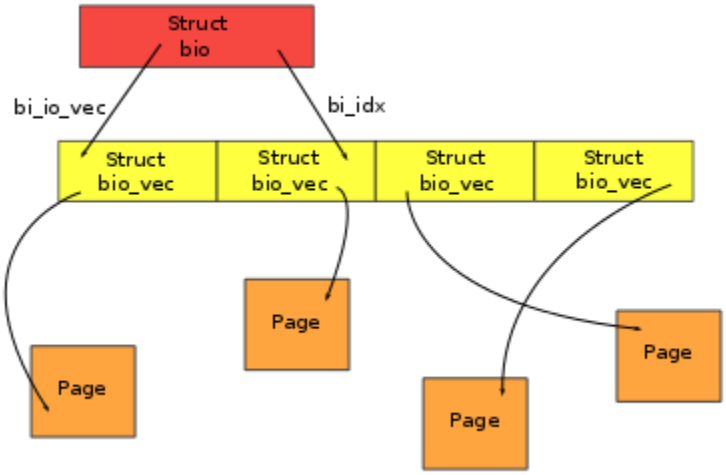
3.5.1.2 Estructura genérica de dispositivos de bloques.

Cada vez que un bloque es alojado en memoria principal, éste es asociado a un buffer. Un buffer es un objeto que representa un bloque en memoria, y, dado que el núcleo requiere más información que ésta, cada buffer se asociará a un descriptor llamado buffer_head. Esta estructura almacenará datos como el estado del bloque, su página asociada, su tamaño, un puntero al comienzo de los datos, y el dispositivo al que pertenece, entre otros detalles. Puede consultarse la estructura en el fichero “include/linux/buffer_head.h”. Aunque mucha de esta información es necesaria para el núcleo, no deja de ser una sobrecarga y un consumo de espacio en memoria.

Otra estructura de importancia, es aquella que almacena la información de una operación de entrada/salida. Ésta es la estructura bio (block input/output, detallada en el fichero “include/linux/bio.h”). Las operaciones descritas por esta estructura operan sobre segmentos, o porciones de un buffer, que no tienen la obligación de estar alojadas de forma contigua en memoria

principal. De esta forma, el núcleo puede operar con un sector repartido a lo largo de toda la memoria. La información de esta estructura es mayoritariamente informativa, y sus campos mas relevantes con bi_vcnt, bi_idx y bi_io_vec. Estas estructuras se organizan en un vector de estructuras llamadas bio_vec. El campo bi_io_vec, almacena la dirección de su vector asociado, bi_idx, su posición dentro del vector, y bi_vcnt, el número de estructuras bio_vec en el vector. Dicho esto sólo queda detallar que las estructuras bio_vec se encargan de almacenar la información relativa a la página de memoria donde se aloja el segmento, su tamaño, y el desplazamiento dentro de la página.

Con esta organización, el subsistema de entrada/salida de dispositivos de bloques de Linux es capaz de describir operaciones, ordenarlas a conveniencia(normalmente por proximidad de páginas dentro del almacenamiento masivo), añadir y eliminar operaciones con relativa facilidad, y utilizar distintos esquemas de acceso al dispositivo de bloques. El panorama general se asemeja a la siguiente figura:



El sistema de ficheros virtual maneja todas estas estructuras, a través de una estructura request_queue, que, como indica su nombre, es una cola de peticiones. Mientras la cola no esté vacía, el driver del dispositivo de bloques irá retirando peticiones, o estructuras request, de la cabeza de la cola y enviándole dichas peticiones al dispositivo que soporta. La estructura request está compuesta de una o más estructuras bio que operan sobre bloques consecutivos en el dispositivo. Estas estructuras vienen definidas en el fichero “include/linux/blkdev.h”.

Estas colas de peticiones serán manejadas por los distintos organizadores de las operaciones de entrada/salida. Entre los organizadores más importantes nos encontramos, el ascensor de Linus(Linus Elevator Scheduler), el organizador por plazos(Deadline Scheduler), el organizador anticipado(Anticipatory Scheduler), el organizador de cola completamente justo(Complete Fair Queue Scheduler), y el organizador de no operación(Noop Scheduler).

3.5.2 Dispositivos de red.

Los dispositivos de red en Linux disponen de un comportamiento similar al de un dispositivo de bloques montado. Éste debe registrarse en el núcleo del sistema operativo utilizando

las estructuras habilitadas para tal efecto antes de que pueda transmitir o recibir ningún paquete. Aún así, parece razonable pensar que existen ciertas diferencias. Los dispositivos de red no disponen de un fichero en el directorio "/dev" que sirva de punto de acceso, puesto que sus operaciones principales ya no van a ser las de leer o escribir, sino las de transmitir o recibir. Además, utiliza un espacio de nombres distintos y provee al usuario de operaciones distintas, que operan sobre objetos diferentes en el núcleo.

Otra diferencia a subrayar es que un dispositivo de bloques actúa como consecuencia de una petición del núcleo, mientras que el dispositivo de red debe pedir al núcleo que procese el paquete que ha recibido asíncronamente. Esto implica que aunque el subsistema de red de este sistema operativo sea independiente del protocolo utilizado, su API sea completamente distinta.

La forma en que un dispositivo de red se registra en Linux es insertar una estructura net_device dentro de una lista global de dispositivos de red. Esta estructura se detalla en el fichero "include/linux/netdevice.h", y engloba toda la información relativa a la comunicación del dispositivo de red, además de información relativa a los diferentes protocolos soportados por el sistema. Los campos más importantes de dicha estructura son el nombre, su estado, la estructura net_device del siguiente dispositivo en la lista de dispositivos de red, los campos relativos a su espacio de memoria relacionado, su dirección, puerto, irq y canal de DMA.

Para alojar dinámicamente la estructura se dispone de la función alloc_netdev, registrando el dispositivo en el núcleo. Sin embargo, también es posible hacer este registro mediante las funciones alloc_etherdev(Ethernet devices, "include/linux/etherdevice.h"), alloc_fcdev(Fiber-channel, "include/linux/fcdevice.h"), alloc_fddidev(FDDI devices, "include/linux/fddidevice.h") o alloc_trdev(Token ring devices, "include/linux/trdevice.h"). La inicialización de las estructuras es realizada por dichas funciones, colocando unos parámetros por defecto que pueden ser modificados posteriormente. Dicha inicialización la realiza la función ether_setup en el caso de llamar a alloc_etherdev.

En el caso de descargar el módulo de la memoria, debe utilizarse las funciones unregister_netdev y free_netdev para liberar todos los recursos registrados durante la inicialización del modulo.

Durante la apertura y cierre del dispositivo cabe destacar la necesidad de activar y desactivar el bit IFF_UP, del campo flag de la estructura netdevice, para cada correspondiente operación., la necesidad de copiar la dirección MAC del dispositivo a la estructura, y el uso de las funciones net_if_start_queue y net_if_stop_queue, para iniciar o detener las transferencias.

En las transmisiones de paquetes se dispone de la función hard_start_xmit, que encola el envío de un paquete de datos almacenado en la estructura socket buffer(struct sk_buff, detallada en el fichero "include/linux/sk_buff.h"). Cualquier paquete que se vaya a transmitir por la interfaz de red debe pertenecer a un socket, puesto que el almacenamiento de entrada/salida de esta estructura son listas de estructuras sk_buff. El paquete debe de estar almacenado en la estructura tal y como deba aparecer en la interfaz física de red desde la que se vaya a enviar.

Es necesario que durante las transmisiones se controle el acceso concurrente a la función hard_start_xmit. Para ello se utiliza el mecanismo de spinlock. Si el dispositivo dispone de memoria limitada, debe controlarse además la detención y reanudación de la cola de transmisión. Esto se realiza mediante las funciones netif_stop_queue y netif_wake_queue.

Dichos dispositivos deben contemplar la posibilidad de fallo en las transmisiones a través de temporizadores. Sin embargo, no es necesario que sea el driver quien se encargue de preparar este mecanismo. El driver sólo requiere indicar el periodo de espera del temporizador anotándolo en el campo watchdog_timeo de la estructura net_device. Esta temporización se mide en Jiffies. En caso de expirar este tiempo, se llama al método tx_timeout.

La recepción de paquetes se controla de dos formas, mediante entrada/salida por interrupciones, o, en caso de redes de gran ancho de banda, por entrada/salida programada. Esto último se debe a que en este tipo de redes, la sobrecarga de pasar de contextos de ejecución, a un contexto de interrupción impide que se aproveche adecuadamente el ancho de banda disponible.

Para la recepción controlada por interrupciones, se define una función de recepción que utilice la función dev_alloc_skb para reservar espacio para un paquete en memoria. Esta función será llamada por el manejador de interrupción del dispositivo, contexto de interrupción, para almacenar el paquete en memoria principal, donde será procesado posteriormente.

Es posible liberar el espacio de almacenamiento de un socket buffer mediante las funciones dev_kfree_skb(para contextos de ejecución), dev_kfree_skb_irq(para contextos de nterrupción) y dev_kfree_skb_any(para cualquier contexto).

Para la entrada/salida programada, se utiliza una API diferente llamada NAPI(New API) que modela el comportamiento programado del dispositivo. Un driver que implemente esta API debe ser capaz de deshabilitar la interrupción de recepción de paquetes, manteniendo las interrupciones para las transmisiones válidas y otros eventos. Además, deberá implementar una función poll que se dedique a preguntar a la interfaz de red si ha recibido algo. En caso de no recibir nada, se reactivará la interrupción de recepción y se mantendrá al dispositivo a la espera.

3.5.3 Network Block Device.

Como reza la documentación incluida en el núcleo de Linux, este módulo se encarga de conectar a un servidor remoto que exporte un dispositivo de bloques, con el fin de que la máquina cliente pueda acceder a este como si de un dispositivo local se tratara. Si establecemos una simple comparación con el objetivo de este proyecto, podemos constatar que la única diferencia con este, es que el servidor, no es uno propio, como el nbd-server, sino el servidor Swift de Openstack, luego es razonable pensar que podemos partir de este módulo para alcanzar nuestro objetivo.

Este módulo envía las peticiones de un dispositivo de bloques genérico a través de la red, utilizando el protocolo TCP, para que el servidor la procese y envíe la respuesta al cliente de vuelta de la misma forma. Dichas peticiones, tienen la forma de estructuras request como las utilizadas por los dispositivos locales existentes en el sistema. Estas estructuras request se generan a raíz de las estructuras bio y bio_vec(comentadas en el apartado 5.1.2), tomando la información correspondiente al direccionamiento del bloque, y los bloques consecutivos que se quieren extraer.

El servidor funciona completamente en espacio de usuario, siendo necesario dicho módulo solamente en el cliente.

En el fichero <include/linux/nbd.h> podemos encontrar tres estructuras que definen el dispositivo de bloques, la petición y la respuesta. La estructura del dispositivo es la siguiente:

```
struct nbd_device {
int flags;
int harderror;                  /* Code of hard error                          */
struct socket * sock;
struct file * file;   /* If == NULL, device is not ready, yet       */
int magic;

spinlock_t queue_lock;
struct list_head queue_head;          /* Requests waiting result */
struct request *active_req;
```

```
wait_queue_head_t active_wq;
struct list_head waiting_queue;      /* Requests to be sent */
wait_queue_head_t waiting_wq;

struct mutex tx_lock;
struct gendisk *disk;
int blksize;
u64 bytesize;
pid_t pid; /* pid of nbd-client, if attached */
int xmit_timeout;
};
```

Sus campos más destacables son el entero flags para configurar sus parámetros, la estructura socket que realiza el envío y recepción de peticiones, la estructura file que define el fichero del dispositivo y las dos colas de peticiones active_wq y waiting_wq. El driver se apoya en dichas colas para ejercer sus funciones. Una petición nueva, es encolada en la cola waiting_wq. Cuando llega su turno en la cola, la petición se envía y pasa a la cola active_wq para esperar su respuesta. Estas operaciones sobre las listas suceden tras la protección de concurrencia del mecanismo de spinlock. Ademas, con el fin de evitar problemas de concurrencia, se protege la transferencia de estructuras mediante un mutex.

La estructura request de NBD es la siguiente:

```
struct nbd_request {
__be32 magic;
__be32 type;      /* == READ || == WRITE            */
char handle[8];
__be64 from;
__be32 len;
} __attribute__((packed));
```

Esta estructura dispone de un número mágico(o número único) para asegurar la consistencia de la petición, un tipo(lectura o escritura), el campo handle, que se utiliza para almacenar la dirección del manejador de la petición en curso, from, para almacenar la posición dentro del dispositivo donde se encuentra el fichero a tratar, y len, el tamaño de la petición, en múltiplos de bloques.

Finalmente la estructura reply se compone de lo siguiente:

```
struct nbd_reply {
__be32 magic;
__be32 error;              /* 0 = ok, else error          */
char handle[8];            /* handle you got from request      */
};
```

Podemos observar que dispone de campos similares a los de la estructura request, exceptuando que, en vez de denotar un tipo de respuesta, se almacena un posible código de error. Tampoco se requiere la posición de los bloques solicitados.

De entre todas las funciones que conforman dicho driver, alojado en <drivers/block/nbd.c> nos interesan mayoritariamente las funciones que se encargan de transmitir y recibir paquetes a través de la red, y las funciones que creen o procesen dichos paquetes pues son las funciones susceptibles de modificación. Entre ellas, encontramos la función sock_xmit, que se encarga de transmitir un buffer utilizando el socket propio del dispositivo. Gracias a que su objetivo es muy

simple, esta función no requiere ningún cambio, suponiendo una base de la comunicación imprescindible. Sin embargo, podemos observar las funciones sock_send_bvec y nbd_send_req, que se encargan de utilizar sock_xmit para enviar la estructura request y las estructuras bio_vec requeridas para procesar la operación en curso, y que con toda probabilidad deberán ser modificadas para permitir la comunicación con el servidor Swift. Estas dos funciones componen la comunicación en el sentido cliente-servidor.

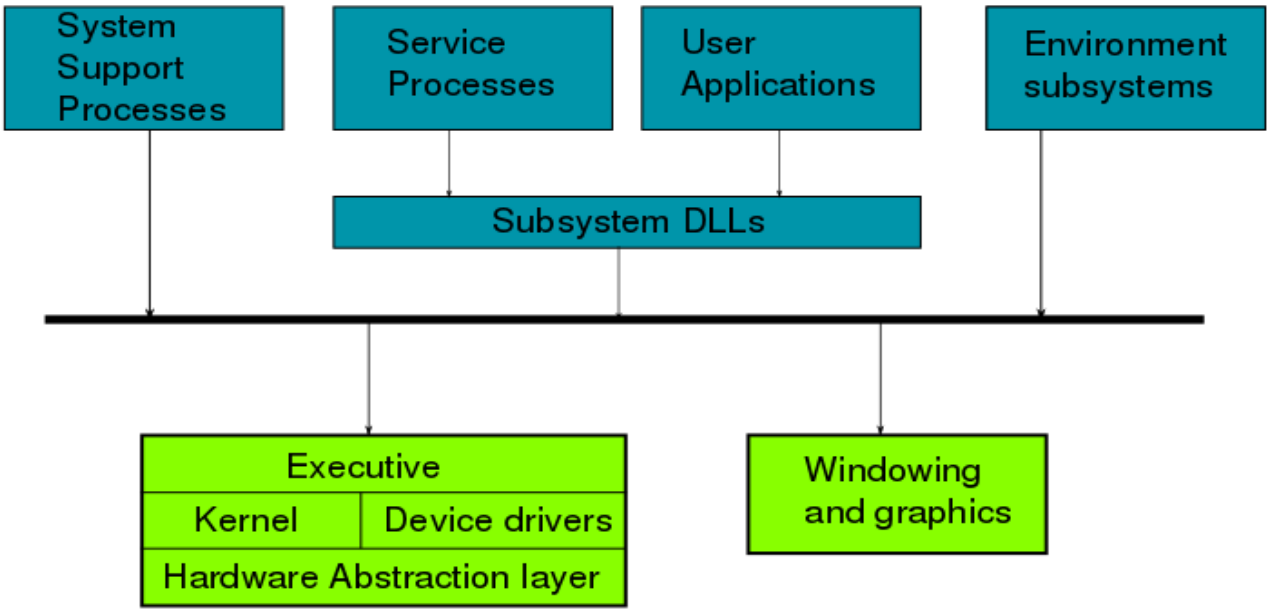
Las funciones sock_recv_bvec y nbd_read_stat son el equivalente a las funciones sock_send_bvec y nbd_send_req en el sentido opuesto de la comunicación, es decir, la comunicación servidor-cliente. La primera de ellas recibe las estructuras bio_vec requeridas, y la segunda se encarga de procesar la respuesta. Con el diseño adecuado, dichas funciones no deberían ser susceptibles de cambio.

La función nbd_handle_req añade la petición activa a la cabeza de la cola de peticiones activas, donde esperará su respuesta correspondiente. La función do_nbd_request será la encargada de encolar una nueva petición en la cola waiting_queue antes de ser enviada.

Se definen en este módulo una serie de operaciones ioctl para trabajar con la estructura del dispositivo NBD, las cuales no resultan de especial interés para el cometido de este proyecto.

3.6 Windows

Dentro de la arquitectura del conocido sistema operativo de Microsoft nos encontramos con diversos componentes organizados de la siguiente manera:



En la figura podemos diferenciar los subsistemas que trabajan en espacio de usuario y en espacio del núcleo. Dentro del espacio de usuario tendremos:

- **Procesos de soporte del sistema:** ejemplos de este subsistema son el proceso de identificación, el organizador de sesiones y otros procesos que no son iniciados por el

organizador de control de servicios

- **Servicios del sistema y aplicaciones de usuario:** éstos se apoyan en librerías dinámicas del sistema, y los subsistemas de entorno para realizar su cometido.
- **Las propias librerías dinámicas:** su cometido es traducir la llamada al sistema realizada por el servicio, o la aplicación de usuario, a la función adecuada dentro del sistema. Para ello, si lo requiere, la librería enviará un mensaje al subsistema de entorno.

En el espacio del núcleo, los componentes más importantes son:

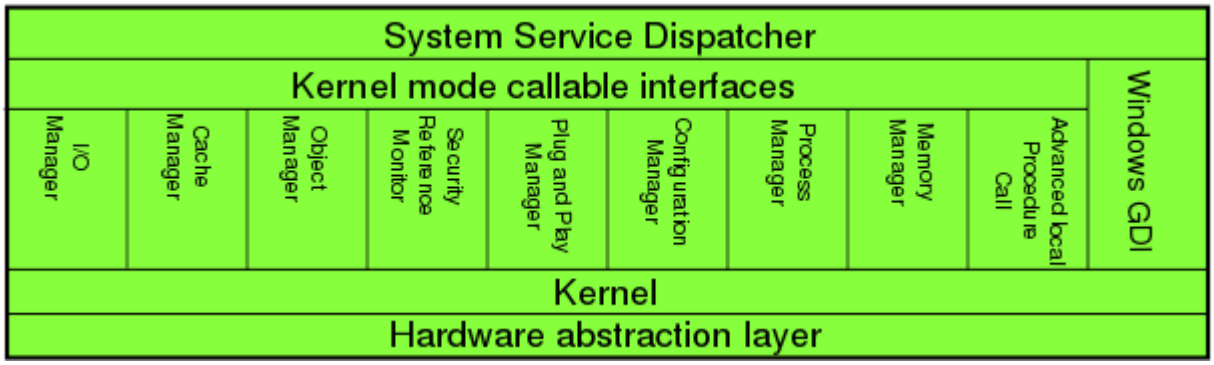
- **El ejecutor de Windows**, que conforma la base principal del sistema, ofreciendo los servicios mínimos del sistema(uso de memoria, creación de procesos e hilos, seguridad, entrada/salida...)
- **El núcleo**, que reúne las funciones en bajo nivel requeridas para que el ejecutor ofrezca los servicios mínimos
- **Los drivers de dispositivos**, que se comunicarán con el hardware
- **La capa de abstracción de hardware**, que aísla el núcleo del ejecutor de Windows. Como se puede observar, el sistema gráfico es independiente del bloque de sistemas antes comentado.

Analizando en mayor detalle los subsistemas que componen el núcleo, podemos encontrar los siguientes componentes:

- **El manejador de entrada salida:** dicho subsistema se apoya en los drivers de dispositivos o en los sistemas de ficheros para realizar delegar la tarea encomendada al componente correspondiente.
- **El manejador de cache**, que se encarga de organizar los datos requeridos recientemente por el espacio de usuario en memoria, manteniéndolos en esta para agilizar posteriores accesos.
- **El manejador de procesos e hilos**, que se encarga de iniciar y finalizar los distintos procesos e hilos y de ofrecerles el soporte requerido.
- **El manejador de objetos del núcleo**, cuyo objetivo principal es mantener las estructuras necesarias para su correcto funcionamiento.
- **El manejador de dispositivos Plug and Play**, subsistema que detecta la aparición de un nuevo dispositivo en el sistema y pone en funcionamiento las infraestructuras necesarias del núcleo que permitirán su correcto funcionamiento.
- **El monitor de seguridad**, que activará las políticas de seguridad de acceso a cualquier medio.
- **El manejador de memoria**, que se encarga de alojar información en memoria principal, o devolverla al dispositivo de bloques del que proviene.
- **El manejador de configuración del sistema**, que gestionará el registro de Windows y la información contenida en él.
- **El manejador de entrada/salida**, que implementa una interfaz general de entrada salida independiente del dispositivo.
- **Las rutinas de instrumentación** ofrecidas por el servicio WMI de Windows, que permiten a los drivers de los dispositivos publicar información de configuración y estadísticas sobre el dispositivo.
- **El invocador de procedimientos avanzado**, que realiza las tareas necesarias para la ejecución de cualquier software.

- **La interfaz de dispositivos de gráficos**, apoyada en sus correspondientes dispositivos, y que realizará la tarea de gestionar el entorno gráfico del sistema.

Estos subsistemas están gobernados por el ejecutor de Windows y quedan resumidos en la siguiente imagen:



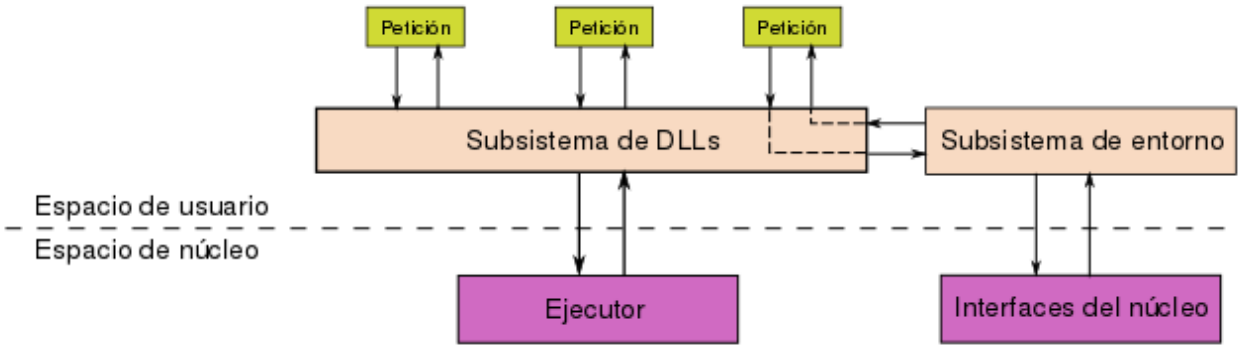
Analizaremos posteriormente en mayor detalle los subsistemas más relevantes.

3.6.1 Subsistema de entorno y librerías dinámicas.

Este componente ofrece aislamiento entre el espacio de usuario y el espacio del núcleo, sin llegar a pertenecer a este último. Actúa como un filtro de primera instancia donde una función de dichas librerías puede actuar de las siguientes formas:

- La función se implementa en la librería, y por tanto, el espacio del núcleo no requiere saber de la llamada a esta función, luego será resuelta por la librería y se continuará con la ejecución normal del sistema.
- La función requiere llamar al ejecutor de Windows para realizar alguna de sus funciones, luego se realizarán algunas tareas puntuales en espacio del núcleo y se continuará el resto de ejecución en espacio de usuario.
- La función requiere el uso del subsistema de entorno, enviándole un mensaje para que resuelva una tarea concreta, y continúe su ejecución posteriormente.

Estos comportamientos se resumen en la siguiente figura:



Entre los subsistemas de entornos, nos encontramos con el subsistema propio de Windows, y, por cuestiones de soporte, el subsistema POSIX (Portable Operating System Interface based on Unix). Puesto que nuestro objetivo se centra en el espacio del núcleo, no estudiaremos en mayor profundidad este componente.

3.6.2 Ejecutor de Windows.

El ejecutor de Windows conforma la primera capa de abstracción con la que una llamada al sistema se encontrará cuando se alcance el espacio del núcleo, haciendo de fachada para el resto del sistema. Está formado por numerosas funciones que pueden ser recogidas en los siguientes tipos:

- Funciones exportadas al espacio de usuario, o servicios del sistema, la librería Ntdll se encarga de ofrecerlos. Muchos de los servicios son accesibles desde la API de Windows, o a través del subsistema de entorno.
- Drivers de dispositivos, llamados a través de la función DeviceIOControl y que ofrece una interfaz general al espacio de usuario para llamar a los servicios de un dispositivo.
- Funciones que sólo pueden ser llamadas desde el espacio del núcleo.
- Funciones definidas como símbolos globales, pero que no son exportados al espacio de usuario.

Además, el ejecutor de Windows contiene cuatro grupos de funciones de soporte para los componentes organizados por éste. Los grupos son los siguientes:

- Las funciones del manejador de objetos del ejecutor, que crea y destruye los objetos que sirven de infraestructura para el núcleo.
- El ALPC(Advanced Local Procedure Call, lanzador de procedimientos locales avanzados) que se encarga del paso de mensajes entre procesos.
- La librería de funciones comunes del sistema.
- Las rutinas de soporte del ejecutor.

3.6.3 El núcleo.

El núcleo consiste en una serie de funciones contenidas en Ntoskrnl.exe que ofrece los mecanismos principales para el funcionamiento del sistema. Es utilizado por el ejecutor como una arquitectura de hardware a bajo nivel y es completamente dependiente, por tanto, de la arquitectura donde se ejecute.

Dentro de él se utilizan objetos de control y primitivas que serán controladas por el ejecutor, que expondrá los recursos compartidos utilizando las distintas políticas de seguridad. Además dispone del control de región del procesador y del bloque de control para almacenar información específica del procesador y su ejecución en curso.

3.6.4 Drivers de dispositivos.

Los drivers en Windows conforman módulos que pueden cargarse en memoria principal bajo demanda. Pueden ser utilizados en contextos donde un hilo en espacio de usuario utiliza una de sus funciones, en contextos en los que un hilo en espacio del núcleo lo utiliza, o como resultado de una interrupción. Los drivers no utilizan los dispositivos directamente, sino que utilizan las funciones de la capa de abstracción de hardware(HAL, Hardware Abstraction Layer) para cumplir su cometido. Los tipos de drivers existentes son los siguientes:

- **Drivers de dispositivos**, que utilizan la capa HAL para comunicarse con estos.
- **Drivers de sistemas de ficheros**, que aceptan peticiones genéricas de entrada salida y las traducen a peticiones de un dispositivo concreto.
- **Drivers de filtro de sistemas de ficheros**, que colaboran con los de sistemas de ficheros para ofrecerle una nueva funcionalidad.
- **Redirectores de red y servidores**, que, siendo drivers de sistemas de ficheros, transmiten las peticiones de un sistema de ficheros a través de la red y reciben las peticiones.
- **Drivers de protocolos**, los cuales implementan los protocolos de comunicaciones básicos y más extendidos como TCP/IP, NetBEUI e IPX/SPX.
- **Drivers de filtro para flujos de ejecución del núcleo**, que implementan el procesos de señalización en grandes flujos de datos.

Con esto, el modelo de drivers puede ser resumido en tres tipos concretos:

- **Drivers de bus**, que se encargan de utilizar los controladores de bus, adaptadores, puentes, y cualquier dispositivo capaz de comunicar diferentes dispositivos.
- **Drivers de función**, que implementan la funcionalidad principal de un dispositivo en cuestión.
- **Drivers de filtro**, que implementan una funcionalidad añadida a un driver de función.

Para el desarrollo de estos drivers, la Windows Driver Foundation provee al usuario de las plataformas para drivers en espacio del núcleo(KMDF, Kernel-mode driver framework), y para drivers en espacio de usuario(UMDF, User-mode driver framework).

3.6.5 Componentes del sistema de entrada/salida.

Dentro del sistema de entrada/salida existen una serie de componentes que colaboran habitualmente para ofrecer los servicios de los drivers al espacio de usuario. Estos componentes son el gestor de entrada/salida, el gestor del consumo energético, el gestor de dispositivos Plug and Play, y las rutinas de servicio para el proveedor de instrumentación de Windows.

El gestor de entrada salida es el principal componente de este subsistema, cumpliendo el objetivo de conectar dispositivos(virtuales, lógicos y físicos) con el espacio de usuario. Conforma una infraestructura de soporte para los drivers.

Los drivers de dispositivos ofrecen una interfaz de comunicación que es dirigida a través del gestor de entrada/salida. Cuando el usuario envía una petición, el gestor de entrada/salida la recoge y decide quien debe responder a esta, dirigiendo la petición al dispositivo adecuado.

El gestor de dispositivos Plug and Play permanece en contacto continuo con el gestor de entrada/salida y con los drivers de bus. Los drivers de bus le ofrecen alojamiento para los nuevos dispositivos, aparte de avisar de la detección de un nuevo dispositivo, o de su desaparición. Este subsistema cargará los módulos correspondientes y si no dispone de ellos, llamará a un gestor en el espacio de usuario.

El gestor de consumo energético realizará las transiciones adecuadas de los dispositivos entre los distintos estados de consumo(encendido, apagado, modo de bajo consumo, etc).

Las rutinas de servicio del proveedor de instrumentación de Windows hacen de interfaz de comunicación entre el proveedor de instrumentación y los drivers de los dispositivos.

Además de estos componentes, es conveniente recordar que en este subsitema utiliza el registro de Windows para mantener una base de datos de los dispositivos básicos, los ficheros INF, que sirven para almacenar la información de los drivers instalados en el sistema, y el nivel de abstracción de hardware.

3.6.5.1 Gestor de entrada/salida.

Este subsistema sirve de infraestructura para recoger peticiones de entrada/salida, en forma de paquetes llamados IRP(I/O Request Packet). Su diseño permite que un hilo de procesamiento se encargue de múltiples peticiones concurrentemente. El IRP contiene la información necesaria para describir la petición. El gestor de entrada/salida enviará un puntero a la dirección de memoria de la petición al driver correspondiente, donde será procesada. Este mismo puntero se utilizar para recuperar la información de la petición., cuando el driver notifique que la operación ha concluido.

Por otro lado este subsistema ofrece un código común para todos los dispositivos, permitiendo que los drivers reduzcan su tamaño, retirándoles el código de las tareas comunes con otros dispositivos.

Los drivers disponen de una interfaz modular y uniforme, de forma que el gestor de entrada salida puede tratarlos sin necesidad de tener ningún conocimiento acerca de como procesa las peticiones. De esta forma, el driver sólo tiene que encargarse de traducir una petición genérica a una petición especifica del hardware que lo gestiona. Es posible que los drivers se llamen unos a otros para resolver la gestión de la petición en curso.

Una operación típica de entrada/salida, es tratada por el gestor de entrada/salida, uno o dos

dispositivos, y el nivel de abstracción de hardware. El sistema operativo, enmascara los dispositivos, de forma que el espacio de usuario dispondrá de las operaciones normales en un fichero cualquiera.

3.6.5.2 Drivers de dispositivos.

Atendiendo a la clasificación de si se trata de un driver en espacio de usuario o en espacio de núcleo, podemos constatar que en el espacio de usuario disponemos de tres tipos: Drivers de Dispositivos Virtuales(VDDs), drivers de impresoras, y drivers de la infraestructura de drivers en espacio de usuario.

Los drivers de dispositivos virtuales se mantienen por compatibilidad con MS-DOS, ya que emulan aplicaciones de 16 bits, capturando sus llamadas a dispositivos y traduciéndolas a los dispositivos reales.

Los drivers de impresoras traducen las peticiones de dispositivos de gráficos genéricos a comandos de impresión.

Los drivers de la infraestructura de drivers en espacio de usuario son drivers de dispositivos que no requieren ejecución en espacio del núcleo, y que, en caso de necesitar algo, les es suficiente con comunicarse a través de las llamadas a procesos locales avanzados de Windows.

En el espacio del núcleo, disponemos de los drivers de sistemas de ficheros, que recogen las peticiones y las envían al dispositivo de bloques, o interfaz de red correspondiente, los drivers de dispositivos Plug and Play(PnP), entre los que se incluyen drivers de dispositivos de almacenamiento masivo, adaptadores de video, dispositivos de en entrada e interfaces de red, y drivers de dispositivos no Plug and Play, que conforman extensiones del núcleo, o nuevas funcionalidades para otros drivers.

Por otro lado, se encuentran los drivers pertenecientes al modelo de drivers de Windows(Windows Driver Model). Entre éstos encontramos los drivers de bus, los drivers de funcionalidad, y los drivers de filtro comentados anteriormente.

Puesto que un driver puede ser dividido en distintos niveles, es posible clasificarlos en otros tres tipos distintos, de clase, de puertos o de minipuertos.

Los dispositivos de clase implementan drivers para una clase en particular de driver(drivers de discos, de cintas, de CR-ROM...).

Los dispositivos de puertos procesan peticiones para enviarlas por un tipo específico de puerto, como un puerto SCSI, COM, USB. Éstos se implementan como librerías del núcleo, ya que son drivers escritos por Microsoft con el sistema operativo.

Los drivers de minipuertos traducen una petición genérica a un petición de un puerto concreto, siendo drivers para un dispositivo que importan las funciones de un dispositivo de puerto.

Si una petición debiera ser atendida por un driver con diferentes niveles, el gestor de entrada/salida recogería la petición del espacio de usuario, la traduciría a una petición para el driver que conforme el primer nivel, con su respuesta, generaría una nueva petición para el siguiente nivel, y seguiría generando posteriores peticiones sucesivamente hasta llegar a la última. Una vez respondida la última petición, se devolvería la respuesta al espacio de usuario, conformando una cola de peticiones por nivel.

3.6.5.3 Estructura de un driver.

Dentro de un driver se encuentran una serie de funciones que resuelven las diversas tareas

que requiere el sistema operativo para ofrecer el resultado de las operaciones al espacio de usuario. Pueden dividirse en dos subgrupos, las funciones principales, que se encuentran en todo driver, y las funciones auxiliares, que según las infraestructuras que requiera el driver, aparecerán, o no lo harán.

Dentro de las rutinas principales, o imprescindibles tenemos las rutinas de inicialización del driver(DriverEntry) que se utilizan recién cargado el driver en memoria principal, e inicializa las estructuras principales del driver.

Para el buen funcionamiento de la infraestructura PnP se incluye una rutina de añadido de dispositivo(add-device routine) que recibe la notificación del gestor PnP y aloja el objeto del dispositivo descubierto en el núcleo.

Es posible encontrar una serie de rutinas dentro del driver que conforman las operaciones que puede realizar el usuario sobre el dispositivo. Las principales suelen ser las rutinas de apertura, cierre, lectura y escritura, aunque pueden existir otras en función de las operaciones que ofrezca el dispositivo.

Si el driver se sirve de las colas de gestión proveídas por gestor de entrada/salida, definirá una operación start que iniciará la transferencia de peticiones. El gestor de entrada/salida se encargará entonces de serializar las peticiones y enviarlas una a una al dispositivo.

Por último se definen dos funciones relacionadas con el uso de interrupciones, la rutina de servicio de interrupción(ISR), y la rutina de resolución para el servicio de interrupción. La primera completará las labores mínimas de procesado, pues su ejecución debe ser lo más rápida posible. Habitualmente encola la petición, para que sea procesado posteriormente en la rutina de proceso del servicio de interrupción.

El resto de rutinas que detallamos a continuación, pertenecen al segundo grupo antes comentado, es decir, estas rutinas aparecen en función de ciertas condiciones que debe cumplir el driver, y de no cumplirse no existe la necesidad de implementarla.

Si el driver dispone de diferentes niveles encontraremos en este al menos una, aunque pueden ser varias, rutinas de finalización. Estas rutinas avisan a los niveles superiores de la resolución de la operación en niveles inferiores, y del estado de esta(correcta, incorrecta, cancelada).

El driver puede ofrecer una o varias rutinas de cancelación. Si la petición puede ser cancelada, esta incluirá la rutina que debe ejecutarse en caso de recibir una petición de cancelación. Esta rutina realizará labores de limpieza para que el driver pase a un estado consistente para procesar nuevas peticiones.

Si el driver está preparado para comunicarse con el gestor de cache, debe implementar una rutina de resolución rápida. Esta rutina permitirá resolver la petición a través de la cache del sistema, evitando realizar nuevas operaciones de entrada salida.

Es posible, pero no necesario, que el driver disponga de una rutina de descarga, para realizar las labores de liberación de memoria y recursos que este utilizara durante su ejecución. Además, es posible definir una rutina, aparte, para la limpieza de las infraestructuras en el apagado del sistema.

Por último, puede definirse una rutina de registro de errores.

3.6.5.4 Objetos de drivers y objetos de dispositivos.

El gestor de entrada/salida se comunica con dos tipos de objetos que le ofrecen las operaciones disponibles y cualquier información que necesite saber para llevarla a cabo. Son los objetos de driver, y de dispositivo. Los objetos de drivers representan el driver que puede manejar los distintos dispositivos conectados al sistema, y los de dispositivo, los dispositivos, físicos o lógicos que pueden ser gobernados por dicho driver.

Cuando el driver es cargado en memoria, el objeto de driver es creado. Inmediatamente después se llamará a la rutina de inicialización. A partir de entonces es posible crear objetos de dispositivo gobernados por dicho driver. Para ello se utilizan las operaciones IoCreateDevice e IoCreateDeviceSecure, aunque habitualmente los dispositivos utilizarán la rutina add-device y la infraestructura Plug and Play.

Al crearse el dispositivo, es posible asignarle un nombre que será almacenado en el gestor del espacio de nombres de objetos, que es inaccesible directamente desde el espacio de usuario. Para que sea accesible, debe hacerse un enlace desde el directorio \Global hacia el directorio \Device, del espacio de nombres, donde el nombre del objeto realmente se encuentra. Para exportar las interfaces se utiliza la función IoRegisterDeviceInterface, y para habilitarlas, se utiliza IoSetDeviceInterfaceState. Desde ese momento, el usuario puede utilizar la infraestructura PnP para utilizar la interfaz.

Con esta infraestructura, un objeto driver es asociado con los distintos objetos dispositivos. De esta forma, el gestor de entrada/salida no requiere conocer los detalles de un dispositivo concreto.

3.6.5.5 Estructuras de ficheros.

Las estructuras de ficheros cumplen los requisitos de diseño de los objetos en Windows(son recursos compartidos, pueden tener nombre, deben estar protegidos y soportan sincronización). Su única diferencia, es que los objetos compartidos suelen estar alojados en memoria, y en los ficheros, se encuentran en un dispositivo físico. Su representación se compone de los siguientes atributos:

- **Nombre de fichero.**
- **Desplazamiento del byte actual**(respecto al comienzo del fichero).
- **Modo de compartición.**
- **Banderas de modo de apertura.**
- **Puntero al objeto de dispositivo.**
- **Puntero a la partición.**
- **Puntero a la sección de punteros de objetos.**
- **Puntero al mapa privado de cache.**
- **Lista de paquetes de petición de entrada/salida.**
- **Contexto de finalización de entrada/salida:** de existir, comunica el estado de finalización de la petición en curso sobre el puerto
- **Extensiones del objeto de fichero:** almacena la prioridad, si requiere comprobaciones de seguridad, y si contiene extensiones que almacenan información de contexto.

El nombre y el desplazamiento se utilizarán para posicionarse en la lectura secuencial del fichero. El modo de compartición y las banderas de modo de apertura controlan la seguridad de los accesos. El puntero al dispositivo, a la partición, y a la sección de punteros de objetos identifican el dispositivo físico que aloja el fichero, el mapa de cache y la lista de IRPs para procesar las peticiones. Las extensiones del objeto de fichero pueden ser las siguientes:

- **Parámetros de transacción.**
- **Detalles del objeto de dispositivo:** Identifica el driver de filtro adecuado.

- **Estatus de bloqueo de rango de entrada/salida:** permite al espacio de usuario bloquear buffers en el espacio del núcleo optimizando las operaciones asíncronas.
- **Genérica:** almacena información específica para el driver de filtro.
- **Entrada/salida programada del fichero:** almacena la reserva de ancho de banda del dispositivo para garantizar el buen funcionamiento de aplicaciones multimedia.
- **Enlaces simbólicos:** información para resolver enlaces simbólicos.

Cuando el espacio de usuario pide al núcleo la apertura de un fichero, el gestor de entrada/salida se comunica con el subsistema adecuado(en este caso, el gestor de objetos), para que éste, con ayuda del núcleo genere un manejador para el fichero. El gestor de entrada/salida, enviará posteriormente este manejador a la aplicación que la solicitó. Durante este proceso, cualquier comprobación de seguridad debe ser resuelta por el gestor de entrada/salida.

El objeto de fichero es alojado en memoria y es una representación de un recurso compartido, no el recurso en sí. Esta representación almacena los datos necesarios para utilizar el fichero, mientras que el fichero almacena los datos a utilizar. Por otro lado, el manejador debe ser único para un proceso, mientras que el fichero no tiene por que serlo(por ejemplo varios procesos utilizan el mismo fichero). En caso de escrituras, se evitan los problemas de concurrencia utilizando la función de Windows LockFile.

Cuando el fichero se abre, se incluye en el espacio de nombres bajo el directorio "\Device\HarddiskVolumeN", que representa el directorio de la partición abierta, de la que se recibe el fichero. Para que sea compartido directamente con el espacio de usuario, es necesario hacer un enlace simbólico en el directorio "\Global".

3.6.5.6 Paquetes de peticiones de entrada/salida.

Las IRPs son las estructuras que Windows utiliza para almacenar la información requerida para realizar una operación sobre un dispositivo. Si un hilo utiliza un servicio de entrada/salida, el gestor de entrada salida generará el IRP correspondiente a la operación solicitada. El sistema gestiona las peticiones utilizando dos colas y una pila, una cola corta para las operaciones que sólo utilicen una posición en la pila, y una cola larga para las que utilicen múltiples posiciones en la pila. Para mejorar la gestión de operaciones entre varios procesadores, se añade una lista global además de las anteriores.

Por defecto un IRP utiliza 10 posiciones en la pila de IRPs, y, por tanto, se aloja en la lista larga de IRPs. Una vez por minuto, el sistema ajusta cuantas posiciones de la pila requiere, llegando a permitir un máximo de 20 posiciones. Después de su alojamiento e inicialización, el gestor de entrada/salida almacena un puntero al solicitante del servicio en el IRP.

Dentro de un IRP se diferencian dos partes: una cabecera fija y una o varias posiciones en la pila. La información del tipo de operación se almacena en la cabecera, mientras que en las posiciones de la pila se almacena código de una función (llamada función mayor), sus parámetros, y el solicitante del objeto fichero. La función mayor indica qué rutina de resolución del driver debe ser llamada por el gestor de entrada/salida cuando el paquete se envíe al driver. Opcionalmente, puede almacenarse una función(llamada menor) para que modifique el comportamiento de la función mayor (añadiendo funcionalidades, comunicando con otros subsistemas, etc).

Habitualmente se ofrecen rutinas de resolución para las operaciones más habituales (apertura, cierre, lectura, escritura, control de entrada/salida, Plug and Play y alguna rutina para WMI).

Una vez almacenado el IRP en la lista correspondiente, el sistema es capaz de encontrarlo y cancelarlo en caso de descontrol.

La creación de un IRP se realiza utilizando los servicios NtReadFile, NtWriteFile, o NtDeviceIoControlFile. El gestor de entrada/salida decide si debe participar en la gestión de los buffers de almacenamiento requeridos por la operación. Puede actuar de tres formas. La primera de ellas sería almacenar un buffer del mismo tamaño que el que ofrece la aplicación que solicita el servicio. En caso de escritura, copia el buffer a su almacenamiento privado, y en el caso de lectura copia la información de su buffer al espacio de usuario.

Otra opción es realizar entrada/salida directa, bloqueando el buffer en espacio de usuario para que no pueda ser escrito, y desbloqueándolo cuando se finalice la operación.

Por último el gestor puede sencillamente no ejercer ninguna gestión de almacenamiento. En este caso, debe ser el driver quien se encargue de estas labores.

En cualquier caso, el gestor de entrada/salida inicializará las referencias adecuadas para localizar el almacenamiento. El tipo de gestión utilizado es seleccionado en función de lo que solicite el driver, el cual registra el tipo deseado para la lectura y escritura, mientras que para el control se almacena en un código de control específico.

Usualmente, para transferencias menores que una página de memoria(4KB) se utiliza la entrada/salida a través de buffers, y para transferencias mayores la entrada/salida directa. Es poco común utilizar la gestión de almacenamiento desde el driver por la posibilidad de que se pierda la referencia al solicitante. Su uso se reduce entonces a drivers en espacio de usuario que garantizan que las referencias son válidas, y pertenecientes al espacio de usuario

3.6.5.7 Petición en drivers de una sola capa.

Este proceso conlleva siete pasos:

- Envío de la petición a través del subsistema de DLLs.
- El subsistema de DLLs solicita el servicio NtWriteFile.
- El gestor de entrada/salida crea el IRP correspondiente y lo envía al driver haciendo uso de IoCallDriver.
- El driver transmite el IRP al dispositivo y comienza la operación de entrada/salida.
- El dispositivo notifica la finalización del servicio mediante una interrupción.
- El driver sirve la interrupción.
- El driver llama al servicio IoCompleteRequest.

Las interrupciones se sirven en dos partes diferenciadas. Una vez ocurrida, el gestor de interrupción reconoce el estado y razón de la interrupción para encolarlo en una cola de resolución, donde se gestionará su finalización en un segundo plano, permitiendo que sucedan nuevas interrupciones rápidamente.

La rutina de finalización se encarga de copiar la información pertinente hacia el espacio de usuario, y limpiar las estructuras requeridas para gestionar una nueva petición.

Debe cuidarse la sincronización al acceder al driver, debido a que el sistema permite que un hilo de mayor prioridad expulse a uno de menor prioridad de su ejecución, que se atienda una interrupción durante ésta, o que se ejecute el mismo código en otro procesador de la misma máquina. De esta forma, la información compartida por el driver debe ser consistente para cualquier ejecución en curso.

3.6.5.8 Petición en drivers de múltiples capas.

En ésta ocasión, el gestor de entrada salida, al igual que en el caso anterior genera un IRP que se pasará al driver del sistema de ficheros. Dependiendo de la petición, el driver la reenvía al driver del dispositivo en cuestión, o genera IRPs adicionales para enviarlas por separado al driver. Solo se reutiliza un IRP en caso de que se traduzca en una sola petición al dispositivo.

Como alternativa a reutilizar una petición, el sistema de ficheros puede generar un grupo de IRPs asociados que trabajen en paralelo para atender una sola petición. Este grupo es transmitido al gestor de volúmenes, que lo enviará posteriormente al driver del dispositivo.

3.6.5.9 Infraestructura de drivers en espacio del núcleo(KMDF),

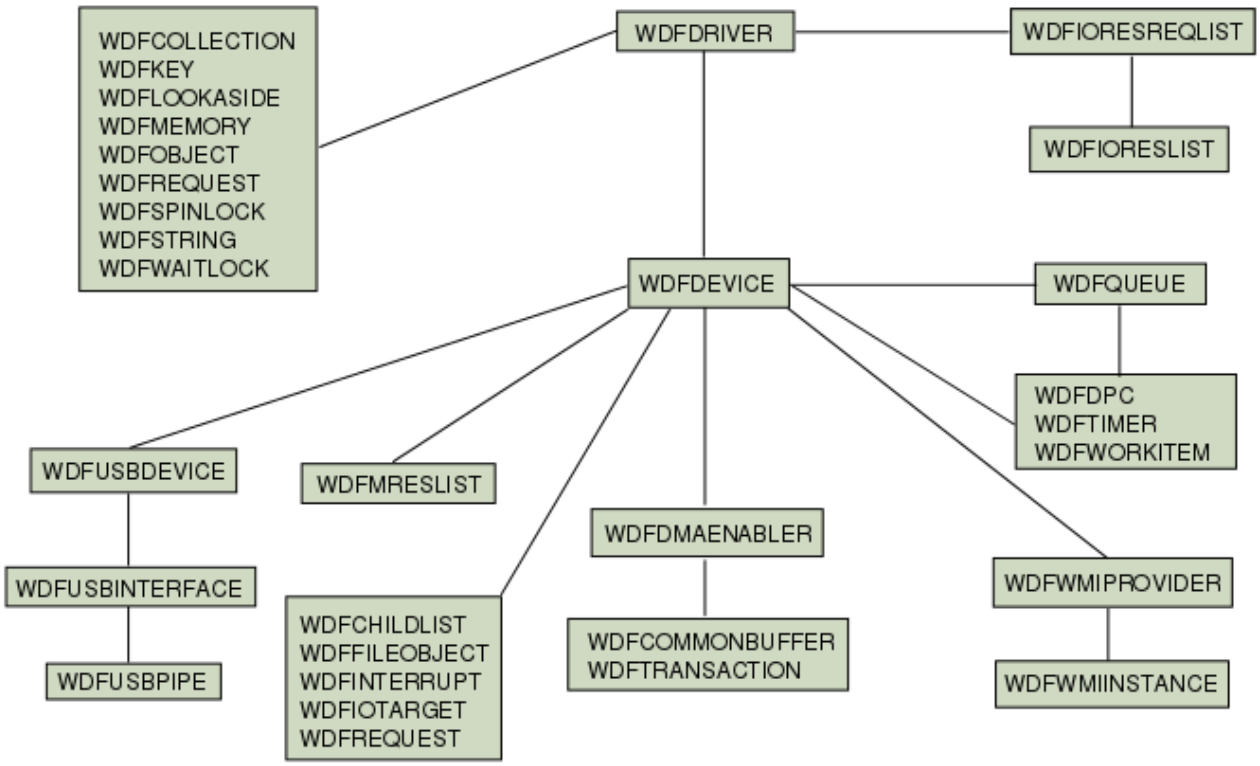
El modelo de drivers en espacio de núcleo no es muy distinto del modelo de drivers WDM. Esto implica que en su interior disponen de las funciones de inicialización, la rutina de añadido de dispositivo al sistema, y una o más rutinas de eventos, que funcionan como las rutinas de resolución de los drivers pertenecientes a WDM. Generalmente, estas últimas, crean y gestionan colas de peticiones para la entrada/salida. El driver no tiene la obligación de implementar estas rutinas, puesto que existen rutinas genéricas proveídas por el sistema operativo. En el caso de implementarlas, la rutina debe ser registrada por el sistema operativo para que pueda ser utilizada.

El modelo de datos de KMDF, es orientado a objetos. Estos objetos se encapsulan internamente, permitiendo sólo la interacción con ellos a través de una interfaz de manejadores. La infraestructura ofrece rutinas para la creación de estos objetos, recuperación de valores, y actualización de estos(WdfDeviceCreate, Get/Set, Assign/Retrieve).

Los objetos de drivers pertenecen a una jerarquía, donde el objeto raíz es el objeto WDFDRIVER, que describe el driver actual, y que es análoga al driver de objeto ofrecido por el gestor de entrada/salida. Todo objeto generado por KMDF para este driver será un hijo de este dentro de la jerarquía. Dentro de ésta podemos destacar los siguientes objetos:

- **WDFDRIVER:** Objeto raíz de la jerarquía.
- **WDFREQUEST:** Objeto auxiliar de petición.
- **WDFIORESREQLIST:** Lista de objetos de rangos de direcciones para la entrada/salida(Resource Requirements).
- **WDFIORESLIST:** Identifica un rango de direcciones para el dispositivo.
- **WDFDEVICE:** Objeto del dispositivo.
- **WDFCHILDLIST:** Lista de objetos hijos del dispositivo asociado.
- **WDFFILEOBJECT:** Objeto fichero.
- **WDFIOTARGET:** Dispositivo que debe resolver la petición de entrada/salida.
- **WDFKEY:** Elemento del registro.
- **WDFINTERRUPT:** Instancia de interrupción.

Dicha jerarquía se resume en la siguiente figura:



Es importante resaltar que se contempla el contexto de un objeto, de forma que, aunque los objetos sean opacos, éstos permiten al driver almacenar su propia información en su objeto padre para mejorar la localización de la información. De hecho, es posible disponer de más de un área de contexto, permitiendo a múltiples niveles de código interactuar con el mismo driver. De esta forma, los niveles trabajan de forma independiente.

3.6.5.10 Modelo de entrada/salida de KMDF.

El modelo de entrada/salida de KMDF utiliza los mismos mecanismos que WDM, las interfaces de WDM y la API del núcleo, de forma que el mismo puede ser observado como un driver de la WDM en sí mismo. Basándose en el modelo de WDM el driver de KMDF realizará alguna de las tres operaciones siguientes:

- Enviar el paquete de petición al gestor de entrada/salida.
- Enviar el paquete al sistema de Plug and Play y el gestor de consumo para procesarlo y modificar el estado de consumo si es requerido.
- Enviar el paquete al gestor de servicios WMI.

Estos componentes notificarán al driver los eventos registrados para dirigir la petición hacia el manejador adecuado. De haber terminado KMDF su proceso del IRP, sin haberse satisfecho la petición, KMDF optará entre completar la petición con estado STATUS_INVALID_DEVICE_REQUEST, en caso de tratarse de un driver de bus, o de función, o redirigir la petición al driver de nivel inferior en caso de utilizar drivers de filtro multinivel.

El proceso de paquetes de petición, una vez más, sigue basándose en colas (WDFQUEUE).

Esto le ofrece la oportunidad de ordenarlas como requiera

Un driver típico de KMDF crea una cola y le asocia uno o varios eventos, además de los callbacks que requiera, los estados de consumo, los métodos de resolución para la cola y si aceptará, o no, buffers vacíos.

Las tareas gestionadas por el sistema de gestión de entrada/salida de KMDF (aparte de las habituales de creación, cierre, liberación de recursos, lectura, escritura y control) son:

- La creación de peticiones, notificada a través del evento EvtDeviceFileCreate. Otra opción es crear una cola automática y registrar un callback para el evento EvtIoDefault. De no utilizarse alguno de estos métodos, KMDF devolverán un código de finalización correcta.
- En caso de liberación de recursos y peticiones de cierre, se notificarán los eventos EvtFileCleanup y EvtFileClose respectivamente, y se llamarán a sus callbacks correspondientes.

3.6.5.11 Gestor de dispositivos Plug and Play(PnP).

Este gestor ofrece la habilidad de reconocer y adaptar los cambios de configuraciones de hardware, instalando y eliminando dispositivos. Para ello se sirve de la cooperación con el hardware, su driver, y los distintos niveles dentro del sistema operativo. Además, se apoya sobre un estándar de identificación de dispositivos alojados en los distintos buses. Este estándar ofrece las siguientes capacidades al sistema de PnP:

- Reconocimiento automático de dispositivos instalados(enumerando los existentes en el proceso de inicio y detectando los añadidos y eliminados después de este).
- Alojamiento de recursos requeridos por el hardware(reuniendo los requerimientos, y asignándolos(o reasignándolos, si es necesario) a través del arbitrador de recursos.
- Carga del driver requerido en memoria. Si este driver no estuviera instalado, el subsistema de PnP en espacio del núcleo pedirá al subsistema de PnP en espacio de usuario que instale el dispositivo, con ayuda del usuario.
- Mecanismos de detección y configuración de hardware para drivers y aplicaciones.
- Almacenamiento para los estados del dispositivo.
- Gestión de dispositivos conectados en la red.

Es importante resaltar que tanto el driver, como el dispositivo deben soportar PnP. De no hacerlo alguno de los dos, el sistema puede verse comprometido. Es posible, aún así, realizar un soporte parcial del sistema PnP sobre un dispositivo que no lo soporte, si el driver si implementa los requerimientos de este sistema.

Para ofrecer soporte al sistema, debe implementarse una rutina de resolución para el mismo, otra para el sistema de control de consumo, y una rutina de añadido del dispositivo. Es importante saber que el soporte de PnP en un driver de bus, será distinto que en un driver de función o de filtro. El driver de bus debe describir de forma única los dispositivos alojados en el bus que representa, cargar su driver de función y llamar a la rutina de añadido de dispositivo.

Los drivers de función y de filtro, deben prepararse para utilizar el dispositivo en la rutina de añadido del dispositivo, sin necesidad de comunicarse con él. Para comunicarse espera a que el gestor de PnP le envíe un comando de comienzo del dispositivo para pasar el control a la rutina de resolución adecuada. Es entonces cuando se configura el dispositivo. Si una aplicación trata de abrir

el dispositivo que no ha finalizado de iniciarse recibirá un código de error indicando que el dispositivo no existe.

Una vez inicializado el dispositivo, el gestor de PnP puede enviar diversos comandos, como por ejemplo, el de eliminación del dispositivo, o reasignación de recursos. Estos comandos guiarán al driver a través de una tabla de transición de estados bien definida.

3.6.5.12 Carga, inicialización e instalación de drivers.

Existen dos tipos de inicialización, la explícita, y la basada en enumeración (utilizando los servicios de Windows). Todo driver dispone de valores de registro en la rama de servicios del conjunto de control actual. Estos valores definen el tipo de imagen, la localización del driver, los valores de control, o el orden de inicialización del servicio. Las diferencias entre la carga explicita frente a la carga a través de servicios de Windows son que el driver puede especificar un valor de comienzo para el comando boot-start (0, se carga durante la inicialización del subsistema de ejecución) o system-start (1, se carga después de la inicialización del subsistema de ejecución), y que el driver puede usar los valores de control Group y Tag para ser ordenados en la fase de inicio, pero no puede usar los valores DependOnGroup o DependOnServices. El valor de registro Group se compara con el valor de registro "HKLM\SYSTEM\CurrentControlSet\Control\ServiceGroupOrder\List" para determinar en qué orden se inicializan los grupos durante el inicio. Para refinar el orden de carga dentro del grupo se utiliza la etiqueta Tag y el registro "HKML\SYSTEM\CurrentControlSet\Control\GroupOrderList" que define la preferencia dentro del grupo. Estos valores sólo son útiles durante la carga del sistema operativo.

Las guías utilizadas para otorgar un valor de comienzo son las siguiente:

- Drivers no Plug and Play reciben el valor de carga en inicio que requieran.
- Cualquier driver(no-PnP o PnP) que requiera inicializarse durante la carga del sistema operativo recibe el valor boot-start(0).
- Si no es requerido para ejecutar el sistema operativo y el driver detecta un dispositivo que los drivers de bus no pueden enumerar, recibe el valor de system-start(1).
- Drivers no-PnP y de sistemas de ficheros que no requieren estar presentes en la ejecución del sistema operativo reciben un valor de auto-start(2).
- Los drivers PnP que no se requieren en el inicio del sistema operativo reciben un valor de demand-start(3).

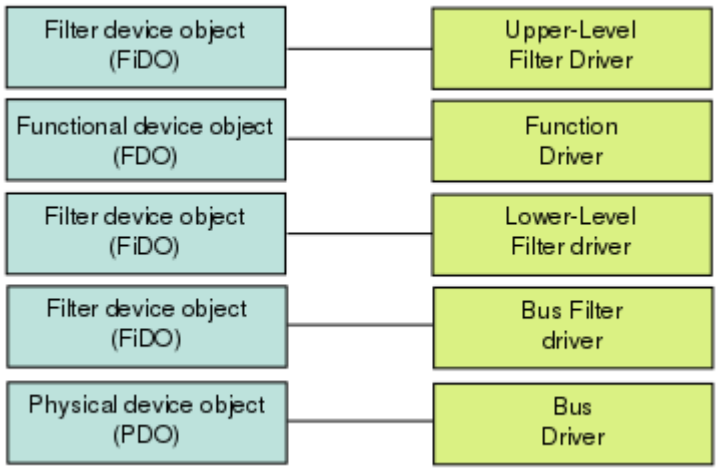
Para la enumeración de dispositivos, se hace uso de un driver virtual no-PnP de bus que ejerce de raíz, para representar el sistema, y la capa HAL, comentada anteriormente, y que se nutre de las descripciones ofrecidas por el registro para detectar el bus primario. Dentro de este bus pueden encontrarse dispositivos y otros buses(como el bus USB, PCI, PCI-e...), en los que pueden encontrarse otros dispositivos, de forma que la enumeración pueda realizarse de forma recursiva, formando un árbol de dispositivos. Los nodos en el árbol se llamarán devnodes, y contendrán objetos de dispositivos. Los buses alojados en el sistema deben cumplir la interfaz ACPI para el control de consumo. La herramienta para poder ver el árbol de dispositivos es el gestor de dispositivos(Device Manager).

La carga e inicialización de dispositivos sigue los siguientes pasos:

- El gestor de entrada/salida invoca la rutina de inicialización del driver, para todo driver que deba cargarse en el inicio del sistema. Si dicho driver tiene dispositivos que atender, el gestor de entrada/salida los enumera y notifica su presencia al gestor de PnP. Si se encuentra un dispositivo que no dispone de un driver de carga en el inicìo, se crea la estructura devnode, pero ni se carga su driver, ni se inicializa la estructura.
- Después de la carga de drivers en el inicio, el gestor de PnP recorre el árbol de dispositivos, cargando los drivers para las estructuras devnodes que no se cargaron en el paso anterior, y enumera todo dispositivo que sea atendido por el driver. En este paso, se obvia cualquier valor de precedencia para los dispositivos. Al finalizar este paso, todo dispositivo PnP está listo para usarse.
- Se comienza la carga de los drivers de carga durante el inicio del sistema, que no hayan sido cargados ya. Los dispositivos detectados por estos drivers son no numerables, y el gestor de PnP no iniciará los dispositivos hasta que los drivers enumerados sean iniciados y configurados.
- El gestor de servicios de control carga los drivers con valor auto-start.

El árbol de dispositivos servirá de guía para el gestor de Plug and Play y el gestor de consumo. Los dispositivos quedarán registrados en la entrada "HKLM\SYSTEM\CurrentControlSet\Services". La forma en que se almacenan los valores de los dispositivos es "Enumerado\Id. de dispositivo\Id. de instancia".

Los devnodes se componen de al menos dos objetos de dispositivos. Uno de ellos es un objeto de dispositivo físico(Physical device object, o PDO) que el gestor de PnP pedirá crear al driver de bus cuando encuentre un dispositivo que esté presente en éste. Otros son los objetos de filtro de dispositivos(filter device objects, o FiDOs) que ofrecen una interfaz entre el dispositivo físico y los objetos de dispositivos funcionales o FDO, que se crean en el driver y serán utilizados por el gestor de PnP para gestionar el dispositivo. Además, los objetos de función de driver, pueden crear algunos objetos de filtro que cumplan una función a nivel superior a éste.



Aunque exista una pila de objetos para controlar el dispositivo, cualquier petición puede ir dirigida a cualquier nivel, y no es necesario que pase ni por los niveles superiores, ni por los inferiores.

Para la carga de dispositivos, el gestor de PnP se apoya en el registro. Al enumerarse, el

identificador de dispositivo es enviado al gestor junto con un identificador de instancia. Con esta información, el gesto genera una instancia de dispositivo para almacenar los valores dentro de la rama de enumeración en el registro. El valor de clase(ClasssUID) se almacena en "HKLM\SYSTEM\CurrentControlSet\Control\Class".

El orden de carga de dispositivos se apoya en las siguientes reglas:

- Primero se cargan los drivers de filtro de bajo nivel especificados en la enumeración de dispositivo.
- Después se cargan los drivers de filtro de bajo nivel especificados en la clase de dispositivo.
- A continuación se cargan los drivers de función especificados en la enumeración de dispositivos.
- Posteriormente se cargan los drivers de filtro de alto nivel especificados en la enumeración de dispositivos.
- Por último se cargan los drivers de filtro de alto nivel especificados por la clase de dispositivo.

Todo driver quedará referenciado en la clave de registro "HKML\SYSTEM\CurrentControlSet\Services".

En cuanto a la instalación de drivers, de no disponerse del driver adecuado para utilizar un dispositivo, deberá solicitarse al gestor de PnP en espacio de usuario. Si se ha detectado en el inicio del sistema, se definirá su estructura devnode, pero su carga será retrasada hasta que el gestor en espacio de usuario resuelva la problemática. Varios componentes entran en juego en este proceso, algunos proveídos por el sistema, y otros por el instalador del driver.

El proceso de instalación comienza en la notificación del driver de bus al gestor de PnP. Éste busca en el registro de Windows el driver de función correspondiente, y al no encontrarlo, informa al gestor de PnP en espacio de usuario, enviándole el identificador de dispositivo. Este gestor, intentará una instalación sin la intervención del usuario, que de no tener éxito llamará al ejecutable Rundll32.exe para que se inicie la interacción con el usuario, con la correspondiente comprobación de privilegios. El instalador utilizará el sistema Setup y CfgMgr para detectar el fichero INF que guarda la información de instalación del driver. Inicialmente lo buscará en el almacén de drivers(%SYSTEMROOT%\System32\DriverStore). Si no dispone de él, se le pregunta al usuario para que lo provea, instalándolo en el almacén de drivers y ejecutando posteriormente Drvinst.exe para que se guarde constancia del nuevo driver en el registro.

Desde entonces, localizar el driver de un dispositivo, es un proceso en el cual se pide a los dispositivos de bus una lista de identificadores de hardware y de identificadores compatibles con dicho bus. Esta lista está ordenada desde el driver más específico al menos específico. En caso de encontrar ocurrencias en más de un fichero INF, el fichero donde la ocurrencia sea más específica tendrá preferencia. Si sólo se encuentra un driver compatible, puede pedirse al usuario que provea al sistema de un driver más actual.

3.6.5.13 Gestor de consumo.

El hardware que quiera utilizar el gestor de consumo debe cumplir la interfaz avanzada de configuración y consumo(Advanced Configuration and Power Interface, ACPI). Esta interfaz estándar define seis estados de consumo fundamentales, del S0(funcionamiento completo) al S5(apagado). Las características principales de estos estados son su consumo energético, el estado

del que se realiza la transición a un estado más activo, y la latencia hardware.

Del estado S1 al S3(inclusive) los estados existentes son los conocidos estados de bajo consumo(Sleeping States) en los que el consumo del dispositivo se reduce hasta casi parecer apagado, salvando la información relevante en memoria RAM y en disco para retomar las tareas desde el punto en que se dejaron.

El estado S4 es el de hibernación, en el que se almacena una imagen de la memoria RAM, comprimida, en el fichero Hiberfil.sys en el directorio raíz del sistema. Después de ello el sistema procede al apagado completo. Durante el inicio, el sistema comprueba si la imagen en dicho fichero es válida, y en caso afirmativo, el proceso Bootmgr ejecuta Winresume, que carga el contenido del fichero en la memoria, y restaura la ejecución.

Es posible que si el hardware lo soporta, se pase a un estado híbrido entre el S3 y S4 llamado hybrid sleep, en el que el sistema pasa al estado S3 almacenando un fichero de hibernación en disco. Para hacer más rápida esta operación, el fichero almacenado sólo guarda los datos que no pueden ser almacenados en disco posteriormente. Los drivers recibirán la notificación de transición al estado de hibernación S4. Si la energía se consume completamente, el sistema dispone de lo necesario para retomar las tareas en curso, en el fichero de hibernación.

Si se desea pasar de un estado de bajo consumo a otro, se hace a través del estado S0, de operación completa.

A pesar de existir estos seis estados de consumo, ACPI define 4 de ellos, siendo el estado D0 el de operación completa, y el D3 el de apagado. Deben ser los drivers de dispositivos los que definan los estados D1 y D2, cumpliéndose que, en D2 debe consumirse menos que en D1. Microsoft, junto con los fabricantes de hardware definen unas especificaciones de referencia.

El gestor de consumo, entonces, dialoga con los drivers de dispositivos para realizar su cometido. El gestor se encarga de definir las políticas adecuadas, decidiendo cuál es el estado apropiado en un momento determinado, y solicitando a los drivers que realicen las transiciones requeridas. Los factores que se tienen en cuenta para definir qué política es la adecuada son el nivel de actividad, el nivel de batería, las peticiones de apagado, hibernación y bajo consumo (dormido) de las aplicaciones, las acciones de usuario (pulsar el botón de apagado, cerrar el portátil, etc.) y la configuración del panel de control de consumo.

Al realizar la enumeración, parte de la información que se ofrece al gestor de Plug and Play son sus capacidades de consumo energético. De entre ellas se encuentra información acerca de si soporta los estados D1 y D2, sus latencias, o el tiempo requerido para las transiciones de estado. La correspondencia entre los estados que soporta el sistema y los ofrecidos por ACPI se resumen en la siguiente tabla:

System Power State	**Device Power state**
S0(operación completa)	D0(operación completa)
S1(dormido)	D1
S2(dormido)	D2
S3(dormido)	D2
S4(Hibernación)	D3(Apagado)
S5(Apagado completo)	D3(Apagado)

Cuando el gestor de consumo decide realizar una transición de estados de consumo, envía el

comando determinado a la rutina de resolución del driver de dispositivo. Para las tareas de consumo sólo un driver se designa como encargado de satisfacer las políticas de consumo. El sistema preguntará a este driver su política a través de la función PoRequestPowerIrp, notificando al resto de drivers la acción a realizar por la rutina de resolución de consumo. De esta forma, el gestor de consumo siempre sabe qué comandos están activos y en qué momento.

Aun así, los drivers pueden realizar transiciones de forma unilateral, permitiéndose realizar transiciones a estados de más bajo consumo en caso de que esté inactivo por un amplio periodo de tiempo(p.e. parar del motor de un disco duro). En el otro sentido, el driver también es capaz de utilizar los servicios proveídos por el gestor de consumo registrándose mediante la función PoRegisterDeviceForIdleDetection. Esta rutina informa de los valores de temporizadores que detectan que el dispositivo está ocioso, y su estado de consumo requerido cuando se encuentra en este estado. Para ello usa dos límites temporales principales, uno configurado por el sistema, y el otro configurado por el usuario para un óptimo aprovechamiento de los recursos. Para notificar que el dispositivo está activo utiliza la función PoSetDeviceBusy.

En ningún caso el dispositivo puede provocar una transición de estado en el sistema, puesto que el gestor de consumo notifica la transición, pero nunca pregunta si el dispositivo está listo para hacerla.

El espacio de usuario puede registrar notificaciones de estado de consumo al núcleo.

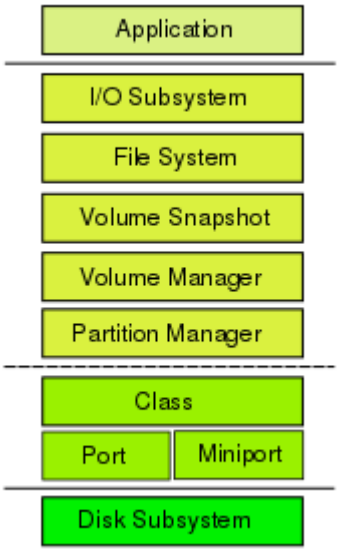
3.6.6 Subsistema de almacenamiento.

El subsistema de almacenamiento se encarga de gestionar los dispositivos de bloques. Al tratarse del mismo tipo de dispositivo utilizado en GNU/Linux, muchas características se repiten, como pueden ser la organización en bloques de 512 bytes(2048 bytes si se trata de un CD-ROM), el concepto de sectores, particiones, volúmenes(simples y multipartición), etc.

Veremos, pues, como se gestionan dichos dispositivos en el sistema operativo de Microsoft.

3.6.6.1 Drivers de disco.

Para este tipo de drivers se dispone de una pila de almacenamiento detallada en la siguiente figura:



El primer componente que requiere el uso del disco es Winload, que, propiamente dicho no pertenece a la pila representada previamente. Sin embargo, ya que requiere acceder a la información de disco para obtener los ficheros de carga del sistema operativo, está obligado a soportar el acceso a éstos. Para ello Bootmgr ofrecerá las opciones de inicio al usuario para que decida desde qué partición iniciar el sistema, para lanzar la ejecución de Winload. Éste cargará en memoria los ficheros de sistema de Windows, el registro, el núcleo Ntoskrnl.exe, y sus dependencias. Para ello se nutre del firmware del sistema, que le ofrece lo mínimo indispensable para que pueda leer el contenido de disco.

Para este tipo de drivers se ofrece una arquitectura basada en clases, puertos y minipuertos, donde los drivers de clases ofrecen funcionalidades comunes a todo tipo de dispositivo, los drivers de puerto ofrecen las funcionalidades comunes a un tipo de bus de dispositivos de almacenamiento, y el driver de minipuerto se comunicará con un controlador particular de almacenamiento.

Los drivers de clase conforman la interfaz estándar de dispositivos de Windows. Los drivers de minipuertos utilizan la interfaz de drivers de puertos, en vez de la interfaz de dispositivos, y los drivers de puertos implementan rutinas que ejercen de interfaz entre los minipuertos y Windows. De esta forma, los desarrolladores de drivers de minipuerto pueden centrarse en la lógica específica del hardware. Las funcionalidades comunes a todo disco vienen implementadas en "\Windows\System32\Drivers\Disk.sys". Otros drivers de interés son los ofrecidos por Scsiport.sys(para los buses SCSI), Ataport.sys(para los buses ATAPI) y Storport.sys(este último reemplaza Scsiport.sys con nuevas funcionalidades como la gestión de RAIDs y la comunicación por fibra óptica, entre otras.

Los drivers de bus SCSI y ATAPI implementan un gestor de peticiones llamado C-LOOK en el que el driver inserta las peticiones en una lista ordenada por número de sector al que la petición va dirigida, utilizando las funciones KeInsertByKeyDeviceQueue y KeRemoveByKeyDeviceQueue para insertar y eliminar elementos respectivamente. El driver entonces recorrerá la lista de peticiones resolviéndolas hasta que alcance el final. Puesto que durante la resolución de peticiones, habrán aparecido nuevas peticiones que atender, el driver retornará al comienzo de la lista y volverá a recorrerla. Esto provocará que la cabeza de disco haga un recorrido desde los cilindros exteriores a los interiores continuamente, en caso de que las peticiones se repartan a lo largo de todo el disco. Si no existe una mayor organización en la atención de peticiones en los drivers de bus, es debido a

que las peticiones se dirigen a vectores de almacenamiento, donde no queda claro dónde se encuentra ni el comienzo, ni el final del disco. Diferentes drivers de controladores de discos se incluyen en la instalación inicial de Windows.

3.6.6.2 Drivers iSCSI.

Especial mención requieren este tipo de drivers por dar soporte a soluciones de almacenamiento distribuido como las conocidas Storage Area Networks (SAN), o áreas de red dedicadas al almacenamiento. Para ello integran el protocolo SCSI con el protocolo TCP/IP. Las SAN suelen utilizar fibra óptica para comunicar los distintos nodos, pero puede utilizarse Gigabit Etherrnet para crear redes de almacenamiento más baratas y con buen rendimiento.

El componente principal que gestiona este almacenamiento es el invocador de software iSCSI (iSCSI Software Initiator) de Microsoft, y sus componentes principales se listan a continuación:

- **Invocador:** Compuesto del driver Storport comentado anteriormente y el driver de minipuerto iSCSI (Msiscsi.sys) implementa el driver necesario para ofrecer soluciones SAN a través de Ethernet.
- **Servicio del invocador:** Implementado en "\Windows\System32\Iscsiexe.exe", descubre y gestiona la seguridad de los invocadores, además de los inicios de sesión y finalizaciones de comunicación con otros nodos.
- **Gestor de aplicaciones:** Iscsicli.exe es un cliente en consola que permite las conexiones seguras a las soluciones iSCSI, además de panel de control requerido para estos dispositivos.

3.6.6.3 Drivers de entrada/salida a través de múltiples caminos.

Servidores que requieren un alto grado de disponibilidad utilizan este tipo de soluciones. Se trata de dispositivos de almacenamiento que disponen de más de un camino para ser alcanzados, permitiendo que se produzcan fallos en las comunicaciones a través de algunos de sus caminos, sin que necesariamente impida que el almacenamiento sea alcanzado por otras vías. En caso de que el sistema operativo no ofreciera soporte a estas situaciones, un mismo disco sería considerado dos distintos por el hecho de poder accederse por diferentes caminos. Este soporte se apoya en drivers propios, o de terceros llamados módulos específicos de dispositivo(device-specific modules), que especifican los detalles de los múltiples caminos.

El soporte de múltiples caminos es ofrecido por la pila de almacenamiento MPIO(Multiple Path Input/Oputput, on entrada/salida por múltiples caminos) y es el driver Disk.sys el que se responsabiliza del dispositivo, localizando el módulo específico para éste. Además avisará al driver Mpio.sys, que representa el driver de bus para dispositivos de múltiples caminos, de la presencia del dispositivo. En esta situación tenemos un total de 3 pilas de dispositivos de almacenamiento, dos representando los caminos físicos, y una representando el dispositivo. Cuando una petición alcanza el disco, el módulo DSM se encargará de decidir por qué camino dirige la respuesta.

3.6.6.4 Objetos de dispositivos de disco.

El driver de clase para dispositivo de disco es encargado de crear los objetos que los representan. En el espacio de nombres serán representados con la nomenclatura

"\Device\HarddiskX\DRX", donde X representa un número en cuestión. Para mantener la compatibilidad con sistemas previos se crean enlaces simbólicos con el formato ofrecido por windows NT 4, que formateaba los nombres con la nomenclatura "\Device\HarddiskX\PartitionX" para referirse a "\Device\HarddiskX\DRX". También se generan enlaces hacia el objeto que representa al dispositivo físico, teniendo enlaces del tipo "\GLOBAL??\PhysicalDriveX" apuntando a "\Device\HarddiskX\DRX", que, como comentamos previamente, permitían exportar los dispositivos al espacio de usuario para que las aplicaciones los utilizaran bajo demanda.

3.6.6.5 Discos básicos y discos dinámicos.

En las placas bases estándar, los servicios mínimos de entrada/salida son ofrecidos por una pequeña memoria FLASH donde se almacena un fichero binario llamado BIOS (Basic Input/Output System). En este tipo de hardware, el primer sector del disco principal contiene el registro maestro de inicio (Master Boot Record, o MBR). Cuando la BIOS se prepara para iniciar el sistema operativo, acude a este registro para conseguir la información necesaria acerca de los volúmenes existentes y de cuál debe iniciar. Los volúmenes se registran en la tabla de particiones, dentro del registro maestro de inicio. En una tabla de particiones de este tipo pueden definirse hasta cuatro particiones principales, con sus correspondientes tipos (usualmente FAT32 y NTFS).

Pueden añadirse más particiones definiendo particiones extendidas, que contienen su propio registro maestro de inicio, y son llamados dispositivos lógicos. Teóricamente, existe un límite de particiones extendidas de cuatro por disco. Sin embargo, Windows es capaz de controlar particiones extendidas anidadas, permitiendo que ese límite desaparezca. Estas características conforman la estructura de un disco básico.

Hoy en día existe una iniciativa propuesta por Intel para promover la estandarización de su plataforma de firmware extendido (Extended Firmware, o EFI). Esta plataforma conforma un sistema operativo mínimo almacenado en memoria FLASH, que sirve para cargar herramientas básicas de diagnóstico, y la carga del sistema operativo. Este tipo de firmware define un nuevo tipo de tabla de particiones basado en identificadores únicos globales (Global Unique Identifiers, o GUIDs), llamado tabla de particiones basado en GUID (GUID Partition Table, o GPT). El objetivo principal de dicho tipo de tabla de particiones es eliminar las restricciones (sobre todo a la hora de redimensionar volúmenes) de las antiguas tablas de particiones basadas en registros maestros de inicio. Este tipo de tablas de particiones conforman lo que se conoce como discos dinámicos.

3.6.6.6 Espacio de nombres de volúmenes.

Éste es el mecanismo de Windows que se encarga de asignar letras a los objetos de los dispositivos de bloques, permitiendo un acceso familiar a las aplicaciones, y ofreciendo la posibilidad de montar y desmontar los volúmenes. En los próximos apartados realizaremos un resumen de sus mecanismos más importantes y cómo se gestionan este tipo de operaciones.

3.6.6.6.1 Gestor de montaje.

El driver del gestor de montaje(Mountmgr.sys) es el encargado de asignar las letras a los volúmenes creados después de la instalación de Windows. Las asignaciones se almacenan en "\HKLM\SYSTEM\MountedDevices", y su clave tiene la forma "\??\Volume{X}" donde X es el GUID del volumen. Los valores asociados tienen la forma "\DosDevices\X:\" siendo X la letra

asignada a la unidad. Todo volumen debe tener una entrada en el espacio de nombres, pero no tiene por qué tener asignada una letra de unidad.

Para los discos básicos, el registro almacena las letras de sus volúmenes, los nombres en estilo Windows NT 4, y la dirección de comienzo del volumen asociado. En el caso de discos dinámicos, sólo se almacena el GUID. Durante el inicio, el gestor de montaje se registra en el sistema de Plug and Play, de forma que cuando un dispositivo es identificado como un volumen, recibe la notificación pertinente del sistema de PnP. Con esta notificación, el gestor de volúmenes determina el GUID para localizar la letra de la unidad almacenada en el registro de Windows. De no encontrarla, sugerirá una y la almacenará en el registro. En el caso de los discos básicos, no se devuelve sugerencia. De no existir ésta, se utiliza la primera letra no asignada a otro volumen, se define su asignación, se crean los enlaces simbólicos necesarios y finalmente se actualiza el registro de Windows. Si se diera el caso de que todas las letras estuvieran utilizadas, no se ofrecería ninguna asignación.

Es necesario resaltar que el GUID utilizado por el gestor de volúmenes no necesariamente debe coincidir con el GUID real del volumen, permitiéndose así que los discos básicos dispongan de un GUID que su tabla de particiones no le asigna inicialmente.

3.6.6.6.2 Puntos de montaje.

Los puntos de montajes permiten enlazar volúmenes en directorios, dentro de los volúmenes NTFS. Éste es el mecanismo ofrecido por Windows, para manejar los volúmenes que no tuvieron la suerte de disponer de una letra asignada. Para ello utiliza una tecnología de reprocesado de puntos de montaje.

El punto reprocesado, es un bloque de datos arbitrario, con una cabecera fija que Windows asocia a un fichero o directorio de un volumen NTFS. Puede ser una aplicación, o el mismo sistema, quien defina el formato y comportamiento de estos puntos. Dentro de la cabecera se almacenará una etiqueta que determine si fue definida por una aplicación, o por el sistema, el tamaño, y el significado del punto reprocesado. De ser una aplicación la que lo define, debe proveer de un driver de filtro para revisar los valores devueltos por el volumen al operar con sus ficheros. Estos valores son devueltos en caso de encontrar una coincidencia de fichero o directorio en la operación realizada.

Este sistema se implementa haciendo uso del sistema de ficheros NTFS, el gestor de entrada/salida, y el gestor de objetos. Este último inicia el procesado de localizaciones, sirviéndose del gestor de entrada/salida como interfaz con el sistema de ficheros. Sin embargo, el gestor de objetos debe volver a realizar las operaciones para las que el gestor de entrada/salida devuelve un código de estado. El gestor de entrada/salida implementa la modificación de la localización y los posibles puntos reprocesados necesarios para que el sistema de ficheros pueda finalizar la operación. Puede entenderse que el gestor de entrada/salida funciona como un driver de filtro para el sistema de reprocesado de puntos de montaje.

El uso más utilizado de esta tecnología, es la creación de enlaces simbólicos en NTFS. Si el gestor de entrada/salida recibe un código de estado del sistema de reprocesado, y el fichero o directorio no está asociado con un punto de montaje reprocesado, ningún driver de filtro se hará cargo de la operación, devolviendo un código de error al gestor de objetos.

Los puntos de montajes son, pues, puntos reprocesados que almacenan un nombre de volumen. Es posible utilizar el gestor de discos para crear puntos de montaje.

El gestor de montaje alberga una base de datos en la que almacenará los puntos de montajes definidos para cada volumen. Ésta se aloja en el directorio “System Volume Information” de

cualquier volumen NTFS.

3.6.6.6.3 Montaje de volúmenes.

Aunque un volumen disponga de letra correspondiente, el contenido del volumen puede estar organizado por un sistema de ficheros no soportado por Windows. Para reconocer el volumen, un sistema de ficheros debe proclamar que la partición le pertenece, proceso que sucederá la primera vez que el núcleo, un driver, o una aplicación accedan a un fichero del volumen. Una vez proclamada la pertenencia, el gestor de entrada/salida redirige todos los paquetes de peticiones al driver correspondiente. El montaje implica a tres componentes: el registro del driver de sistema de ficheros, los bloques de parámetros de volumen(Volume Parameters blocks, o VPBs) y las peticiones de montajes.

El gestor de entrada/salida supervisa el proceso de montaje, y es consciente de que existen los drivers de sistemas de ficheros puesto que son registrados en la inicialización de estos mismos, a través de la función IoRegisterFileSystem. Al registrarse, una referencia será almacenada en una lista que el gestor de entrada/salida utiliza durante las operaciones de montaje.

Los bloques de parámetros de volúmenes se utilizan como enlaces entre el objeto que representa un volumen, y el objeto de dispositivo. Si su referencia está vacía, ningún sistema de ficheros habrá montado el volumen, y el gestor de entrada/salida deberá comprobar la integridad de la referencia si se utiliza la operación open sobre un fichero del volumen.

La convención utilizada por los sistemas de ficheros para comprobar si el volumen fue montado con su formato, es examinar el registro de inicio del volumen, almacenado al comienzo de éste. En los sistemas de ficheros de Microsoft, este registro contiene un campo que almacena el sistema de ficheros utilizado, aparte de información necesaria para que el driver de sistema de ficheros encuentre ficheros de metadatos críticos en el volumen.

Si el sistema de ficheros afirma que el volumen le pertenece, el gestor de entrada/salida rellena el bloque de parámetros de volumen y pasa la petición de apertura con el resto de la localización al driver del sistema de ficheros. Este finalizará la petición utilizando su formato para interpretar los datos que el volumen contiene. Una vez el montaje rellena el bloque de parámetros, el gestor de entrada/salida puede resolver el resto de aperturas dirigidas al volumen montado. Si ningún sistema de ficheros proclama la pertenencia del volumen, el sistema de ficheros en bruto(Raw) lo hará, y toda petición de apertura de fichero fallará.

Para evitar tener todos los drivers de sistemas de ficheros cargados, aunque no dispongan de volúmenes que gestionar, Windows intenta minimizar el uso de memoria delegando el reconocimiento del sistema de ficheros en un driver llamado reconocedor de sistema de ficheros (File System Recognizer). Éste conoce lo estrictamente necesario para reconocer si el sistema de ficheros estudiado se asocia a uno de los soportados por Windows. Durante el inicio, el reconocedor es registrado como un driver de sistema de ficheros, y es llamado por el gestor de entrada/salida cada vez que se solicita una operación de montaje. Es entonces cuando el reconocedor cargará el driver del sistema de fichero en memoria y redirigirá las peticiones a éste, dejando que el sistema de ficheros proclame la pertenencia del volumen.

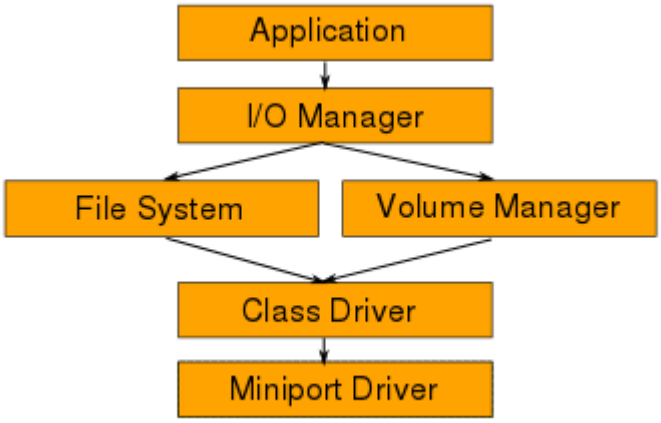
Cuando se solicita una comprobación de consistencia de sistema de ficheros a través de la aplicación Chkdsk, durante el comienzo del inicio, casi todos los volúmenes son montados. La aplicación que ejerce dicha comprobación es Autochk.exe, y es invocada por el gestor de sesiones (Smss.exe) cuando es especificada para ejecutarse en el inicio a través de la clave de registro "HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\BootExecute". Chkdsk accede a todos los dispositivos para ver si requieren la comprobación de consistencia.

El montaje de un mismo dispositivo puede suceder en varias ocasiones cuando se hace referencia a dispositivos desmontables. Si se vuelve a montar un dispositivo previamente montado se le pregunta por su identificador de volumen. Si este ha cambiado, lo desmontará para remontarlo posteriormente.

3.6.6.6.4 Operaciones de entrada/salida de volúmenes.

Cuando el sistema de ficheros recibe una petición de entrada/salida, delega en el gestor de volúmenes para transferir datos desde y hacia el disco donde reside el volumen. En el proceso de montaje, el sistema de fichero recibe la referencia del objeto de volumen con el que enviará las peticiones al gestor de volúmenes. Sin embargo, si una aplicación lo requiere, puede obviar el sistema de ficheros y acudir directamente al objeto de volumen para cumplir sus labores(por ejemplo programas de restauración de ficheros eliminados).

Cuando una petición se envía al objeto de volumen, el gestor de entrada/salida lo redirige al gestor de volúmenes que creó el objeto de dispositivo. El gestor de volúmenes debe entonces traducir el desplazamiento para localizar el fichero dentro del volumen, a un desplazamiento dentro del disco. Este desplazamiento será enviado al driver de clase de disco, que a su vez, delegará en el driver de minipuerto. Este último se encarga de resolver la petición de entrada/salida finalizando la operación. El panorama se resume en la siguiente figura:



3.6.6.7 Servicio de discos virtuales.

Los desarrolladores de productos de almacenamiento, facilitan una serie de herramientas para gestionar sus dispositivos, que, como cabe esperar, suponen una dificultad a salvar por los administradores de sistemas. En ocasiones, es necesario que aprendan múltiples interfaces para desarrollar aplicaciones específicas debido a que las herramientas de Windows no son capaces de resolver las peticiones.

Para facilitar este tipo de labores, el sistema ofrece el servicio de discos virtuales, que supone una capa de abstracción que unifica el almacenamiento a alto nivel. De esta forma, el administrador puede utilizar los distintos dispositivos, sirviéndose de una interfaz estándar.

Las interfaces ofrecidas por este subsistema son dos, una para proveedores de software, y otra para proveedores de hardware:

- La interfaz para proveedores de software supone una interfaz con el alto nivel, y trabaja con discos, particiones y volúmenes. Ofrece operaciones como la creación, redimensionado, y eliminación de volúmenes, además de crear y destruir unidades réplicas de otras unidades, y crear y formatear las unidades. Las aplicaciones deben registrarse en la clave "HKLM\SYSTEM\CurrentControlSet\Services\Vds\SoftwareProviders". Se ofrecen interfaces para trabajar con unidades dinámicas o básicas.
- Los proveedores de hardware deben implementar su software como librerías dinámicas (DLLs) y registrarlas en la clave "HKLM\SYSTEM\CurrentControlSet\Services\Vds\HardwareProviders". Dichas librerías se encargarán de traducir los comandos independientes del almacenamiento proveídos por VDS, a comandos específicos del almacenamiento que ofrecen.

Si una aplicación requiere la API de VDS, y el servicio no se encuentra en funcionamiento, Svchost recibirá la petición e iniciará el servicio (Vdsldr.exe), el cual, gestionará las peticiones requeridas y, en la finalización de comunicaciones con la API, terminará su ejecución.

3.6.7 Subsistema de redes.

Dentro de los componentes básicos que integran una red, nos encontramos con los servicios, APIs, los protocolos y los adaptadores de red. Estos componentes se estratifican para formar una pila de red. Las pilas de red más conocidas hoy día son TCP/IP, de la cual existe implementación, aunque su diferenciación en capas no llega a ser tan clara, y el modelo OSI, del cual no existe implementación pero se utiliza como modelo de diseño para cualquier tipo de redes. Puesto que posteriormente estableceremos una analogía entre los subsistemas de Windows y el modelo OSI, realizaremos un breve repaso de los conceptos más importantes.

3.6.7.1 Modelo OSI.

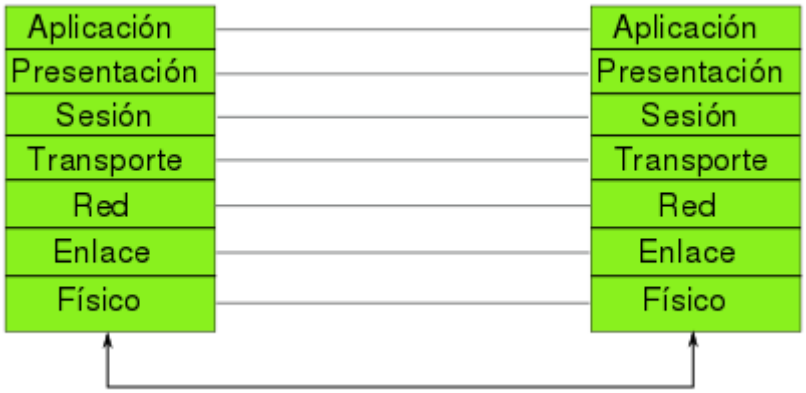
Con el objetivo de estandarizar e integrar el software de red de las grandes corporaciones, en 1984 la Organización Internacional de Estandarización(International Organization for Standarization, o ISO), definió el modelo de envío de mensajes, conocido como modelo de referencia de interconexión de sistemas abierto(Open System Interconnection Reference Model).

El modelo define siete niveles, de los cuales, uno es un nivel hardware, y los seis restantes son software. Cada nivel define una serie de reglas y condiciones para entender qué quiere transmitirle el mismo nivel, ubicado en una máquina distinta. La comunicación siempre se produce entre niveles homónimos. Cada uno de los niveles cumple un objetivo claro y diferenciado dentro de la pila, disponiéndose de los siguientes niveles:

- **Aplicación:** Gestiona la información que requiera transmitirse entre dos aplicaciones de red, incluyendo sus funciones, comprobaciones de seguridad , identificación de máquinas e inicio del intercambio.
- **Presentación:** Se encarga del formato de los datos transmitidos, como el formato de fin de línea, la compresión, o el cifrado utilizado.

- **Sesión:** Gestiona la conexión entre las aplicaciones que colaboran a través de la red, definiendo la sincronización a alto nivel, y la monitorización de las aplicaciones que utilizan la comunicación.
- **Transporte:** fragmenta los mensajes en paquetes, les asigna un número de secuencia para garantizar su orden, y ensambla los paquetes en mensajes en el destino. Independiza completamente al nivel de sesión de cualquier cambio realizado en el hardware.
- **Red:** dirige los paquetes, realiza el control de congestión, e intercambia paquetes entre distintas tecnologías. Éste es el nivel más alto que entiende de topologías de red, con sus reglas y limitaciones correspondientes.
- **Enlace:** Transmite tramas de datos, controla y resuelve los errores de envío, retransmitiendo en caso de pérdidas y colisiones.
- **Físico:** Se encarga de la transmisión de bits brutos a través del medio de comunicación.

La comunicación entre dos máquinas puede resumirse en la siguiente figura:



3.6.7.2 Componentes de red de Windows.

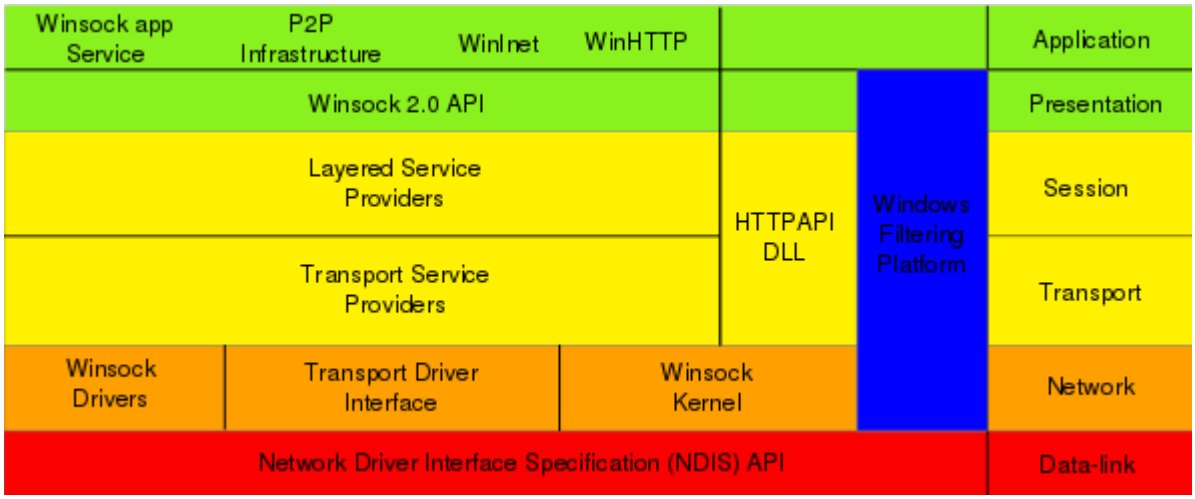
Como cabe esperar, los componentes de red de Windows no se ajustan adecuadamente al modelo OSI, de modo que la correspondencia entre niveles es aproximada, y sólo pretende dar una idea de cómo se estructura la comunicación en el sistema. Los componentes relevantes son los siguientes:

- **APIs de red:** interfaz independiente del protocolo de red que facilita la comunicación de aplicaciones.
- **Clientes de la interfaz de drivers de transporte:** antiguos drivers en espacio del núcleo para definir la parte de la API de red que funcionaba en este espacio. Sus paquetes de peticiones(IRPs) se formatean acorde a la interfaz de driver de transporte(Transport Driver Interface, o TDI), interfaz de la que adquieren el nombre, y que supone un estándar de programación para los drivers en espacio de núcleo.
- **Drivers de protocolo en espacio del núcleo:** Entre ellos podemos encontrar los drivers de la interfaz TDI, los drivers de protocolos de la especificación de interfaces de drivers de red(Network Driver Interface Specification protocol drivers, o NDIS protocol drivers), y los drivers de protocolos. Su objetivo es aceptar y procesar las peticiones en forma de IRPs de

los clientes de la interfaz de drivers de transporte. Esto puede requerir distintas tareas, como comunicar con otra estación, o pedir a los drivers de la interfaz TDI que añadan una cabecera. Para ello, se implementa el paso de mensajes entre la infraestructuras de este subsistema, haciendo transparente ciertas tareas como su segmentación, ensamblado, secuencia y reconocimiento.

- **Núcleo Winsock:** Interfaz en espacio de núcleo, independiente del nivel de transporte, que reemplaza los antiguos mecanismos de la interfaz de drivers de transporte. El núcleo Winsock ofrece una interfaz de programación utilizando sockets, al igual que la ofrecida en el espacio de usuario, pero permitiendo algunas características más como la comunicación asíncrona de operaciones de entrada/salida. Esta infraestructura añade el soporte de IP versión 6(IPv6).
- **La plataforma de filtro de Windows:** ésta se compone de un conjunto de APIs y servicios de sistema que proporcionan las capacidades necesarias para crear aplicaciones de filtrado de paquetes, procesando los diferentes niveles de la pila de red de Windows. Además, la información puede ser monitorizada, filtrada o modificada antes de alcanzar su destino.
- **Drivers de funcionalidades para la plataforma de filtro de Windows:** estos drivers se limitan a añadir funcionalidades no contempladas en la plataforma de filtro de Windows.
- **Librería NDIS:** Ndis.sys permite encapsular los dispositivos de red, ocultando sus peculiaridades del entorno del núcleo de Windows. Exporta también, funciones para que los clientes de la interfaz de drivers de transporte puedan utilizarlas.
- **Drivers de minipuerto para NDIS:** estos drivers se encargan de ofrecer una interfaz intermedia entre los clientes de la interfaz de drivers de transporte y los dispositivos de red. Se escriben de forma que queden encapsuladas tras la interfaz NDIS. Ésta no procesa IRPs, sino que registra una tabla de funciones que se corresponden con las funciones que NDIS exporta a los clientes de la interfaz de drivers de transporte. Para resolver las peticiones recibidas por la interfaz NDIS, los drivers acudirán a la capa de abstracción de hardware.

En la siguiente figura se resume la correspondencia de subsistemas con OSI.



La zona verde de la imagen se corresponde con las APIs de red, la zona amarilla son los clientes de la interfaz de drivers de transporte y del núcleo Winsock, en la zona naranja se dispone del núcleo Winsock, la interfaz de drivers de transporte(TDI) y los antiguos drivers de Winsock, y

finalmente en la zona roja podemos ver la API NDIS y sus drivers.

3.6.7.3 APIs de red de Windows.

Existen varias APIs distintas dentro del entorno de red de Windows, y cada una tiene sus características particulares, dejando en manos del desarrollador que decida cual es la API más adecuada para solucionar su problema. Estudiaremos las más importantes de cara al objetivo del proyecto.

3.6.7.3.1 Windows Sockets(Winsock).

Esta API es la implementación de Microsoft de los conocidos sockets de la Berkeley Software Distribution(BSD, una de las distribuciones del sistema Unix más conocidas). Gracias a esta API, traducir a Windows una aplicación para Unix que utilizara la red, se hacía de forma directa. Sus sucesivas versiones incluyen la mayor parte de las funcionalidades de los sockets originales, añadiendo mejoras específicas de Microsoft. Soporta comunicaciones fiables orientadas a conexión y comunicaciones no orientadas a conexión. Su versión 2.2 permite disponer de la entrada/salida asíncrona que mejora el rendimiento y escalabilidad de frente a los sockets de BSD.

Sus características principales son:

- Soporte de aplicaciones asíncronas y vectorizadas.
- Soporte de calidad del servicio (Quality of Service), lo que permite a las aplicaciones negociar la latencia y ancho de banda requerido.
- Capacidad de soportar protocolos desconocidos por Windows.
- Soporte integrado de espacios de nombres distintos a los definidos por el propio protocolo utilizado por la aplicación.
- Soporte de mensajes multipunto.

Dentro de su modo de operación, el cliente, el primer paso que se realiza es llamar a una función de inicialización de la infraestructura. Posteriormente, puede crearse un socket que representará la comunicación con el nodo destino. Pueden utilizarse las funciones getaddrinfo y freeaddrinfo para obtener y liberar, respectivamente, la información relativa al servidor al que se quiere conectar. Getaddrinfo ofrece una lista de direcciones IP asignadas al servidor, que el cliente recorrerá en el momento de la conexión, una por una, hasta que consiga establecerla, asegurando que el cliente se conecte a la dirección más apropiada y eficiente soportada por el sistema. Obtenida la dirección, pasará a realizarse una conexión a través de la función connect. Desde este momento, el cliente puede servirse de las funciones send y recv para enviar y recibir información a través del socket. Si se desea utilizar la interfaz no orientada a conexión, puede utilizarse las funciones sendto, and recvfrom.

Por el lado del servidor, también se llamará a una función de inicialización de la infraestructura, que ofrece las herramientas para crear el socket. Este socket será asociado a una dirección IP utilizando la función bind. Si el servidor utiliza comunicaciones orientadas a la conexión se usará la función listen indicando información como el fichero para almacenar información de las conexiones realizadas, o el número máximo de conexiones. Posteriormente, la función accept permitirá la conexión de clientes, y ésta finalizará tan pronto como se establezca una conexión. En este momento se dispone de un socket de conexión que representa la comunicación en

el lado del servidor, con el cliente, y puede utilizarse la función send y recv para intercambiar información. Si se quisiera establecer comunicación sin conexión, sencillamente no se haría uso de la función accept, y se utilizaría las funciones send y recv, para satisfacer la comunicación.

Existen un par de funciones que merece la pena mencionar para mejorar el rendimiento de las comunicaciones. Éstas son AcceptEx y TransmitFile. AcceptEx, cumple las funciones de accept, ofreciendo la dirección del cliente, y su primer mensaje, permitiendo procesarlo con mayor presteza. Además permite el procesado de múltiples conexiones gestionando grandes volúmenes de peticiones. TransmitFile permite el uso de la cache del sistema para enviar ficheros a través de la red.

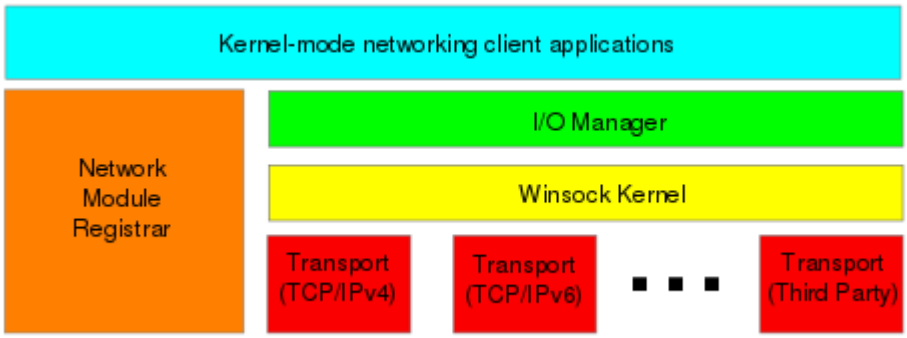
Los proveedores de servicio de transporte son el mecanismo que permite soportar protocolos desconocidos. Éstos interactúan con las interfaces Winsock para ofrecer funciones como ejercer de proxy, ofrecer espacios de nombres, o mejorar las infraestructuras de resolución de nombres. Los proveedores deben registrarse en la interfaz de proveedores de servicios de Winsock(Winsock Service Provider, WSPI), para pasar a utilizar las funciones propias del proveedor, que debe implementar las funciones del socket. No se restringe cómo las implementa.

La implementación de Winsock se compone de una interfaz para el espacio de usuario, ofrecida por la librería dinámica Ws2_32.dll, que proporciona los servicios para acceder a los espacios de nombres y a los proveedores de servicios registrados. Los proveedores que ofrece Microsoft se recogen en la librería Mswsock.dll, que se nutre de las librerías de apoyo de Winsock para comunicarse con el driver en espacio de núcleo.

Para integrarse con el modelo entrada/salida de Windows, los sockets son representados mediante un fichero, requiriendo un driver de sistema de ficheros. Para ello utiliza los servicios del driver de función auxiliar(Ancillary Function Driver), incluida en Msafd.dll, para implementar las funciones de los sockets. Éste es un cliente de la interfaz de proveedores de la capa de transporte de red, y genera paquetes de peticiones para la interfaz de drivers de transporte(TDI IRPs).

3.6.7.3.2 Núcleo Winsock.

Para permitir que los drivers de espacio de núcleo tengan acceso a la API de red ofrecida al espacio de usuario, se implementa esta interfaz basada en sockets. Dicha infraestructura reemplaza la antigua interfaz de drivers de transporte, presente en versiones anteriores de Windows, manteniendo la interfaz de proveedores de servicios de la interfaz de transporte de drivers. Ofrece mejor rendimiento y escalabilidad, que esta última gracias a que utiliza menos en los mecanismos internos del núcleo, para usar una semántica más parecida a la de los sockets de Unix. Además pretende utilizar todas las funcionalidades de las últimas tecnologías ofrecidas por la pila de red TCP/IP de Windows, como por ejemplo, el registro de módulos de red. La estructura interna queda de la siguiente manera:



Su implementación, como la de la API Winsock estudiada anteriormente, está implementada en los drivers de función auxiliares, y su infraestructura es responsable de la API para proveedores de servicio. Se encarga, por tanto, de ofrecer soporte para distintos protocolos. El subsistema se comunica con los drivers de espacio de núcleo que se comunican a su vez con el espacio de usuario, llamados drivers de aplicación del núcleo Winsock. Entre ambos subsistemas resuelven las operaciones de red. El núcleo Winsock notificará eventos asíncronamente a los drivers de aplicación.

Los drivers de aplicación deben registrarse en el subsistema del núcleo Winsock utilizando el registro de módulos de red, o utilizando las funciones propias del núcleo Winsock, permitiéndole saber cuándo este último está disponible para enviarle su tabla de funciones de resolución, que describe su proveedor, y su implementación de la API del núcleo Winsock. Funciones de esta API son, por ejemplo, WskSocket, WskAccept, WskBind, etc, que comparten semántica con las interfaces Winsock del espacio de usuario. Su principal diferencia con el espacio de usuario es que define cuatro tipos de socket:

- Básicos, que ofrecen y recuperan información del nivel de transporte.
- De escucha, que sólo aceptan conexiones entrantes.
- De datagramas, utilizados para enviar y recibir datagramas.
- Orientados a conexión, que soportan toda clase de funcionalidades para enviar y recibir datos.

El núcleo Winsock ofrece notificaciones del estado de la red a las aplicaciones, de forma que el subsistema se encarga de controlar el enlace, y el cliente sencillamente recibe la notificación del evento. Se proveen de diversas rutinas para gestionar eventos.

3.6.7.3.3 APIs de acceso web.

Las APIs de acceso web ofrecen infraestructuras de cliente y servidor con objeto de aislar al desarrollador de los detalles necesarios para la comunicación utilizando protocolos de nivel de aplicación. Los protocolos ofrecidos son FTP y HTTP, y en el cliente se dispone de la API de internet de Windows(WinInet), que se encarga de ofrecer ambos protocolos, y WinHTTP, que ofrece otra alternativa, en ocasiones más adecuada, para interactuar con el protocolo HTTP. La interfaz HTTP Server, posibilita el desarrollo de servidores web.

WinInet ofrece servicios de consistencia de cookies, servicios de conexión automática, cache de almacenamiento de ficheros en cliente, y gestión de credenciales automáticas. Esta API es utilizada por el explorador de Windows e Internet Explorer.

WinHTTP, en su versión 6.0, ofrece servicios de protocolo HTTP 1.1, al igual que WinInet, aunque su API está orientada al uso conjunto con la API HTTP Server. La aplicaciones desarrolladas sobre esta API disponen de mayor escalabilidad y funcionalidades de seguridad no ofrecidas por WinInet, como la suplantación de identidad en hilos de ejecución. Otras de sus funcionalidades son el troceado de transferencias, el listado de proveedores de servicio para la identificación por SSL, la solicitud de certificados y la reserva de rangos de puertos.

HTTP Server ofrece la API del lado servidor que comunica con la interfaz WinHTTP, atendiendo las peticiones de ésta. Ofrece soporte de SSL, caches de almacenamiento, modelos de entrada/salida síncronos y asíncronos, direccionamiento por IP versión 4 e IP, versión 6. Se accede a dicha interfaz por la librería dinámica Httpapi.dll, que se apoya en el driver en espacio del núcleo Http.sys. Para las funciones de cache, la interfaz se sirve del gestor de memoria, alojando en ésta las páginas de memoria solicitadas, hasta que una aplicación las invalide o expire un temporizador opcional asociado a éstas. Existen ciertas configuraciones de la comunicación que se ofrecen al usuario, como las políticas de seguridad, el balanceo de carga, el registro de sucesos, los límites de conexión, la gestión de cache de respuestas y la comprobación de certificados. Su aplicación por excelencia es el servicio de información de internet(Internet Information Service, IIS), que ejerce de servidor web, en los entornos Windows.

3.6.7.3.4 Otras APIs de red de Windows.

Existen diversas APIs de red de propósito general que resumiremos en dicho apartado, puesto que detallarlas ampliamente excede de los objetivos del proyecto.

La primera de ellas es la API de llamada a procedimientos remotos(Remote Procedure Call, o RPC) que se encarga de ofrecer la resolución de algunas funciones o tareas en máquinas remotas. Esta resolución se realiza de forma transparente permitiendo al desarrollador implementar su solución y decidir qué funciones deben ejecutarse localmente, y qué funciones se ejecutan de forma remota. Esta infraestructura hace uso de otras APIs como la API Winsock, o la API de tuberías para el envío de las funciones y parámetros, en forma de paquetes, requeridos para la ejecución remota.

Las APIs de tuberías, junto con los buzones, son APIs adecuadas para la comunicación entre procesos(Inter Process Communication, o IPC). Las tuberías ofrecen una comunicación bidireccional y fiable, mientras que los buzones son unidireccionales y no fiables. Estos últimos mejoran la eficiencia en servicios de múltiples receptores(Broadcast).

NetBIOS, es el sistema de entrada/salida básico de red(Network Basic Input/Output System), y es la API más utilizada en aplicaciones de red, aunque Microsoft desaconseja su uso en favor de Winsock y las APIs de tuberías. Permite conexiones fiables, u orientadas a conexión, y no fiables, o no orientadas a conexión. La API es soportada por el protocolo TCP/IP de Windows.

El servicio de transferencia inteligente en segundo plano(Background Intelligent Transfer Service, o BITS), ofrece una interfaz simple para las tareas de transferencia de ficheros. Las operaciones ofrecidas son para descargar, cargar, y confirmar la carga. Este subsistema continúa las tareas aunque el programa que iniciara la transferencia haya finalizado su ejecución, tan solo requiriendo que la conexión permanezca activa. Si la transferencia falla, se encarga de almacenar la información necesaria para retomarla tan pronto como le sea posible.

La infraestructura P2P, ofrece soporte flexible a la conocida tecnología que su nombre indica, para transferir información entre máquinas iguales. Soporta el uso de nodos y eventos, conocido como la tecnología de grafos P2P, proveedores de espacios de nombres para la resolución de nombres de nodos, la agrupación de dichos nodos, y la gestión de identificación. Esta infraestructura es similar a la ofrecida por la interfaz de colaboración P2P de Microsoft, para

facilitar la implementación de aplicaciones de colaboración en grupo.

La API COM de Microsoft define componentes definidos en módulos, como objetos exportables, y ofrece servicios para manipularlos directamente. La interfaz DCOM añade la funcionalidad de que dichos objetos no tengan la necesidad de residir en la máquina local, facilitando el envío de objetos a través de la red de forma transparente.

La cola de mensajes es una plataforma de propósito general para facilitar la comunicación en aplicaciones distribuidas. Su servidor es un repositorio en el que los transmisores encolarán sus mensajes, y, tan pronto como los clientes puedan, desencolarán dichos mensajes para su procesado. No se requiere realizar conexiones en ninguna dirección, puesto que el servicio es no fiable.

La arquitectura Universal de Plug and Play, estudiada anteriormente, es una infraestructura P2P, distribuida, que utiliza la arquitectura de red existente (a través de TCP/IP y servicios Web) para localizar y configurar nuevos dispositivos. Esto permite conectar dinámicamente a una red, obtener una IP y ofrecer sus servicios a través de la red. Su extensión PnP-X descubrirá los dispositivos existentes en la red para integrarlos a través del gestor de Plug and Play en el sistema.

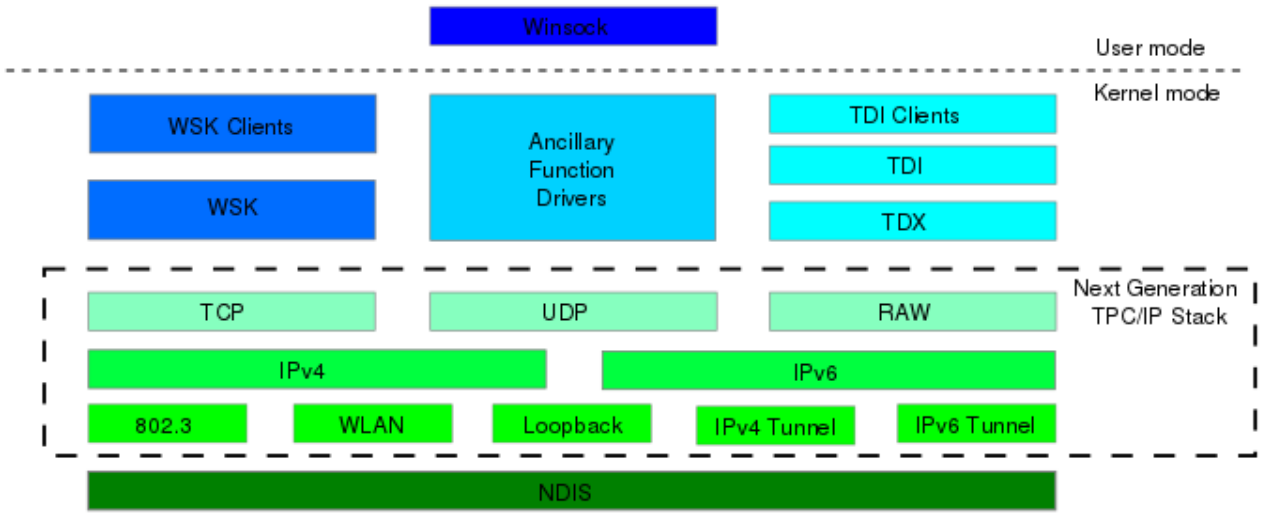
3.6.7.3.5 Drivers de protocolo.

La función de estos drivers es aceptar las peticiones provenientes de la API y traducirlas a un protocolo de bajo nivel para transmitirlos a través de la red. Para ello hacen uso de los drivers de protocolo de nivel de transporte, que son los que realizan la verdadera traducción. Al disponer de APIs separadas de los protocolos, es posible que la interfaz pueda utilizar diversos protocolos. Al haberse convertido TCP/IP un estándar de facto, es el protocolo que más se utiliza en Windows.

Debido a la necesidad de adoptar la versión 6 del protocolo IP, la pila de red de Windows, es una pila combinada, permitiendo la coexistencia de ambas versiones, utilizando túneles. Sus características más importantes son las siguientes:

- **Ajuste automático de la ventana de recepción**, pudiendo incrementar y reducir el tamaño de esta en función de la tecnología subyacente, y de la carga de las comunicaciones.
- **TCP compuesto (Compound TCP, o CTCP)**, tecnología que le permite cambiar agresivamente la cantidad de información enviada.
- **Notificación explícita de congestión (Explicit Congestion Notification, ECN)**. Si un paquete TCP se pierde, el sistema asume que existen congestiones en la red y reduce drásticamente la tasa de transmisión. Además, el router es capaz de marcar los paquetes como dirigidos durante la congestión para que el sistema disponga de otra razón para relajar la tasa de transmisión.
- **Mejoras de rendimiento en situaciones de muchas pérdidas**. Para ello se incluyen una serie de algoritmos que reducen la retransmisión de reconocimientos, o de segmentos TCP, en este tipo de escenarios, manteniendo la integridad del flujo procesado.

Resumimos los componentes de la pila TCP/IP en la figura siguiente, donde se observan los distintos estratos, y los componentes que conforman la pila, junto con el resto de componentes con los que ésta se relaciona:



3.6.7.3.6 Drivers NDIS.

El propósito de estos drivers es el de comunicar a la interfaz de red la información que debe transmitir, ofreciendo una interfaz estándar de comunicación, independiente de la lógica interna de la interfaz. Estos drivers, deben cumplir la especificación NDIS 6.1.

La librería NDIS limita con los drivers de nivel de transporte existentes, y, por otro lado, con los drivers de minipuerto NDIS, siendo una librería auxiliar para dar formato a los comandos enviados a los drivers NDIS. Ésta interacciona, además, con los drivers intermedios de NDIS, los drivers de filtro, encargándose de que el mensaje sea procesado por todos los drivers requeridos, ejerciendo de entorno de ejecución. De esta forma los drivers quedan envueltos por la librería, y no requieren recibir los paquetes de peticiones (IRPs) para procesarlos, sino que lo hará la librería por ellos. Los drivers de protocolo, entonces, utilizarán la función NdisAllocateNetBuffer, y posteriormente envían el paquete al driver de minipuerto a través de otra función. Todos los componentes de Windows hacen uso de buffers de red para gestionar el flujo de comunicaciones con NDIS.

Las características de NDIS principales son:

- La capacidad de anunciar si el medio está disponible.
- Los drivers pueden ser pausados, y reanudados, permitiendo su reconfiguración.
- Descarga TCP/IP, característica que permite que ciertas funcionalidades, como IPSec, IPv6, etc, sean implementadas por la tarjeta de red, y el sistema delegue en ella para realizar la transmisión.
- El escalado en recepción permite que múltiples procesadores se encarguen de procesar los paquetes recibidos, repartiendo la carga para un uso más eficiente de los procesadores disponibles.
- Tecnología Wake-on-LAN, que añade la funcionalidad de poder encender un ordenador apagado, o activarlo cuando se encuentra en estado de suspensión. Para ello se espera un evento que puede ser la conexión de una interfaz al medio de comunicación, la recepción de un patrón registrado en el protocolo, o un paquete especial recibido por el interfaz de red.
- División de los datos de cabecera en las interfaces de red que así lo permitan para dividir las

tramas de red en distintos buffers y poder combinarlos en regiones de memoria menores que si éstas se enviaran junto con los datos.

- NDIS orientado a conexión, permitiendo a la especificación utilizar dispositivos que requieran este tipo de comunicación, como los dispositivos del protocolo punto a punto(Point to Point Protocol).

Todas las funciones ofrecidas por la especificación NDIS, se traducen a las funciones correspondientes en la capa de abstracción de hardware (HAL).

3.6.7.3.7 Ensamblado.

Por último, existe una pieza en Windows que debe ser mencionada, puesto que después de un amplio análisis de todos los componentes de red, cabe preguntarse si deben estar todos activos al mismo tiempo, y de no estarlo, cómo se encuentran los unos a los otros. Es razonable pensar que los componentes se activan y desactivan en función de las distintas topologías de red, y funcionalidades utilizadas, y para reconocer la posibilidad de utilizar los distintos componentes se ofrece un servicio de ensamblado.

Un ejemplo de uso de dicho servicio, se produce cuando se instala un nuevo componente de red. Durante el proceso, el usuario debe proveer de un fichero INF en el que se ofrecen directrices para instalar y configurar el componente, además de las dependencias de ensamblado. Los desarrolladores especifican qué componentes requieren sus interfaces de modo que el gestor de control de servicios los cargará en memoria en el orden adecuado, y si otros componentes que utilicen la nueva interfaz están presentes. El servicio de ensamblado establece la conexión entre los componentes en las distintas capas, sirviéndose de la información ofrecida en el fichero INF.

Un ejemplo habitual se produce cuando se carga el servicio redirector de estaciones de trabajo, que automáticamente carga en memoria el protocolo TCP/IP.

3.7 Windows vs. Linux.

Después de un estudio detallado de como funcionan los dos sistemas operativos más populares del escritorio, cabe destacar que, aunque sus filosofías diverjan completamente, de cara a la programación, disponen de ciertas similitudes.

Por un lado, se dispone de una división clara y detallada en subsistemas que colaboran unos con otros para conseguir un objetivo dado. Esto nos permite indagar en ellos y estudiar las interfaces de los subsistemas que nos interesen, pudiendo obviar el resto. Por otro lado los subsistemas disponen de interfaces que permiten que se pueda trabajar con ellos obviando los detalles de la implementación, Los subsistemas en los que hemos centrado nuestra atención eran principalmente el subsistema de entrada/salida, el de dispositivos de bloques, y el de interfaces de red.

Los subsistemas de dispositivos de bloques de ambos sistemas operativos son muy distintos. Sin embargo, se apoyan en una idea común que facilita ampliamente las labores de este estudio. Ésta es la descripción de las peticiones, que, aunque detallábamos anteriormente que, en Linux se correspondía con una estructura, y en Windows con un paquete llamado IRP, un vistazo rápido a la documentación ofrecida en la infraestructura MSDN de Microsoft puede descubrirnos que dicho paquete se describe a través de una estructura. Esto nos permitirá buscar los campos comunes que

necesitemos para elaborar las peticiones que el servidor Swift deba procesar.

Parece razonable resaltar que, las interfaces con las que se relacionará este proyecto con el subsistema de bloques de los sistemas operativos, son distintas. En el caso de Linux, la interfaz con la que se relacionará, se corresponde con el organizador de peticiones, y en Windows con el gestor de entrada/salida.

Si observamos la parte relacionada con el subsistema de red, podemos resaltar que tanto en Windows como en Linux, se utilizan interfaces basadas en los sockets de Unix, lo que nos ofrecerá facilidades para aislar la relación dichas interfaces de forma simple, pudiendo hacer un diseño más independiente de la plataforma. Mientras más independiente sea el código del sistema operativo, más facilidad se ofrecerá para soportar nuevos sistemas, y se reducirán las complicaciones de mantenimiento.

3.8 Swiftness.

3.8.1 Bases de diseño.

Para definir la plataforma de drivers Swiftness, vamos a definir un protocolo de comunicación que utilice los servicios del protocolo TCP/IP para comunicarse con el servidor. Si quisiéramos establecer una analogía con los niveles del protocolo TCP/IP, este protocolo se encontraría en el nivel de aplicación, que, se corresponde con el conjunto de niveles del modelo OSI con los niveles de aplicación, presentación y sesión. Puesto que el objetivo principal es comunicarnos con el servidor Swift, de Openstack, es posible utilizar las bases de diseño de protocolos de red para este cometido.

En el diseño de un nivel de red, es posible definir cuatro primitivas: Request, Indication, Response y Confirm. La primitiva Request, se produce en el cliente, en un nivel cualquiera, y es la primitiva que inicia la comunicación. Esta primitiva utilizará los servicios del nivel inferior(en nuestro caso TCP/IP), para realizar la comunicación con su entidad par, en este caso el receptor del mensaje. Esta primitiva provoca que en la entidad par se utilice la primitiva Indication para recibir el mensaje transmitido por nuestro nivel de protocolo. La entidad par procesará el mensaje y hará uso de la primitiva Response para ejecutar una respuesta. La primitiva Response, por su parte, provocará que en el receptor, se ejecute la primitiva Confirm, para recibir la respuesta. De esta forma, y dado que nuestros drivers trabajan en el cliente, las primitivas Indication y Response son atendidas por el servidor Swift, y las primitivas Request y Confirm son las que se implementarán en Swiftness. Dado que la nomenclatura de diseño de los sistemas operativos que estudiamos, y por tanto, de los sockets de Unix, es distinta a la aquí nombrada, procederemos a utilizar los nombres Send y Recv para las primitivas Request y Confirm.

Dado que queremos disfrutar de un diseño lo más independiente posible de la plataforma donde se ejecute, desarrollaremos una interfaz que se comporte como un envoltorio, para poder utilizar los servicios de sockets de los distintos sistemas operativos, sin necesidad de utilizar la interfaz predefinida que nos ofrezca el sistema. Puesto que es posible que algunas opciones de la interfaz del sistema no nos sean útiles, podemos simplificarla conforme detallemos el diseño.

La función Send de nuestro protocolo, debe recibir una estructura de petición para traducirla a la correspondiente petición de la interfaz REST del servidor Swift. Dado que las peticiones a dispositivos de bloques se corresponden con operaciones de lectura y escritura, éstas se traducirán en operaciones de listado, creación, actualización, o recuperación de objetos, o en listados de contenedores. La operación debe decidirse en función de los valores de la estructura de petición y

traducirse correspondientemente.

La función Recv debe recoger la respuesta del servidor Swift y realizar el proceso inverso, es decir, traducirla para generar una estructura de respuesta con la información recibida del servidor Swift. Aquí la traducción puede convertirse en un listado de contenedores, un listado de objetos, una recuperación de objetos, o un mensaje de finalización de la operación. Para ello será necesario procesar la información recibida en el formato XML o JSON y crear la estructura correspondiente, para lo que podemos diseñar un pequeño reconocedor de lenguaje.

Otro punto a tener en cuenta durante el diseño, es que el servidor Swift requiere de identificación de usuario, luego es necesario que durante la operación de apertura del dispositivo, además de proveer el usuario y la contraseña, además de la dirección y puerto del servidor Swift objeto de las transferencias. Más allá de esto, la información debe ser almacenada, junto con los tokens identificativos que el servidor nos ofrezca, en estructuras internas del driver, puesto que, de provocarse una desconexión del servidor debido a inactividad, debe gestionarse una nueva conexión con el mismo usuario y contraseña, para posteriormente actualizar los tokens que ofrece el servidor.

Otro detalle a tener en cuenta es que los drivers son independientes del sistema de ficheros. Esto se traduce en que, las estructuras de peticiones y respuestas no disponen de los nombres de objetos y contenedores que se procesan en cada momento, sino de direcciones de almacenamiento en bruto, luego es necesario establecer una correspondencia entre las direcciones de las peticiones y los contenedores y objetos existentes, de cara a ambas operaciones descritas anteriormente.

Por último, es necesario definir las operaciones de registro del driver en el sistema, que en el caso de Linux se corresponde con las funciones init y exit, y en el caso de Windows con la rutina add-device.

4 Documentación de requisitos.

4.1 Participantes.

4.1.1 Organizaciones.

Organización	Departamento de Lenguajes y Sistemas Informáticos
Dirección	Avenida de Reina Mercedes S/N
Teléfono	(+34)954555964
Fax	(+34)954557139
Comentarios	Ninguno

Organización	Rackspace Cloud Computing
Dirección	5000 Wazelm Road, San Antonio, TX 78218
Teléfono	1-800-961-4454
Fax	PD
Comentarios	Ninguno

4.1.2 Participantes.

Participante	José Antonio Pérez Castellanos
Organización	Departamento de Lenguajes y Sistemas Informáticos
Rol	Tutor
Es desarrollador	No
Es cliente	Sí
Es usuario	No
Comentarios	Ninguno

Participante	José Ramón Muñoz Pekkarinen
Organización	Freelance
Rol	Desarrollador
Es desarrollador	Sí
Es cliente	No
Es usuario	No

Participantes.

Comentarios	Ninguno

4.2 Objetivos del sistema.

OBJ-0001	**Soporte de Openstack Swift para GNU/Linux**
Versión	1.0 (23/10/2012)
Autores	• José Ramón Muñoz Pekkarinen
Fuentes	• José Ramón Muñoz Pekkarinen
Descripción	El sistema deberá *ofrecer soporte de la plataforma Openstack Swift para el núcleo del sistema operativo GNU/Linux.*
Subobjetivos	• **[OBJ-0002] Identificación de usuario**: El sistema deberá *identificarse adecuadamente con la nube de almacenamiento cuando se le ofrezcan unas credenciales válidas.* • **[OBJ-0003] Recuperación de contenedores**: El sistema deberá *recuperar correctamente el listado de contenedores disponibles para el usuario identificado.* • **[OBJ-0004] Recuperación de objetos**: El sistema deberá *recuperar adecuadamente el listado de objetos dentro de un contenedor.* • **[OBJ-0005] Creación de contenedores**: El sistema deberá *crear contenedores dentro de la nube de almacenamiento, para el usuario identificado.* • **[OBJ-0006] Creación de objetos**: El sistema deberá *crear adecuadamente objetos dentro de un contenedor.* • **[OBJ-0007] Eliminación de contenedores**: El sistema deberá *eliminar adecuadamente contenedores.* • **[OBJ-0008] Eliminación de objetos**: El sistema deberá *eliminar objetos de un contenedor adecuadamente.*
Importancia	vital
Urgencia	inmediatamente
Estado	en construcción
Estabilidad	media
Comentarios	Ninguno

4.3 Requisitos del sistema.

4.3.1 Requisitos de información.

IRQ-0003	Nube de almacenamiento	
Versión	1.0 (24/10/2012)	
Autores	• José Ramón Muñoz Pekkarinen	
Fuentes	• José Ramón Muñoz Pekkarinen	
Dependencias	• [OBJ-0002] Identificación de usuario • [UC-0001] Identificación de usuario • [IRQ-0004] Sesión	
Descripción	El sistema deberá almacenar la información correspondiente a *datos de conexión con la nube de almacenamiento.* En concreto:	
Datos específicos	• Dirección IP • Puerto	
Tiempo de vida	**Medio**	**Máximo**
	60 minuto(s)	1 día(s)
Ocurrencias simultáneas	**Medio**	**Máximo**
	1	1
Importancia	importante	
Urgencia	inmediatamente	
Estado	en construcción	
Estabilidad	alta	
Comentarios	Ninguno	

IRQ-0001	Usuario
Versión	1.0 (23/10/2012)
Autores	• José Ramón Muñoz Pekkarinen
Fuentes	• José Ramón Muñoz Pekkarinen
Dependencias	• [OBJ-0002] Identificación de usuario • [UC-0001] Identificación de usuario

	• [IRQ-0004] Sesión	
Descripción	El sistema deberá almacenar la información correspondiente a *credenciales de identificación del usuario en la nube de almacenamiento.* En concreto:	
Datos específicos	• Nombre de usuario • Contraseña • Token de operación	
Tiempo de vida	**Medio**	**Máximo**
	60 minuto(s)	1 día(s)
Ocurrencias simultáneas	**Medio**	**Máximo**
	1	1
Importancia	vital	
Urgencia	inmediatamente	
Estado	en construcción	
Estabilidad	media	
Comentarios	Ninguno	

IRQ-0002	**Mapa de almacenamiento**	
Versión	1.0 (23/10/2012)	
Autores	• José Ramón Muñoz Pekkarinen	
Fuentes	• José Ramón Muñoz Pekkarinen	
Dependencias	• [OBJ-0003] Recuperación de contenedores • [OBJ-0004] Recuperación de objetos • [OBJ-0005] Creación de contenedores • [OBJ-0006] Creación de objetos • [OBJ-0008] Eliminación de objetos • [OBJ-0007] Eliminación de contenedores • [UC-0003] Listado de objetos • [UC-0004] Creación de un contenedor • [UC-0005] Creación de un objeto • [UC-0006] Borrado de contenedor • [UC-0007] Borrado de un objeto	
Descripción	El sistema deberá almacenar la información correspondiente a *mapa de almacenamiento de objetos en la nube.* En concreto:	
Datos específicos	• Listado de contenedores • Listado de objetos	
Tiempo de vida	**Medio**	**Máximo**

	60 minuto(s)	1 día(s)
Ocurrencias simultáneas	**Medio**	**Máximo**
	1	1
Importancia	vital	
Urgencia	inmediatamente	
Estado	en construcción	
Estabilidad	alta	
Comentarios	Ninguno	

IRQ-0004	**Sesión**	
Versión	1.0 (24/10/2012)	
Autores	• José Ramón Muñoz Pekkarinen	
Fuentes	• José Ramón Muñoz Pekkarinen	
Dependencias	• [OBJ-0003] Recuperación de contenedores • [OBJ-0004] Recuperación de objetos • [OBJ-0005] Creación de contenedores • [OBJ-0006] Creación de objetos • [OBJ-0007] Eliminación de contenedores • [OBJ-0008] Eliminación de objetos • [UC-0002] Listado de contenedores • [UC-0003] Listado de objetos • [UC-0004] Creación de un contenedor • [UC-0005] Creación de un objeto • [UC-0006] Borrado de contenedor • [UC-0007] Borrado de un objeto	
Descripción	El sistema deberá almacenar la información correspondiente a *datos de sesión para realizar sucesivas operaciones.*. En concreto:	
Datos específicos	• Token de operación • Dirección	
Tiempo de vida	**Medio**	**Máximo**
	30 minuto(s)	60 minuto(s)
Ocurrencias simultáneas	**Medio**	**Máximo**
	1	1
Importancia	vital	
Urgencia	inmediatamente	
Estado	en construcción	
Estabilidad	media	

Comentarios	Ninguno

4.3.2 Requisitos funcionales.

4.3.2.1 Diagrama de casos de uso.

Figura 1: Diagrama de casos de uso

4.3.2.2 Definición de actores.

ACT-0001	Usuario
Versión	1.0 (24/10/2012)
Autores	• José Ramón Muñoz Pekkarinen
Fuentes	• José Ramón Muñoz Pekkarinen
Descripción	Este actor representa *el usuario del sistema.*
Comentarios	Ninguno

4.3.2.3 Casos de uso.

4.3.2.3.1 Subsistema de login.

UC-0001	Identificación de usuario
Versión	1.0 (24/10/2012)
Autores	• José Ramón Muñoz Pekkarinen
Fuentes	• José Ramón Muñoz Pekkarinen
Dependencias	• [OBJ-0002] Identificación de usuario • [OBJ-0003] Recuperación de contenedores • [OBJ-0004] Recuperación de objetos • [OBJ-0005] Creación de contenedores • [OBJ-0006] Creación de objetos • [OBJ-0007] Eliminación de contenedores • [OBJ-0008] Eliminación de objetos • [UC-0002] Listado de contenedores • [UC-0003] Listado de objetos • [UC-0004] Creación de un contenedor • [UC-0005] Creación de un objeto

	• [UC-0006] Borrado de contenedor • [UC-0007] Borrado de un objeto • [IRQ-0004] Sesión • [UC-0008] Recuperación de objeto
Descripción	El sistema deberá comportarse tal como se describe en el siguiente caso de uso cuando *se abra el dispositivo de bloques o se reconecte a la nube.* o durante la realización de los siguientes casos de uso: [UC-0002] Listado de contenedores
Precondición	Los datos de conexión y de usuario deben ser válidos, y el dispositivo de bloques debe existir.
Secuencia normal	**Paso** / **Acción** 1 / El actor Usuario (ACT-0001) *solicita la apertura del dispositivo de bloques, ofreciendo la información de conexión y las credenciales de usuario.* 2 / El sistema *devuelve el token de operación y la dirección para las siguientes operaciones.*
Postcondición	Se ofrecerá al usuario un token de operación válido para las sucesivas operaciones.
Excepciones	**Paso** / **Acción** - / -
Rendimiento	**Paso** / **Tiempo máximo** - / -
Frecuencia esperada	5 veces por minuto(s)
Importancia	vital
Urgencia	inmediatamente
Estado	en construcción
Estabilidad	media
Comentarios	Ninguno

4.3.2.3.2 Subsistema de almacenamiento.

UC-0002	**Listado de contenedores**
Versión	1.0 (24/10/2012)
Autores	• José Ramón Muñoz Pekkarinen
Fuentes	• José Ramón Muñoz Pekkarinen
Dependencias	• [OBJ-0003] Recuperación de contenedores

	• [OBJ-0004] Recuperación de objetos • [OBJ-0005] Creación de contenedores • [OBJ-0006] Creación de objetos • [OBJ-0007] Eliminación de contenedores • [OBJ-0008] Eliminación de objetos • [UC-0003] Listado de objetos • [UC-0004] Creación de un contenedor • [UC-0005] Creación de un objeto • [UC-0006] Borrado de contenedor • [UC-0007] Borrado de un objeto • [UC-0008] Recuperación de objeto
Descripción	El sistema deberá comportarse tal como se describe en el siguiente caso de uso cuando *se solicite visualizar los contenedores del usuario.* o durante la realización de los siguientes casos de uso: [UC-0003] Listado de objetos, [UC-0004] Creación de un contenedor, [UC-0006] Borrado de contenedor
Precondición	El usuario debe estar conectado a la nube y el dispositivo abierto.
Secuencia normal	**Paso** / **Acción** 1 / Si *el usuario no está conectado a la nube, o el dispositivo está cerrado,* se realiza el caso de uso Identificación de usuario (UC-0001) 2 / El actor Usuario (ACT-0001) *solicita el listado de contenedores.* 3 / El sistema *devuelve un mapa con los contenedores disponibles y sus tamaños correspondientes*
Postcondición	El sistema devuelve un mapa de los distintos contenedores con sus respectivos tamaños.
Excepciones	**Paso** / **Acción** - / -
Rendimiento	**Paso** / **Tiempo máximo** - / -
Frecuencia esperada	**PD**
Importancia	vital
Urgencia	inmediatamente
Estado	en construcción
Estabilidad	media
Comentarios	Ninguno

UC-0003	**Listado de objetos**
Versión	1.0 (24/10/2012)
Autores	• José Ramón Muñoz Pekkarinen

Fuentes	• José Ramón Muñoz Pekkarinen
Dependencias	• [OBJ-0004] Recuperación de objetos • [OBJ-0006] Creación de objetos • [OBJ-0008] Eliminación de objetos • [UC-0005] Creación de un objeto • [UC-0007] Borrado de un objeto • [UC-0008] Recuperación de objeto
Descripción	El sistema deberá comportarse tal como se describe en el siguiente caso de uso cuando *se solicita la apertura de un contenedor.* o durante la realización de los siguientes casos de uso: [UC-0005] Creación de un objeto, [UC-0007] Borrado de un objeto, [UC-0008] Recuperación de objeto
Precondición	El usuario debe estar conectado a la nube, el dispositivo debe estar abierto, y el contenedor debe existir.
Secuencia normal	**Paso** / **Acción** 1 / Se realiza el caso de uso Listado de contenedores (UC-0002) 2 / El actor Usuario (ACT-0001) *solicita la apertura del contenedor.* 3 / El sistema *devuelve el listado de objetos del contenedor.*
Postcondición	Se devolverá un mapa con los objetos dentro de este contenedor.
Excepciones	**Paso** / **Acción** - / -
Rendimiento	**Paso** / **Tiempo máximo** - / -
Frecuencia esperada	**PD**
Importancia	vital
Urgencia	inmediatamente
Estado	en construcción
Estabilidad	media
Comentarios	Ninguno

UC-0004	**Creación de un contenedor**
Versión	1.0 (24/10/2012)
Autores	• José Ramón Muñoz Pekkarinen
Fuentes	• José Ramón Muñoz Pekkarinen
Dependencias	• [OBJ-0005] Creación de contenedores • [OBJ-0006] Creación de objetos • [OBJ-0007] Eliminación de contenedores

	• [OBJ-0008] Eliminación de objetos • [UC-0005] Creación de un objeto • [UC-0006] Borrado de contenedor • [CRQ-0002] Anidación de contenedores • [CRQ-0003] Creación de contenedores • [UC-0007] Borrado de un objeto
Descripción	El sistema deberá comportarse tal como se describe en el siguiente caso de uso cuando *se solicite la creación de un nuevo contenedor.*
Precondición	El usuario debe estar conectado a la nube, el dispositivo abierto y el contenedor no debe existir previamente.
Secuencia normal	**Paso** / **Acción** 1 / Se realiza el caso de uso Listado de contenedores (UC-0002) 2 / El actor Usuario (ACT-0001) *solicita la creación de un nuevo contenedor.* 3 / El sistema *devuelve una respuesta de operación terminada con éxito.*
Postcondición	Se creará un nuevo contenedor y se devolverá un mensaje de estado válido.
Excepciones	**Paso** / **Acción** - / -
Rendimiento	**Paso** / **Tiempo máximo** - / -
Frecuencia esperada	**PD**
Importancia	vital
Urgencia	inmediatamente
Estado	en construcción
Estabilidad	media
Comentarios	Ninguno

UC-0005	**Creación de un objeto**
Versión	1.0 (24/10/2012)
Autores	• José Ramón Muñoz Pekkarinen
Fuentes	• José Ramón Muñoz Pekkarinen
Dependencias	• [OBJ-0006] Creación de objetos • [OBJ-0008] Eliminación de objetos • [CRQ-0004] Creación de objetos • [UC-0007] Borrado de un objeto
Descripción	El sistema deberá comportarse tal como se describe en el siguiente caso de uso cuando *se solicita la creación de un nuevo objeto dentro de un contenedor.*
Precondición	El usuario debe estar conectado, el dispositivo de bloques debe estar abierto, el

	contenedor debe estar abierto, y el objeto no debe existir.
Secuencia normal	**Paso** **Acción** 1 Se realiza el caso de uso Listado de objetos (UC-0003) 2 El actor Usuario (ACT-0001) *solicita la creación de un objeto.* 3 El sistema *devuelve un mensaje de operación finalizada con éxito.*
Postcondición	El sistema devolverá un código de estado de la operación finalizado con éxito.
Excepciones	**Paso** **Acción** - -
Rendimiento	**Paso** **Tiempo máximo** - -
Frecuencia esperada	**PD**
Importancia	vital
Urgencia	inmediatamente
Estado	en construcción
Estabilidad	media
Comentarios	Ninguno

UC-0006	**Borrado de contenedor**
Versión	1.0 (24/10/2012)
Autores	• José Ramón Muñoz Pekkarinen
Fuentes	• José Ramón Muñoz Pekkarinen
Dependencias	• [CRQ-0001] Borrado de contenedores • [OBJ-0007] Eliminación de contenedores
Descripción	El sistema deberá comportarse tal como se describe en el siguiente caso de uso cuando *se solicite el borrado de un contenedor.*
Precondición	El usuario debe estar conectado, el dispositivo de bloques debe estar abierto, el contenedor debe existir, y no contener ningun objeto.
Secuencia normal	**Paso** **Acción** 1 Se realiza el caso de uso Listado de contenedores (UC-0002) 2 El actor Usuario (ACT-0001) *solicita el borrado de un contenedor.* 3 El sistema *devuelve un mensaje de estado de la operación finalizado con éxito.*
Postcondición	El sistema devuelve un código de operación realizada con éxito.
Excepciones	**Paso** **Acción** - -
Rendimiento	**Paso** **Tiempo máximo**

	-	-
Frecuencia esperada	**PD**	
Importancia	vital	
Urgencia	inmediatamente	
Estado	en construcción	
Estabilidad	media	
Comentarios	Ninguno	

UC-0007	**Borrado de un objeto**	
Versión	1.0 (24/10/2012)	
Autores	• José Ramón Muñoz Pekkarinen	
Fuentes	• José Ramón Muñoz Pekkarinen	
Dependencias	• [OBJ-0007] Eliminación de contenedores • [OBJ-0008] Eliminación de objetos • [UC-0006] Borrado de contenedor	
Descripción	El sistema deberá comportarse tal como se describe en el siguiente caso de uso cuando *se solicita el borrado de un objeto.*	
Precondición	El usuario debe estar conectado, el dispositivo de bloques debe estar abierto, el contenedor donde se aloja el objeto debe estar abierto, el objeto debe existir dentro del contenedor.	
Secuencia normal	**Paso**	**Acción**
	1	Se realiza el caso de uso Listado de objetos (UC-0003)
	2	El actor Usuario (ACT-0001) *solicita el borrado de un objeto.*
	3	El sistema *devuelve un código de operación realizada con éxito.*
Postcondición	El sistema devuelve un mensaje de finalización de la operación con éxito.	
Excepciones	**Paso**	**Acción**
	-	-
Rendimiento	**Paso**	**Tiempo máximo**
	-	-
Frecuencia esperada	**PD**	
Importancia	vital	
Urgencia	inmediatamente	
Estado	en construcción	
Estabilidad	media	
Comentarios	Ninguno	

UC-0008	Recuperación de objeto	
Versión	1.0 (28/10/2012)	
Autores	• José Ramón Muñoz Pekkarinen	
Fuentes	• José Ramón Muñoz Pekkarinen	
Dependencias	• [OBJ-0004] Recuperación de objetos	
Descripción	El sistema deberá comportarse tal como se describe en el siguiente caso de uso cuando *se solicite recuperar un objeto de la nube de almacenamiento.*	
Precondición	El usuario debe estar conectado a la nube, y el dispositivo de bloques abierto.	
Secuencia normal	**Paso**	**Acción**
	1	Se realiza el caso de uso Listado de objetos (UC-0003)
	2	El actor Usuario (ACT-0001) *solicita la recuperación de un objeto.*
	3	El sistema *devuelve un mensaje con el objeto.*
Postcondición	**PD**	
Excepciones	**Paso**	**Acción**
	-	-
Rendimiento	**Paso**	**Tiempo máximo**
	-	-
Frecuencia esperada	**PD**	
Importancia	vital	
Urgencia	inmediatamente	
Estado	en construcción	
Estabilidad	media	
Comentarios	Ninguno	

4.3.3 Requisitos no funcionales

NFR-0001	Soporte de sistemas GNU/Linux
Versión	1.0 (23/10/2012)
Autores	• José Ramón Muñoz Pekkarinen
Fuentes	• José Ramón Muñoz Pekkarinen

Dependencias	• [OBJ-0001] Soporte de Openstack Swift para GNU/Linux
Descripción	El sistema deberá *soportar el almacenamiento en la nube sobre sistemas operativos GNU/Linux*
Importancia	vital
Urgencia	inmediatamente
Estado	en construcción
Estabilidad	alta
Comentarios	Ninguno

NFR-0002	**Visibilidad de la nube como dispositivo de bloques**
Versión	1.0 (23/10/2012)
Autores	• José Ramón Muñoz Pekkarinen
Fuentes	• José Ramón Muñoz Pekkarinen
Dependencias	• [OBJ-0001] Soporte de Openstack Swift para GNU/Linux
Descripción	El sistema deberá *mostrar la nube de almacenamiento como si de un dispositivo de bloques local se tratara.*
Importancia	vital
Urgencia	inmediatamente
Estado	en construcción
Estabilidad	media
Comentarios	Ninguno

NFR-0003	**Reconexión automática**
Versión	1.0 (23/10/2012)
Autores	• José Ramón Muñoz Pekkarinen
Fuentes	• José Ramón Muñoz Pekkarinen
Dependencias	• [OBJ-0001] Soporte de Openstack Swift para GNU/Linux • [OBJ-0003] Recuperación de contenedores • [OBJ-0004] Recuperación de objetos • [OBJ-0005] Creación de contenedores • [OBJ-0006] Creación de objetos • [OBJ-0008] Eliminación de objetos • [OBJ-0007] Eliminación de contenedores
Descripción	El sistema deberá *reconectar automáticamente con la nube, en caso de cierre de conexión por inactividad.*
Importancia	importante

Urgencia	puede esperar
Estado	en construcción
Estabilidad	media
Comentarios	Ninguno

4.3.4 Reglas de negocio

CRQ-0003	**Creación de contenedores**
Versión	1.0 (24/10/2012)
Autores	• José Ramón Muñoz Pekkarinen
Fuentes	• José Ramón Muñoz Pekkarinen
Dependencias	• [OBJ-0005] Creación de contenedores
Descripción	La información almacenada por el sistema deberá satisfacer la siguiente restricción: *No se puede crear un contenedor que ya exista.*
Importancia	vital
Urgencia	inmediatamente
Estado	validado
Estabilidad	alta
Comentarios	Ninguno

CRQ-0004	**Creación de objetos**
Versión	1.0 (24/10/2012)
Autores	• José Ramón Muñoz Pekkarinen
Fuentes	• José Ramón Muñoz Pekkarinen
Dependencias	• [OBJ-0006] Creación de objetos
Descripción	La información almacenada por el sistema deberá satisfacer la siguiente restricción: *No se puede crear un objeto dentro de un contenedor que ya exista en este.*
Importancia	vital
Urgencia	inmediatamente
Estado	validado
Estabilidad	alta
Comentarios	Ninguno

CRQ-0001	Borrado de contenedores
Versión	1.0 (23/10/2012)
Autores	• José Ramón Muñoz Pekkarinen
Fuentes	• José Ramón Muñoz Pekkarinen
Dependencias	• [OBJ-0007] Eliminación de contenedores • [IRQ-0003] Nube de almacenamiento • [UC-0007] Borrado de un objeto
Descripción	La información almacenada por el sistema deberá satisfacer la siguiente restricción: *No se puede borrar un contenedor que contenga objetos.*
Importancia	vital
Urgencia	inmediatamente
Estado	validado
Estabilidad	media
Comentarios	Ninguno

CRQ-0002	Anidación de contenedores
Versión	1.0 (23/10/2012)
Autores	• José Ramón Muñoz Pekkarinen
Fuentes	• José Ramón Muñoz Pekkarinen
Dependencias	• [OBJ-0005] Creación de contenedores
Descripción	La información almacenada por el sistema deberá satisfacer la siguiente restricción: *No se puede crear un contenedor dentro de otro contenedor.*
Importancia	vital
Urgencia	inmediatamente
Estado	validado
Estabilidad	media
Comentarios	Ninguno

4.4 Matriz de rastreabilidad

TRM-0001	OBJ-0001	OBJ-0002	OBJ-0003	OBJ-0004	OBJ-0005	OBJ-0006	OBJ-0007	OBJ-0008

Matriz de rastreabilidad

IRQ-0001	-	↑	-	-	-	-	-	-
IRQ-0002	-	-	↑	↑	↑	↑	↑	↑
IRQ-0003	-	↑	-	-	-	-	-	-
IRQ-0004	-	-	↑	↑	↑	↑	↑	↑
UC-0001	-	↑	↑	↑	↑	↑	↑	↑
UC-0002	-	-	↑	↑	↑	↑	↑	↑
UC-0003	-	-	-	↑	-	↑	-	↑
UC-0004	-	-	-	-	↑	↑	↑	↑
UC-0005	-	-	-	-	-	↑	-	↑
UC-0006	-	-	-	-	-	-	↑	-
UC-0007	-	-	-	-	-	-	↑	↑
UC-0008	-	-	-	↑	-	-	-	-

Matriz de rastreabilidad: Requisitos de información

TRM-0002	OBJ-0001	OBJ-0002	OBJ-0003	OBJ-0004	OBJ-0005	OBJ-0006	OBJ-0007	OBJ-0008	IRQ-0001	IRQ-0002	IRQ-0003	IRQ-0004
UC-0001	-	↑	↑	↑	↑	↑	↑	↑	-	-	-	↑
UC-0002	-	-	↑	↑	↑	↑	↑	↑	-	-	-	-
UC-0003	-	-	-	↑	-	↑	-	↑	-	-	-	-
UC-0004	-	-	-	-	↑	↑	↑	↑	-	-	-	-
UC-0005	-	-	-	-	-	↑	-	↑	-	-	-	-
UC-0006	-	-	-	-	-	-	↑	-	-	-	-	-

UC-0007	-	-	-	-	-	-	↑	↑	-	-	-	-
UC-0008	-	-	-	↑	-	-	-	-	-	-	-	-

Matriz de rastreabilidad: Requisitos funcionales

5 Documentación de análisis.

5.1 Modelo estático.

5.1.1 Subsistema de Login.

Figura 1: Diagrama de clases de Subsistema de Login

5.1.2 Subsistema de Almacenamiento.

Figura 1: Diagrama de clases de Subsistema de almacenamiento

5.2 Modelo dinámico.

5.2.1 Subsistema de Login.

Figura 1: Diagrama de secuencia de login

5.2.2 Subsistema de almacenamiento.

Figura 1: Diagrama de secuencia de listado de contenedores

Figura 2: Diagrama de secuencia de listado de objetos

Figura 3: Diagrama de secuencia de creación de un contenedor

Figura 4: Diagrama de secuencia de borrado de un contenedor

Figura 5: Diagrama de secuencia de creación de un objeto

Figura 6: Diagrama de secuencia de recuperación de objetos

Figura 7: Diagrama de secuencia de borrado de un objeto

5.3 Modelo funcional.

5.3.1 Subsistema de Login.

SOP-0001	login
Tipo del resultado	Sesión
Versión	1.0 (27/10/2012)
Autores	• José Ramón Muñoz Pekkarinen
Dependencias	• [UC-0001] Identificación de usuario

Descripción	Esta función comprueba si el usuario ofrecido es válido y devuelve los datos de sesión necesarios en caso afirmativo.
Parámetros	`Dirección : String --` *Dirección del servidor.* `Puerto : Integer --` *Puerto de escucha del servidor.* `Nombre : String --` *Nombre de usuario.* `Contraseña : String --` *Contraseña del usuario.*
Expresiones de precondición	**Servidor válido**: Los datos de conexión deben ser válidos. **Usuario válido**: El usuario debe existir en el servidor.
Expresiones de precondición (OCL)	**`Servidor válido`**`: Servidor.allInstances()->one( srv | srv.dirección = Dirección AND srv.puerto = Puerto)` **`Usuario válido`**`: (Servidor.allInstances()->select( srv | srv.dirección = Dirección AND srv.puerto = Puerto)).Usuario->one( usr | usr.nombre = Nombre and usr.password = Contraseña)`
Expresiones de postcondición	**Objeto de sesión válido.**: El objeto de sesión debe contener un token y una url
Expresiones de postcondición (OCL)	**`Objeto de sesión válido.`**`: let token = Sesión.allInstances() -> count() + 1, session: Sesión serv: Servidor = any(srv | srv.dirección = Dirección and srv.puerto = Puerto) user: Usuario = any(usr | usr.nombre = Nombre and user.password = Contraseña) in session.oclIsNew() = True and session.url = url and session.token = token and session.servidor = serv and session.usuario = user and result = session`
Comentarios	Ninguno

5.3.2 Subsistema de almacenamiento.

SOP-0002	**listContainers**
Tipo del resultado	`Lista de Contenedor`
Versión	1.0 (27/10/2012)
Autores	• José Ramón Muñoz Pekkarinen
Dependencias	• [UC-0002] Listado de contenedores
Descripción	Esta función genera una lista de contenedores del usuario.

Parámetros	`Token : String --` *Token de operación.* `Url : String --` *Url donde se procesa la operación.*
Expresiones de precondición	**Sesión existente**: La sesión debe existir.
Expresiones de precondición (OCL)	`Sesión existente: Sesión.allInstances()->one(session | session.token= Token and session.url = URL)`
Expresiones de postcondición	**Listado de contenedores**: Genera la lista de contenedores
Expresiones de postcondición (OCL)	`Listado de contenedores: result = Contenedor.allInstances()`
Comentarios	Ninguno

SOP-0003	**listObjects**
Tipo del resultado	`Lista de Objetos`
Versión	1.0 (27/10/2012)
Autores	• José Ramón Muñoz Pekkarinen
Dependencias	• [UC-0003] Listado de objetos
Descripción	La operación devuelve el listado de objetos de un contenedor dado.
Parámetros	`Token : String --` *Token de sesión.* `Url : String --` *Url donde se procesa la operación.* `Container : String --` *Nombre del contenedor del que se requiere el listado.*
Expresiones de precondición	**Sesión existente**: La sesión debe existir. **Contenedor existente**: El contenedor indicado debe existir
Expresiones de precondición (OCL)	`Sesión existente: Sesión.allInstances()->one(session | session.token= Token and session.url = URL)` `Contenedor existente: Contenedor.allInstances()->includes(Container)`
Expresiones de postcondición	**Listado de objetos**: Lista de objetos del contenedor indicado.
Expresiones de	`Listado de objetos: result = Container.Objeto.allInstances()`

postcondición (OCL)	
Comentarios	Ninguno
SOP-0004	**createContainer**
Tipo del resultado	`Código de estado.`
Versión	1.0 (27/10/2012)
Autores	• José Ramón Muñoz Pekkarinen
Dependencias	• [UC-0004] Creación de un contenedor
Descripción	La operación debe crear un contenedor nuevo en el servidor.
Parámetros	`Token : String --` *Token de sesión.* `Url : String --` *Url donde procesar la operación.* `Name : String --` *Nombre del nuevo contenedor.*
Expresiones de precondición	**Sesión existente**: La sesión debe existir. **Contenedor no existente**: No puede existir un contenedor con el mismo nombre.
Expresiones de precondición (OCL)	**`Sesión existente`**`: Sesión.allInstances()->one(session | session.token= Token and session.url = URL)` **`Contenedor no existente`**`: not Contenedor.allInstances()->one(cont | cont.nombre = name)`
Expresiones de postcondición	**Creación de contenedor**: Creación de un nuevo contenedor.
Expresiones de postcondición (OCL)	**`Creación de contenedor`**`: let session: Sesión = any(session | session.token = Token and session.url = Url) container: Contenedor in container.oclIsNew() = True and container.Nombre = Name and container.tamaño = 0 and container.Sesión = session and session.Contenedor->includes(container) and result = container`
Comentarios	Ninguno
SOP-0005	**deleteContainer**
Tipo del resultado	`Código de finalización de la operación.`
Versión	1.0 (27/10/2012)
Autores	• José Ramón Muñoz Pekkarinen
Dependencias	• [UC-0006] Borrado de contenedor
Descripción	Elimina un contenedor vacío.

Parámetros	`Token : String --` *Token de sesión.* `Url : String --` *Url de proceso de operaciones.* `Name : String --` *Nombre del contenedor a eliminar.*
Expresiones de precondición	**Sesión existente**: La sesión debe existir **Contenedor existente**: El contenedor debe existir.
Expresiones de precondición (OCL)	**`Sesión existente`**`: Sesión.allInstances()->one(session | session.token= Token and session.url = URL)` **`Contenedor existente`**`: Contenedor.allInstances()->one(container | container.Nombre = name)`
Expresiones de postcondición	**Contenedor no existente.**: El contenedor no debe seguir existiendo.
Expresiones de postcondición (OCL)	**`Contenedor no existente.`**`: not Contenedor.allInstances()->one(cont | cont.Nombre = Name)`
Comentarios	Ninguno

SOP-0006	**createObject**
Tipo del resultado	`Código de finalización de la operación.`
Versión	1.0 (27/10/2012)
Autores	• José Ramón Muñoz Pekkarinen
Dependencias	• [UC-0005] Creación de un objeto
Descripción	Creación de un objeto en un contenedor dado.
Parámetros	`Token : String --` *Token de sesión.* `Url : String --` *Url de proceso de operaciones.* `ContainerName : String --` *Contenedor donde debe crearse el objeto.* `Name : String --` *Nombre del objeto.* `Size : Integer --` *Tamaño del objeto.*
Expresiones de precondición	**Sesión existente**: La sesión debe existir **Contenedor existente**: El contenedor indicado debe existir. **Objeto no existente**: El objeto no debe existir en el contenedor.
Expresiones de precondición (OCL)	**`Sesión existente`**`: Sesión.allInstances()->one(session | session.token= Token and session.url = URL)` **`Contenedor existente`**`: Contenedor.allInstances()->one(container | container.Nombre = ContainerName)` **`Objeto no existente`**`: not Contenedor.allInstances()->any(container | container.Nombre = ContainerName).Objeto->one(obj | obj.Nombre = name)`

Expresiones de postcondición	**Creación de objeto**: Creación de un nuevo objeto.
Expresiones de postcondición (OCL)	`Creación de objeto: let session: Sesión = any(session | session.token = Token and session.url = Url) container: Contenedor = any(cont | cont.Nombre = ContainerName) object: Objeto in object.oclIsNew() = True and object.Nombre = Name and object.tamaño = size and object.Sesión = session and session.Objeto->includes(object) and object.Contenedor = container and container.Objeto->includes(object) and result = Object`
Comentarios	Ninguno

SOP-0007	**deleteObject**
Tipo del resultado	`Código de finalización de operación.`
Versión	1.0 (27/10/2012)
Autores	• José Ramón Muñoz Pekkarinen
Dependencias	• [UC-0007] Borrado de un objeto
Descripción	Eliminación de un objeto dentro de un contenedor.
Parámetros	`Token : String --` *Token de sesión.* `Url : String --` *Url de proceso de operaciones.* `Container : String --` *Contenedor donde se aloja el objeto.* `Name : String --` *Nombre del objeto a eliminar.*
Expresiones de precondición	**Sesión existente**: La sesión debe existir **Contenedor existente**: El contenedor debe existir. **Objeto existente**: El objeto debe existir dentro del contenedor.

Expresiones de precondición (OCL)	`Sesión existente: Sesión.allInstances()->one(session | session.token= Token and session.url = URL)`
	`Contenedor existente: Contenedor.allInstances()-> one(cont | cont.nombre = Container)`
	`Objeto existente: Contenedor.allInstances()->any(cont | cont.Nombre = Container).Objeto.one(obj | obj.Nombre = Name)`

Expresiones de postcondición	**Objeto no existente**: El objeto debe dejar de existir en el contenedor.
Expresiones de postcondición (OCL)	`Objeto no existente: not Contenedor.allInstances()->any(cont | cont.Nombre = Container).Objeto->one(obj | obj.Nombre = Name)`
Comentarios	Ninguno

6 Documentación de diseño.

6.1 Arquitectura del sistema.

Dentro de las posibles arquitecturas existentes, y por similitud con los objetivos de este proyecto, se convierte en objeto de estudio la arquitectura orientada a servicios. A lo largo de este apartado haremos un resumen de las características de dicha arquitectura, y comentaremos las razones que hacen de ésta, la arquitectura adecuada para afrontar nuestra situación.

Los principios y metodologías de dicha arquitectura, fueron pensados para el diseño y desarrollo de servicios interoperables. Cada servicio definido, desarrolla una funcionalidad concreta, lo más independiente posible del resto, evitando así problemas de acoplamiento. Estas funcionalidades, se definen como componentes software que pueden ser reutilizados con diferentes propósitos. Los principios definidos por la arquitectura cobran mayor importancia en las fases de desarrollo e integración. Este tipo de arquitecturas es altamente adecuado para resolver problemas donde las interfaces tengan gran importancia, deban ser claras, y bien definidas. Su principal objetivo es definir la integración de sistemas dispares para su uso conjunto.

Aunque en sus principios básicos, dicha arquitectura debe ser independiente del sistema operativo, para cumplir nuestro objetivo, dicha condición es inviable, debe haber cierto grado de dependencia puesto que, aunque la plataforma será lo más independiente posible del sistema operativo, hacerla completamente independiente elevaría los costes y complejidad de la solución a valores no rentables. Otra característica principal de dicha arquitectura es la capacidad de utilizar servicios a través de la red, lo cual supone un objetivo principal en esta ocasión, haciendo que la interfaz REST del servidor, sea interpretada por el sistema operativo. Esta última característica hace que dicha arquitectura se asocie habitualmente con los sistemas distribuidos.

Sus servicios, no deben estar asociados, deben ser desacoplados, y no requerirse los unos a los otros. Éstos implementarán una funcionalidad, simple, y de requerir los servicios de otra funcionalidad, definen un protocolo de comunicación que pase un mensaje, en lugar de realizar una llamada internamente, al servicio requerido. Para relacionar los distintos objetos suele utilizarse la técnica de orquestación, dotando al sistema de la inteligencia necesaria para preparar, coordinar y gestionar los recursos necesarios para realizar su función. Con este fin, deben definirse claramente los distintos servicios, y cómo se desenvuelven, hasta la finalización de su operación. Además, es habitual que se utilicen metadatos que sean utilizados a lo largo del uso de dicha técnica, y que ayuden como información complementaria para decidir, cuál debe ser el paso siguiente.

Con estas características se pretende que el usuario pueda disponer de grandes funcionalidades, en un entorno ad-hoc, basado en amplias funcionalidades. A más amplias sean, más pequeñas serán las interfaces requeridas. Sin embargo, debe buscarse un acuerdo entre este punto, y el grado de granularidad ofrecido, para que el sistema no pierda la capacidad de reutilización. No es, tampoco razonable utilizar un amplio número de interfaces para alcanzar la mayor granularidad posible, puesto que por cada interfaz se añade una sobrecarga de ejecución, pudiendo comprometer la capacidad de respuesta de éste. Una vez definida la arquitectura, construir aplicaciones sobre ella se reduce a orquestar los servicios adecuadamente, reuniendo los componentes software necesarios, ahorrando tiempo de desarrollo, y reutilizando los servicios ofrecidos.

Puesto que un objetivo principal es que las funcionalidades no interaccionen con otras, debe ser el usuario quien define en qué orden se utilizan los servicios para que se resuelva su propósito. Sus requisitos se resumen en los siguientes:

- Interoperabilidad entre sistemas y lenguajes para ofrecer de integración entre diversos sistemas a través de protocolos de comunicación.
- Crear un conjunto de mecanismos básicos en los que se apoyen las distintas aplicaciones para ofrecer nuevas funcionalidades.

Sus principios se resumen en los siguientes:

- Reutilización, granularidad, modularidad, composición e interoperabilidad.
- Capacidad de cumplir con estándares generalizados.
- Identificación, categorización, aprovisionamiento, distribución, monitorización y seguimiento de servicios.

Estos principios y requisitos, se ven reflejados en una gran mayoría de sistemas orientados a la red, implicando que muchos de los servicios web e interfaces se apoyen en este tipo de arquitecturas, no siendo REST, una excepción, pero además, dicha arquitectura no dista demasiado de las ideas utilizadas para diseñar un sistema operativo, en cuanto a que se diseña un conjunto de servicios mínimos, que puedan ser utilizados de diversas maneras por el usuario, tratando de que los mecanismos requieran la menor cooperación posible entre ellos. Es por esto, que el desarrollo de este proyecto que se encuentra en un término medio entre ambos tipos de sistemas encuentra en esta arquitectura una gran oportunidad para resolver el problema siguiendo su metodología.

6.2 Patrones de diseño.

En los siguientes apartados realizaremos un breve resumen de los patrones requeridos para estructurar el driver, y cómo se pretende aplicarlos para ir definiendo el diseño del driver que implementaremos en el proyecto. Para ello haremos un diseño Top-down que defina cómo una petición se procesará por las distintas partes del driver hasta llegar a la interfaz de red, y cómo debe ir transformándose la petición para que el servidor Swift la procese correctamente.

Conviene recordar que, el objetivo de nuestra plataforma es hacer un driver de dispositivo de bloques, y no crear un sistema de ficheros que sea capaz de resolver la comunicación, luego trabajamos en los niveles más bajos de hardware. En el estudio realizado previamente acerca del sistema operativo GNU/Linux, comentábamos la existencia del módulo NBD que ya ejercía ciertas tareas relacionadas con nuestro objetivo. En concreto, éste driver define una interfaz con el sistema de ficheros virtual que recogía las peticiones que este realizaba, en bruto, para enviarlas a través de la red y ser procesadas en el servidor, en espacio de usuario, que exponga su espacio de almacenamiento.

La diferencia en nuestro contexto, es que no esperamos que la petición realizada por el sistema de ficheros, sea resuelta por el servidor NBD, sino por el servidor Openstack Swift, luego, la petición generada por el sistema de ficheros debe ser traducida a la interfaz REST del servidor Swift, para ser enviada por la interfaz de red. Esto, además añadirá cierta complejidad, puesto que hay que resolver las ambigüedades producidas, cuando se quiere crear un contenedor, o un objeto,

cuando se quiere listar un contenedor, o recuperar un objeto, o eliminar un contenedor u objeto.

En resumidas cuentas, traduciremos las peticiones distintas que el sistema de ficheros virtual nos ofrece, a las peticiones correspondientes a las operaciones posibles que el servidor nos ofrezca, y para ello debemos servirnos de los siguientes patrones.

6.2.1 Fachada

En este conocido patrón de diseño se pretende ocultar los detalles irrelevantes al usuario, ofreciéndole una interfaz simplificada que resuelva sus peticiones. Sus principales características son las siguientes:

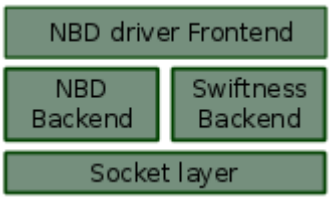
- Hacer que un componente software sea más sencillo de utilizar, entender y probar, ofreciendo métodos convenientes para tareas comunes.
- Hacer que la librería sea más legible.
- Reducir el número de dependencias con el código exterior, puesto que sólo utilizará lo que la fachada ofrezca.
- Envolver una librería con diseño pobre.

Cabría la posibilidad de pensar que un adaptador es suficiente para el fin perseguido, puesto que la plataforma debe traducir las peticiones generadas por el sistema de ficheros virtual a las peticiones REST de Openstack Swift. La situaciones en las que el adaptador es utilizado implican interfaces previamente definidas, y, en nuestro caso, no es necesario utilizar una interfaz ya existente. Es más, éstas cambiarán en función del sistema operativo, o en situaciones en las que se utilice el polimorfismo, que, por tratarse de programación de sistemas operativos, esta característica se encuentra fuera de los lenguajes de programación utilizados habitualmente.

En concreto conviene ofrecer la posibilidad, al dispositivo NBD de que seleccione que backend quiere utilizar para resolver sus peticiones. Esto puede resolverse fácilmente añadiéndole la posibilidad de elegir qué función de transmisión, y de recepción debe utilizar para las comunicaciones.

De esta forma, en nuestro driver sólo es necesario definir las funciones de envío y de recepción. Haciendo analogía con los diseños de redes de computadores, deberemos definir las primitivas Request y Confirm, que son las primitivas que se utilizan en el transmisor. Al encontrarnos en el sistema operativo, dichas primitivas suelen ser más conocidas como send y receive. Estas dos funciones serán las que conformen la interfaz de fachada, y sus parámetros requeridos serán el dispositivo en cuestión, y la estructura de la petición determinada.

Definida la fachada, el sistema podemos representarlo como una caja negra, que iremos descomponiendo con el uso del resto de patrones, conformando un panorama parecido al siguiente:



6.2.2 Adaptador.

Puesto que debe producirse una traducción, de la petición ofrecida por el sistema de ficheros virtual a la petición requerida por el servidor Swift, este patrón también se convierte en objeto de estudio, y debe aplicarse, puesto que es lo que definirá la pieza de código del backend de Swiftness que orquestará el uso de las funciones de nuestra arquitectura, decidiendo en cada momento qué operación realizar, y en que orden.

El objetivo de este patrón es adaptar interfaces distintas que impiden que determinados objetos colaboren unos con otros traduciendo sus llamadas, de forma que esta restricción desaparezca. En nuestro caso, los objetos que deben colaborar, son la petición, o respuesta del sistema de ficheros virtual, con respecto a los sockets de comunicación internos al núcleo de Linux. Sus principales características son las mismas que la de una fachada, con la restricción de utilizar interfaces fijas, y adaptarse a un, inexistente en nuestro caso, polimorfismo.

Para que la traducción pueda realizarse en ambos sentidos, la interfaz que limitará con la fachada previamente definida dispondrá de dos funciones, que traducirán cada uno en un sentido de la comunicación. Éstas serán las funciones RequestTranslation, que recibirá la estructura de la petición en proceso, la traducirá a la operación que corresponda con la petición, y ReplyTranslation, que recibirá una respuesta del servidor Swift en formato XML y la procesará generando la respuesta adecuada al sistema de ficheros virtual, o nuevas operaciones a procesar por el servidor Swift. Estas nuevas operaciones se relacionarían con el tratamiento de errores.

Puede contemplarse, en un futuro ciclo de desarrollo de este proyecto, añadir la funcionalidad a este adaptador, de poder trabajar con respuestas en formato JSON.

6.2.3 Envoltorio.

Por último, y con objeto de que nuestra implementación reduzca las dependencias con el sistema operativo que lo gobierna, debe implementarse una interfaz que permita a la plataforma desentenderse del sistema operativo, y especializando las interfaces ofrecidas por éstos a las funciones requeridas por nuestra plataforma. Puesto que las comunicaciones en ambos sistemas operativos, se basan en sockets UNIX, será suficiente con envolver las funciones específicas de creación, destrucción, envío y recepción, en una interfaz genérica utilizada por las operaciones. De esta forma estaremos simplificando su uso, adaptándolo a nuestra estricta necesidad, a la par que reduciendo el acoplamiento con el sistema operativo, características claves de dicho patrón.

6.3 Diagrama de clases de diseño.

En resumidas cuentas, los tres patrones guardan mucho en común, y la elección de cada uno de ellos trata de ajustarse lo más posible a las necesidades de cada tarea a resolver. Conviene recordar que, aunque los patrones definan tres interfaces, es necesaria una interfaz más que reunirá los servicios de la arquitectura. De las cuatro interfaces que se definirán, sólo quedará expuesta la fachada, para que interaccione con el sistema de ficheros virtual, incluyéndose en el fichero “include/linux/swt.h”, y el resto de interfaces se incluirán directamente en el fichero “drivers/block/swt.c”. El panorama de dichas interfaces, en colaboración con los objetos será el

siguiente.

6.4 Diseño de datos.

Para el diseño de los datos gestionados por nuestra plataforma utilizaremos el conocido modelo de dominios, por su proximidad con los modelos de datos orientados a objetos. Aunque las posteriores implementaciones se realizarán con lenguajes no orientados a objetos, si existe cierta correspondencia de los elementos a almacenar con objetos pertenecientes a sus respectivas clases, salvando la distancia de que estos no tendrán la posibilidad de hacer operaciones sobre sí mismos o

sobre otras clases con las que se relacionen.

El modelo siguiente, resalta ciertos objetos, con intención de indicar que son objetos propios del núcleo de GNU/Linux. Estos objetos no serán detallados ampliamente, y pueden ser consultados en la documentación ofrecida dentro de la carpeta “Documentation” de las fuentes del núcleo, en la documentación ofrecida por el proyecto “The Linux Documentation Project”, TLDP, o en cualquiera de las referencias ofrecidas al final de este documento. La intención de incluirlos es su relación con los objetos que nuestra plataforma gestiona.

Las entidades existentes en nuestro sistema se recogen en las siguientes tablas:

swt_usr			
Atributo	Dominio	Es nulo	Descripción
Name	Cadena	No	Nombre de usuario
Key	Cadena	No	Password

swt_srv			
Atributo	Dominio	Es nulo	Descripción
Address	Cadena	No	Dirección del servidor Swift
Port	Entero	No	Puerto del servidor

swt_sess			
Atributo	Dominio	Es nulo	Descripción
Token	Cadena	No	Token de operación devuelto por el servidor
Url	Cadena	No	Url de operaciones siguientes

swt_op			
Atributo	Dominio	Es nulo	Descripción
Op	Entero	No	Operación a realizar
Container	Cadena	Sí	Contenedor utilizado en la operación
Object	Cadena	Sí	Objeto utilizado en la operación

reply			
Atributo	Dominio	Es nulo	Descripción
Op	Entero	No	Operación de la que se recibe respuesta
Size	Entero	Sí	Tamaño del objeto modificado

6.5 Diseño lógico.

En cuanto al diseño lógico, poco puede destacarse, puesto que las entidades detalladas anteriormente, que serán de interés de cara a la implementación del proyecto que nos concierne, tienen su correspondencia directa con una o varias estructuras en lenguaje de programación C. Estas estructuras se almacenan en espacio del núcleo, conllevando el consecuente peligro de perjudicar el rendimiento de éste, si no se limpia cuidadosamente el espacio utilizado por cualquiera de ellas, cuando dejan de ser útiles.

Las estructuras estarán formadas por los campos detallados en el diseño de datos detallado en el apartado 6.4, añadiéndole los campos que corresponden a la relación de la estructura, con otra del mismo tipo, o de uno distinto. Las estructuras entonces son las siguientes:

```
struct swt_auth {
        char * user;
        char * key;
};

struct swt_serv {
        char * host;
        int port;
};

struct swt_sess {
        char * url;
        char * token;
};

struct swt_op {
        int op;
        const char * container;
        const char * object;
        struct request * req;
};

struct object
{
        char * name;
        int size;
        struct object * next;
};
```

```
struct container
{
	char * name;
	int size;
	struct container * next;
	struct object * first;
};

struct swt_reply {
	int op;
	int size;
	struct list_head unprocessed;
};
```

7 Glosario.

- **Cloud Storage:** almacenamiento en la nube, es decir, en internet.
- **Agilidad:** capacidad de mover los datos al área adecuada para mejorar su disponibilidad.
- **Escalabilidad:** capacidad de manejar crecientes cargas de trabajo, o de adaptarse para ser capaces de soportarlas.
- **Elasticidad:** capacidad de escalarse mas allá limites impuestos por las condiciones del sistema.
- **Latencia:** Tiempo de espera, normalmente entre que se realiza una petición y se recibe una respuesta.
- **Persistencia:** Si existen distintas copias de un archivo, todas deben ser iguales.
- **Cluster:** Conjunto de sistemas informáticos que actúan como un solo sistema.
- **Nodo:** Sistema capaz de procesar información.
- **Multiusuario:** capacidad de que el recurso sea utilizado por varios usuarios concurrentemente.
- **Servicio distribuido:** servicio ofrecido por un conjunto de sistemas que actúa como un solo sistema.
- **API:** Application programming interface, interfaz de programación de la aplicación.
- **Backend:** infraestructura que ofrece una lógica de negocio interna, que no se relaciona directamente con el usuario.
- **Frontend:** infraestructura ofrecida a un usuario para transmitir información a un sistema que resolverá sus peticiones de forma interna, sin que éste pueda comunicarses con él directamente.
- **Tolerancia a fallos:** capacidad de soportar errores sin dejar sin servicio al cliente.
- **Redundancia de datos:** duplicación de datos, habitualmente para evitar su pérdida.
- **Software duradero:** software capaz de almacenar distintas versiones de un mismo fichero.
- **World Wide Web:** Sistema de enlace de documentos a través de internet.
- **P2P:** Peer to Peer, tecnología de intercambio de datos entre procesos iguales.
- **Amazon EC2 Query Interface:** Interfaz de consultas de la plataforma de Cloud Computing de Amazon.
- **Amazon S3:** Sistema de almacenamiento propietario de la plataforma de Cloud Computing de Amazon.
- **Data store drivers:** Drivers de almacenamiento de datos.
- **Transfer manager drivers:** Drivers de gestión de transferencias.
- **Imagen virtual:** fichero que simula el disco duro de un ordenador para, en conjunto con la tecnología de virtualización, se pueda lanzar un sistema operativo con sus aplicaciones propias.
- **Sector:** Unidad mínima de almacenamiento de un dispositivo de bloques.
- **Buffer:** zona de memoria para alojar información.
- **Cache:** memoria pequeña pero muy rápida, dentro de la jerarquía de memorias.
- **Puntero:** tipo abstracto de dato útil para referenciar direcciones de memoria.
- **Memoria principal:** almacenamiento de instrucciones y datos principal del sistema físico, habitualmente, la memoria RAM.
- **Partición:** Zona de almacenamiento alojada dentro de un dispositivo de bloques.
- **Volumen:** Objetos que representan sectores de disco.
- **Volumen simple:** Objeto que representa sectores de una sola partición.
- **Volumen multipartición:** Objeto que representa sectores de múltiples particiones como uno solo.

Glosario.

- **Enlace simbólico:** enlace entre un punto determinado del dispositivo físico y un fichero alojado en otro punto completamente distinto.
- **Metadatos:** Información adicional de un fichero o directorio, necesario para su adecuada gestión por los distintos subsistemas del sistema operativo.
- **Espacio de nombres:** juego de identificadores que permiten la desambiguación de homónimos.
- **IRQ:** Interrupt request, o número de interrupción del dispositivo para detener la ejecución de código en la CPU.
- **DMA:** Direct Memory Access, metodo de acceso a memoria en el que se delega la tarea del acceso a la información a un dispositivo intermedio que permita al procesador continuar con sus labores sin apenas provocar mínimos retrasos.
- **Socket:** Abstracción utilizada en los sistemas operativos Unix para representar una conexión de red.
- **Cookie:** información temporal de sesión de red, normalmente almacenada en ficheros, para facilitar la navegación web.
- **Spinlock:** Mecanismo de espera activa provocada cuando se requiere adquirir un recurso no disponible.
- **Callback:** Función ejecutada, tan pronto como es posible, para atender la situación registrada por un nuevo evento.
- **Primitiva:** Función básica de un nivel de red, que se ejecuta en todo nodo que tenga dicho nivel activo.
- **Protocolo de red:** Conjunto de reglas que definen una comunicación.
- **Entidad par:** elemento activo del sistema que se comunicará con el nivel homónimo en un sistema distinto.
- **Entorno ad hoc:** entorno de comunicaciones con poca o ninguna planificación.
- **Mutex:** Mecanismo de concurrencia basado en una variable, que recibe, normalmente, dos valores, abierto y cerrado, y en función de él, detiene la ejecución de código hasta que su valor cambie, reanudándola.
- **Jiffies:** Unidad de tiempo mínima del núcleo de Linux.
- **Dispositivos Plug and Play:** dispositivos que, al conectarse al sistema, se configuran automáticamente y se ofrecen sus servicios al usuario sin que éste realice ninguna tarea previa.

8 Referencias.

- Wikipedia:
  - http://en.wikipedia.org/wiki/Cloud_computing_comparison
  - http://en.wikipedia.org/wiki/OpenNebula
  - http://en.wikipedia.org/wiki/Nimbus_(cloud_computing)
  - http://en.wikipedia.org/wiki/Amazon_Elastic_Block_Store
  - http://en.wikipedia.org/wiki/Eucalyptus_%28computing%29
  - http://en.wikipedia.org/wiki/Cloudstack
  - http://en.wikipedia.org/wiki/OpenQRM
  - http://en.wikipedia.org/wiki/Abiquo_Enterprise_Edition
  - http://en.wikipedia.org/wiki/OpenStack
- Amazon:
  - http://aws.amazon.com/s3/
- OpenNebula:
  - http://opennebula.org/documentation:rel3.6
- Nimbus:
  - http://www.nimbusproject.org/docs/2.9/
- CloudStack:
  - http://docs.cloudstack.org/CloudStack_Documentation
- OpenQRM:
  - http://www.openqrm-enterprise.com/news/details/article/in-depth-documentation-of-openqrm-available.html
- Abiquo:
  - http://community.abiquo.com/display/ABI20/Abiquo+Documentation+Home
- OpenStack:
  - http://www.openstack.org/software/openstack-storage/
  - http://docs.openstack.org/api/
  - http://docs.openstack.org/developer/
  - http://docs.openstack.org/developer/swift/development_saio.html
- Linux Kernel Development – Robert Love, Editorial Addison Wesley.
- Linux Device Drivers – Jonathan Corbet, Alessandro Rubini and Greg KroaH-Hartman, Editorial O'Reilly.
- Understanding the linux kernel – Daniel P. Bovet and Marco Cesati.
- Windows Internals – Mark E. Russinovich and David A. Solomon.