Universidad del País Vasco
Euskal Herriko Unibertsitatea

ZIENTZIA ETA TEKNOLOGIA FAKULTATEA
FACULTAD DE CIENCIA Y TECNOLOGÍA

Gradu amaierako lana / Trabajo fin de grado
Ingeniaritza Elektronikoko Gradua / Grado en Ingeniería Electrónica

Sistemas de archivos: Estudio y análisis comparativo de ZFS

Egilea/Autor:
Iñigo José Pérez Gámiz
Zuzendaria/Director:
José María Alcaide Salinas

Leioa, 2022eko uztailaren 25a / Leioa, 25 de julio de 2022

Contents

1 Introducción

Hoy en día vivimos en un mundo cada vez más tecnológico. La información que durante siglos se ha recogido en libros y papeles, se almacena ahora en dispositivos electrónicos. Un ejemplo claro es el de un ordenador que cuenta con dispositivos para almacenamiento de datos. Ahora bien, ¿cómo es capaz el sistema operativo de un ordenador de gestionar el almacenamiento de los datos en dichos dispositivos? La respuesta es que necesita un sistema de archivos para desempeñar dicha tarea.

Existen numerosos sistemas de archivos con distintas finalidades. Algunos de ellos priorizan la compatibilidad, como es el caso de los sistemas de archivos de dispositivos de almacenamiento externos (dígase USB), muchos de ellos la funcionabilidad para ser utilizados en ordenadores a nivel usuario, y otros se centran más en asegurar la integridad de los datos, principalmente aquellos destinados a servidores. Además, cada empresa diseña su propio sistema de archivos de acuerdo a las necesidades de sus sistemas operativos. A modo de ejemplo, Microsoft utiliza NTFS para sus sistemas operativos Windows, mientras que Apple emplea APFS para macOS. Lo que es común a todos ellos es que existe la constante búsqueda por intentar mejorarlos y adaptarlos a nuevas y más demandantes necesidades.

En este contexto, este proyecto estudia los conceptos generales de los sistemas de archivos, para posteriormente centrarse en el análisis de un sistema de archivos particular llamado ZFS. Como veremos, se trata de uno de los sistemas de archivos más sofisticados para el almacenamiento masivo de datos.

En primer lugar, hablaremos de sistemas de archivos de forma general en la Sección 3. Definiremos previamente qué es un archivo y después explicaremos el concepto de sistema de archivos. Posteriormente, nos adentraremos en los fundamentos de los sistemas de archivos. Veremos su funcionamiento y arquitectura, su organización, cómo gestionan los archivos y el espacio de almacenamiento y los servicios que ofrecen a procesos sobre archivos, entre otras muchas cosas. También mencionaremos los principales sistemas de archivos que existen.

Una vez conocidos los principios básicos de los sistemas de archivos, nos dedicaremos a estudiar uno en concreto: ZFS. Dicho estudio tendrá lugar en la Sección 4. Intentaremos contextualizar el uso de ZFS y analizaremos su estructura y principales características. Veremos que se trata de un sistema de archivos que difiere de los más tradicionales. ZFS se olvida de los dispositivos de almacenamiento físicos como tal y los utiliza en conjuntos para trabajar con espacio virtual efectivo. De este modo, puede llevar a cabo redundancia de datos y así asegurar la integridad de los datos. Además, cuenta con otras características como el copy-on-write o los snapshots que le proporcionan una gran versatilidad.

El último paso será comparar en la Sección 5 a ZFS con los sistemas de archivos mencionados previamente en la Sección 3. Construiremos una tabla comparativa y debatiremos las ventajas y desventajas que tiene ZFS frente a los demás sistemas de archivos.

Para finalizar, trataremos de sacar algunas conclusiones relacionadas con los temas analizados.

2 Objetivos

El principal objetivo del proyecto es aprender el funcionamiento y características del sistema de archivos ZFS. Además, también podemos establecer una serie de objetivos secundarios:

1. Entender qué es un sistema de archivos y cuál es su función.
2. Familiarizarse con las características generales de los sistemas de archivos.
3. Conocer algunos de los sistemas de archivos más utilizados.
4. Ser capaz de comparar ZFS con otros sistemas de archivos, destacando sus ventajas e inconvenientes.

3 Sistemas de archivos

En esta sección, vamos a estudiar los sistemas de archivos de forma general. Daremos una definición de sistema de archivos y analizaremos sus principales características. No obstante, antes de nada conviene aclarar el concepto de archivo.

3.1 ¿Qué es un archivo?

A la hora de definir lo que es un archivo, previamente es necesario conocer los conceptos de campo y registro [1].

Un **campo** es la unidad de datos más elemental. Contiene un valor de una tipología determinada y se caracteriza por tener una longitud fija o variable. Un caso concreto puede ser un nombre, una medida de temperatura, los litros de agua contenidos en una botella, etc.

Un **registro** es un conjunto de campos relacionados entre sí. Por ejemplo, un registro *alumno* podría tener varios campos como pueden ser el nombre, la edad, el grado que estudia, su lugar de residencia, etc.

Un **archivo** es una colección de registros parecidos que es considerada como una unidad por los usuarios y aplicaciones [1]. Se caracteriza por los siguientes aspectos [2]:

- Tiene un **nombre** concreto.
- Ocupa un **tamaño** determinado.
- Pertenece a una **tipología**.
- Cuenta con una serie de **fechas** de creación, modificación...
- Tiene una **localización** en los dispositivos de almacenamiento.
- Existe un **dueño** del archivo y un **control de acceso** para su lectura, escritura y ejecución.

Cuando un usuario lleva a cabo una operación en un archivo, realmente está manipulando alguno de sus registros. Entre las operaciones más comunes con estos registros se encuentran insertar un registro, borrarlo, recuperar uno o varios o actualizarlo [1].

La definición de archivo dada hasta ahora es una definición teórica de un archivo con estructura. No obstante, en años recientes, los archivos son un tanto diferentes, puesto carecen de estructura y son simplemente **secuencias de bytes** [3]. Esta idea fue introducida por el sistema operativo Unix (y su antecesor Multics) y ha sido adoptada por otros muchos sistemas operativos hasta el punto de que hoy en día la mayoría tienen dicho concepto de archivo. El objetivo de considerar un archivo como una secuencia de bytes es simplificar. La contrapartida es que la responsabilidad de interpretar dicha secuencia de bytes recae en los programas de aplicaciones. Es decir, no existe una forma universal de interpretar los archivos como ocurre con los que sí tienen estructura.

Los archivos se gestionan por medio de un sistema de archivos. El siguiente paso es, por tanto, analizar los sistemas de archivos. En este proyecto vamos a estudiar los sistemas de archivos considerando que los archivos están compuestos por registros, puesto que se obtiene una buena perspectiva de su funcionamiento.

3.2 ¿Qué es un sistema de archivos?

El **sistema de archivos** es una parte fundamental de un sistema operativo. El término engloba dos conceptos. Por un lado, hace referencia a la forma en que se almacenan los datos en los dispositivos de almacenamiento, es decir, la estructura y el conjunto de normas y métodos para el acceso, búsqueda, escritura, lectura, asignación de espacio y eliminación de archivos [4]. En resumen, se refiere a cómo se gestionan los archivos. Por otro lado, el sistema de archivos es también el encargado de ofrecer servicios a los procesos para llevar a cabo acciones sobre los archivos y directorio. Ejerce de interfaz entre el proceso y los archivos y directorios [3].

Existen diferentes tipos de sistemas de archivos atendiendo a distintos parámetros y finalidades. Todos ellos tienen en común que usan directorios para organizar los archivos y la gran mayoría cuentan con una estructura de árbol en la cual existe un directorio raíz [1].

Normalmente cada sistema operativo tiene un sistema de archivos propio o nativo, lo cual no quiere decir que no sea compatible con otros. A modo de ejemplo, Windows 10 utiliza por defecto NTFS, pero también admite FAT, FAT32 y exFAT. Los dispositivos de almacenamiento también tienen un sistema de archivos concreto. En el caso de un ordenador, los discos usan el sistema de archivos propio de su sistema operativo. En cambio, un dispositivo de almacenamiento externo, como puede ser un USB, puede tener un sistema de archivos diferente que, si es compatible con el sistema operativo del ordenador, no daría ningún problema. En caso de incompatibilidad, se podría establecer en el dispositivo el sistema de archivos necesario.

El sistema de archivos puede estar alojado en una partición del dispositivo de almacenamiento. Esto quiere decir que si, por ejemplo, tenemos el disco duro de un ordenador que cuenta con varias particiones, en cada una de ellas puede haber un sistema de archivos diferente. De este modo, cuando nos referimos a dispositivo de almacenamiento, puede ser una unidad completa o tan solo una partición.

3.3 Cometido de un sistema de archivos

De acuerdo a Ref[5], un sistema de archivos debe cumplir con una serie de requerimientos para un buen funcionamiento general:

- Gestionar correctamente el almacenamiento de los datos y asegurar la validez de los mismos.
- Tratar de minimizar las posibles pérdidas de datos.
- Ser capaz de atender a las demandas del usuario en relación a operaciones con los datos.
- Optimizar el rendimiento con la vista puesta en aumentar la productividad global del sistema y cumplir con las exigencias del usuario lo más rapido posible.
- Proporcionar soporte y un conjunto estándar de rutinas de interfaz de I/O para distintos tipos de dispositivos de almacenamiento.

3.4 Funcionamiento

Se trata ahora de conocer el **funcionamiento** y principales tareas desempeñadas por un sistema de archivos. Cuando un usuario o una aplicación realizan una operación en un archivo, ya sea creación, borrado o modificación del mismo, lo primero que hace el sistema de archivos es ubicar dicho archivo. Para facilitar la localización de los archivos, el sistema de archivos utiliza la estructura basada en directorios. Antes de conceder acceso al archivo, comprueba los permisos que tiene sobre él el usuario o la aplicación. Recordemos además que cuando se realiza una operación en un archivo, realmente se están manipulando sus registros, por lo que el sistema de archivos necesita emplear un método de acceso a dichos registros.

Los usuarios y aplicaciones trabajan con registros, pero los datos se almacenan y se leen de los dispositivos de almacenamiento por bloques, por los que el sistema de archivos debe llevar a cabo internamente las transformaciones necesarias. A la hora de realizar los procesos de I/O de los bloques de datos, debe haber un control sobre ello. Es decir, es necesario saber cómo están asociados los bloques a los distintos archivos. También debe conocerse qué espacio hay libre en los dispositivos para almacenamiento de bloques de archivos y se deben poder hacer reservas. Todo ello sin perder de vista la optimización de los procesos [1].

3.5 Arquitectura

La mayoría de los sistemas de archivos tienen una **arquitectura** basada en niveles, donde cada uno de ellos desempeña alguna tarea determinada. Dicha arquitectura se puede visualizar en la Figura 1.

Los **drivers** o **gestores de dispositivos** controlan los dispositivos de almacenamiento y son los encargados de comenzar y finalizar las operaciones de I/O en los mismos. Cada dispositivo está gestionado por su propio driver. Normalmente estos drivers se consideran parte del sistema operativo.

El **sistema de archivos básico** o **I/O física** ejerce de interfaz entre el sistema operativo y el exterior, si bien también se considera parte del sistema operativo. Lleva a cabo el intercambio y la colocación de los bloques de datos de los archivos en los dispositivos de almacenamiento. También se encarga del almacenamiento intermedio de dichos bloques de datos en la memoria principal. Es importante remarcar que el sistema de archivos básico trabaja con bloques de datos.

El **supervisor básico de I/O** se hace cargo de iniciar y terminar los procesos de I/O con archivos. Su función es planificar en qué dispositivos de almacenamiento tienen lugar estos procesos y el orden de acceso a los dispositivos para optimizar el rendimiento. Dentro de los dispositivos de almacenamiento, reserva espacio para los archivos. También gestiona los buffers de I/O. Una vez más, el supervisor básico de I/O se considera parte del sistema operativo.

La **I/O lógica** es la que hace posible que usuarios y aplicaciones tengan acceso a los registros de archivos. Trabaja a nivel de registros, no de bloques de datos.

Por último, podríamos considerar también como una capa del sistema de archivos al **método de acceso** a los registros de archivos. Existen distintos métodos como es la pila, el secuencial, el secuencial indexado, el indexado o el directo y cada uno de ellos tiene sus propias características como veremos más adelante [1].

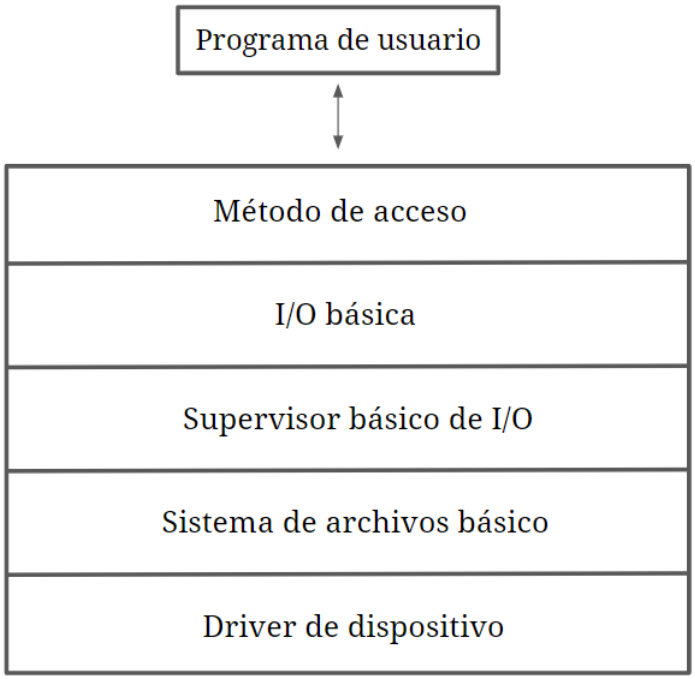

Figure 1: Arquitectura de un sistema de archivos. Figura basada en Ref[5].

3.6 Organización por directorios

Los archivos en un sistema de archivos se organizan por directorios. Un **directorio** normalmente es otro archivo como tal, que contiene mucha información acerca de los archivos a los que está enlazado. Un archivo puede estar ligado a más de un directorio. A continuación se lista la información contenida sobre los archivos en un directorio [1]:

- **Información básica**: nombre y tipo de archivo.
- **Información de dirección**: dispositivo donde se almacena el archivo, dirección de comienzo dentro del dispositivo, espacio usado y espacio asignado.
- **Información sobre control de acceso**: propietario de archivo, información de acceso (nombre de usuario y contraseña) y acciones permitidas (lectura, escritura, ejecución y transmisión por red).
- **Información sobre uso**: fecha de creación del archivo, usuario que lo creó, fecha de última lectura, usuario que lo leyó por última vez, fecha de modificación, usuario que lo modificó por última vez, fecha de último backup y uso actual del archivo.

El método más eficiente para organizar los directorios y archivos es por medio de una **estructura de árbol**, como se puede visualizar en la Figura 2. En ella existe un **directorio raíz** del cual cuelgan otros directorios. A su vez, de estos directorios pueden colgar otros subdirectorios y archivos. Dentro de esta estructura de árbol, los directorios y archivos tienen un **nombre de ruta** o **dirección** (en inglés, path). Todos parten del directorio raíz (por eso se suele omitir escribir su nombre y se empieza directamente por /) y, a partir de ahí, se escriben el resto de nombres de directorios y archivos. A modo de ejemplo, en la Figura 2, el archivo ArchivoB2 tiene una ruta /DirectorioB/SubdirectorioB/ArchivoB2.

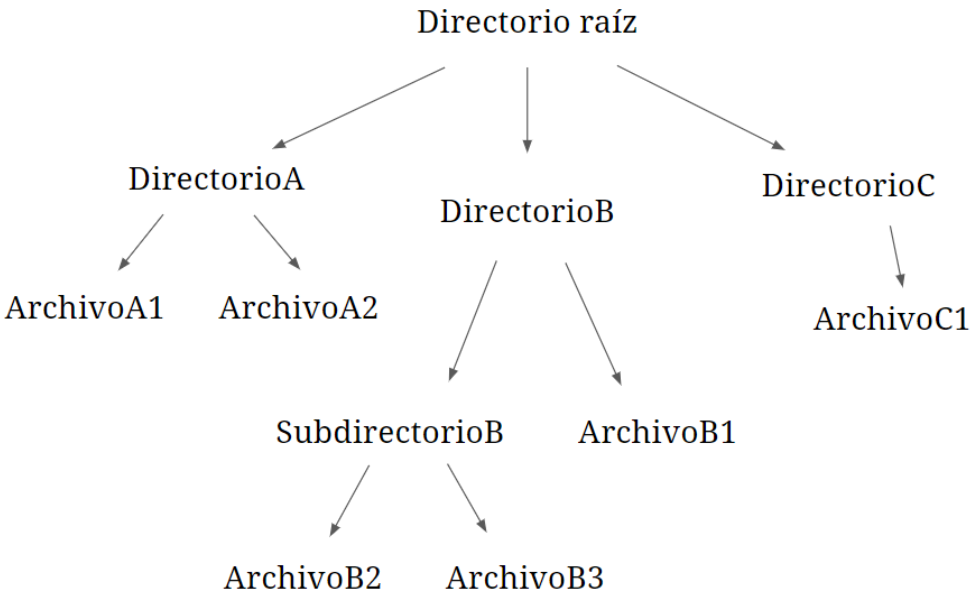

Figure 2: Ejemplo de organización por directorios con estructura de árbol.

Cuando un usuario quiere acceder a un archivo o directorio, no siempre es necesario escribir el nombre de ruta entero. Tan solo hay que escribir la ruta desde el directorio en que se encuentra en ese momento, conocido como **directorio de trabajo** [1].

3.7 Control de acceso

En este apartado tratamos de analizar los principales métodos de control de acceso a archivos que tiene un sistema de archivos.

3.7.1 Permisos

Los sistemas operativos multiusuario requieren unos **permisos** sobre los archivos para evitar acciones indeseadas. Cuando un usuario crea un archivo, por lo general es el propietario de dicho archivo y tiene la posibilidad de conceder o restringir el acceso al mismo a otros usuarios. Los permisos que establece un propietario son a nivel de **usuario individual**, un **grupo de usuarios** o **todos los usuarios**.

Los permisos que normalmente puede tener un usuario sobre un archivo son de **lectura**, **escritura** y **ejecución**. Si tiene permiso de lectura, puede visualizar el contenido del archivo y copiarlo. El permiso de escritura permite también modificar el archivo y, el de ejecución, cargar y ejecutar el programa, pero sin poder copiarlo. A parte de estos permisos, en función del sistema de archivos que se trate, puede haber otros permisos como los de eliminar el archivo, añadirle datos o leer y escribir sus atributos [1].

3.7.2 Listas de control de acceso

Algunos sistemas de archivos cuentan con las llamadas **listas de control de acceso (ACL)**. Una lista de control de acceso es un conjunto de entradas en las que se establece qué usuario o usuarios tienen o no tienen ciertos permisos sobre un archivo. Es decir, es una forma de imponer permisos más a medida. Existen distintos tipos de implementa-

ciones de ACLs, como es el caso de POSIX ACL o NFSv4 ACL. Esta última, por ejemplo, viene incorporada en el sistema de archivos ZFS que posteriormente analizaremos.

3.8 Estructura de archivo y acceso a registros

Como se ha comentado anteriormente, los archivos están formados por registros, que en realidad son los elementos que manipula el usuario al realizar operaciones. Para poder acceder y tratar con estos registros, primero es necesario concocer la **estructura del archivo** y el **método de acceso** a sus registros. En esta sección tratamos de analizar las principales estructuras de archivo [1]. En la Figura 3 se pueden visualizar la mayoría de ellas.

- **Pilas**: los datos se van guardando en registros en el orden cronológico en que llegan. Esto implica que cada resgistro puede tener un tamaño diferente. Además, los campos de los registros también son variables, por lo que los propios campos deben incluir un nombre y longitud que permitan identificarlos.

  Para acceder a un registro o a algún campo concreto hay que ir comprobando todos los registros uno a uno hasta encontrar el deseado. La estructura de pila se suele utilizar para recoger datos sin procesar o de difícil organización.

- **Archivos secuenciales**: todos los registros tienen un mismo estilo predeterminado. Son de un tamaño fijo y están formados por el mismo número de campos, también de tamaño fijo. En este caso, no es necesario incluir identificadores como el nombre y la longitud.

  Todos los registros cuentan con un campo denominado **campo clave** (normalmente es el primero del registro), que permite identificar al registro al que pertenece. Los registros se guardan de forma secuencial atendiendo a sus claves, por orden alfabético si el campo clave contiene texto o por orden númerico si contiene números.

  La estructura secuencial de archivos se suele utilizar en aplicaciones que usan todos los registros como un conjunto. Para aplicaciones que requieren actualización y acceso a registros, no es una estructura recomendada, ya que para encontrar un registro hay que buscar por claves de forma secuencial. Por otro lado, cuando se desean añadir registros al archivo, también hay inconvenientes. Los archivos se escriben en los dispositivos de almacenamiento en bloques de datos siguiendo el mismo orden que el de los registros. Cuando se quiere agregar un registro, este se escribe en un archivo especial con estructura de pila llamado archivo de registro (log file). Permanece ahí hasta que, de forma periódica, se actualiza el archivo secuencial ya escrito en el dispositivo con el archivo de registros para generar un nuevo archivo secuencial que contiene todos los registros (incluidos los nuevos) como un lote.

- **Archivos secuenciales indexados**: tienen como objetivo paliar las carencias de los archivos secuenciales, manteniendo sus principales características. Los registros siguen teniendo un campo clave, pero se incluyen dos elementos. El primero es un **índice de archivo** que permite ejecutar accesos aleatorios de forma rápida y, el segundo, un **archivo de desbordamiento** (overflow).

  En la estructura secuencial indexada más sencilla (un nivel de indexación), el índice es un archivo secuencial. Este archivo índice tiene menos registros que el archivo principal y los que contiene cuentan con dos campos: un campo clave igual que el

de algún registro del archivo principal y un puntero al registro del archivo principal que posee dicho campo clave. Cuando se desea hallar un campo concreto, primero se busca en el archivo índice el valor de clave igual o superior más cercano. A continuación, se sigue la búsqueda en el archivo principal desde donde indica el puntero de dicha clave. Si se desea aumentar el rendimiento, se podrían añadir más niveles de indexación. Esto consiste en crear archivos secuenciales índice de otros archivos secuenciales índice e ir buscando en ellos hasta llegar al archivo principal.

Para poder incorporar un nuevo registro, los registros ya existentes en el archivo principal cuentan con un campo no visible que ejerce de puntero al archivo de desbordamiento. El nuevo registro se agrega al archivo de desbordamiento y, en el registro del archivo principal que le ha de preceder, se hace que el puntero señale al registro recién añadido al archivo de desbordamiento. En caso que el registro predecesor ya estuviera en el archivo de desbordamiento, se actualizaría el puntero en este. Posteriormente, se lleva a cabo de forma periódica la fusión entre el archivo principal y de desbordamiento para formar un solo conjunto de registros.

- **Archivos indexados**: tanto el archivo secuencial como el archivo secuencial indexado tienen la desventaja de que la búsqueda solo se puede realizar por el campo clave. Si se desea poder buscar un registro usando algún campo distinto al clave, hacen falta varios índices, uno por cada tipo de campo que pueda ser buscado. Esto es lo que sucede en los archivos indexados. Son de gran utilidad si se prioriza la rapidez de búsqueda. A los registros se accede a través de estos índices, pero ya no de manera secuencial. De este modo, los registros no tienen por qué estar ordenados y, además, pueden ser de longitud variable.

  Los índices suelen ser de dos tipos. Por un lado, un índice en forma de archivo secuencial que tiene una entrada para cada registro del archivo principal. Por otro, un índice parcial que solo tendrá entradas a los registros donde esté el campo que estamos buscando. Estos índices han de actualizarse cuando se añaden nuevos registros al archivo principal.

- **Archivos directos o de acceso aleatorio**: los registros tienen también un campo clave, pero no se ordenan de forma secuencial. Se puede acceder a un registro de forma directa aplicando un hashing a la clave. Se emplean principalmente cuando el acceso a registros debe ser muy rápido.

3.9 De registros a bloques

Previamente se ha visto que un usuario trabaja a nivel de registros, pero luego los datos se guardan en los dispositivos de almacenamiento en forma de **bloques**. Es decir, debe haber una transformación previa entre ambos. En este apartado vamos a analizar cómo se organizan los registros por bloques.

Los bloques normalmente son de longitud fija. Por otro lado, se ha de establecer un tamaño al bloque. A la hora de la elección de tamaño, se tiene en cuenta también el tamaño de los registros. Cuanto más grande es el tamaño del bloque respecto al de los registros, más registros se procesan en cada I/O de bloque. No obstante, esto no siempre es una ventaja y se debe buscar un equilibrio.

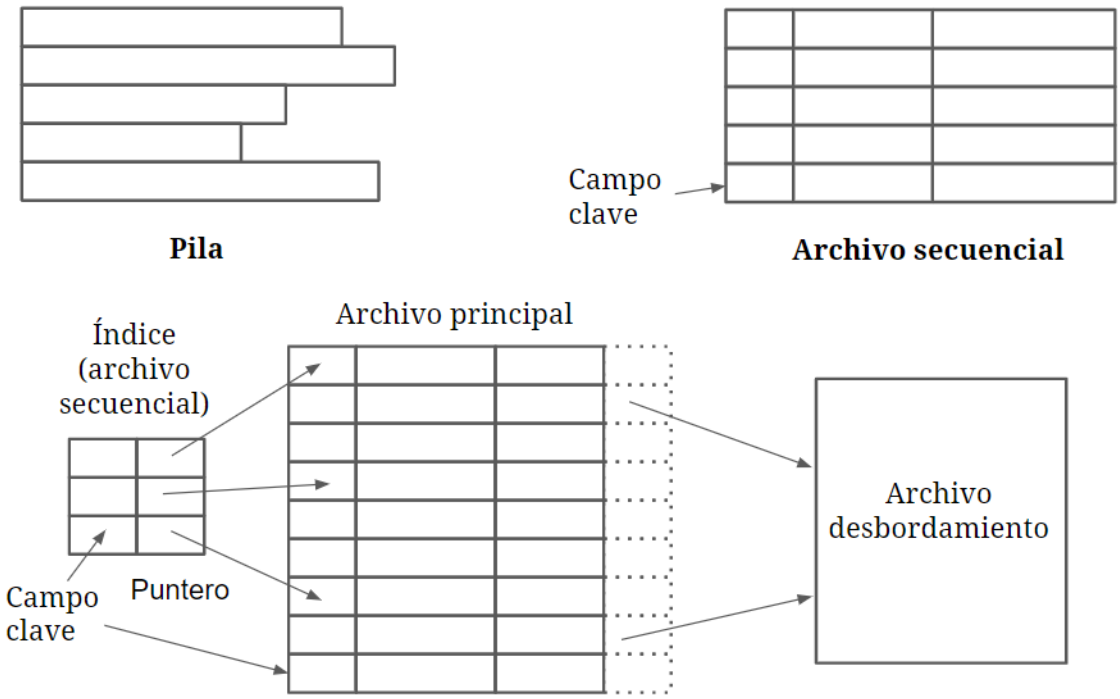

Figure 3: Estructura de archivos y método de acceso a registros. Figura mayoritariamente basada en Ref[1].

Una vez se ha escogido un tamaño de bloque, se puede decir que existen principalmente tres métodos de agrupación de registros en bloques [1]:

- **Bloques con registros fijos**: los registros son todos de la misma longitud y tiene que haber un número entero de ellos por bloque, aunque puede quedar espacio libre.
- **Bloques con registros de longitud variable con tramos**: los registros son de longitud variable y se juntan en el bloque sin dejar espacio libre. Puede ser que algún registro necesite abarcar también parte de otro bloque, en cuyo caso se necesitaría un puntero que indique el tramo donde continúa el registro.
- **Bloques con registros de longitud variable sin tramos**: los registros son de longitud variable, pero no se pueden dividir en tramos, de manera que puede quedar espacio libre sin utilizar en los bloques.

El método de agrupación más eficaz es el de bloques con registros de longitud variable con tramos, ya que aprovecha más el espacio. No obstante, cuenta con el inconveniente de que requiere dos procesos de I/O para los registros divididos en dos bloques.

3.10 Gestión del espacio de almacenamiento

Los archivos se almacenan en los dispositivos de memoria secundaria por bloques. La pregunta que ahora nos debemos hacer es cómo asigna el sistema de archivos los bloques a los archivos y cómo se gestiona el espacio libre. Ese el el objeto de estudio en este apartado. En primer lugar, analizaremos la asignación de espacio a archivos.

3.10.1 Asignación de espacio a archivos

A la hora de asignar conjuntos de bloques a los archivos, el sistema de archivos utiliza la llamada **tabla de asignación de archivos** o **FAT (File Allocation Table)**. Esta tabla contiene información acerca de la ubicación de los bloques de datos de un archivo para poder localizarlos en el dispositivo de almacenamiento.

Antes de entrar en los métodos como tal, se puede establecer una primera diferenciación entre dos estilos de asignación de espacio. Por un lado, está la **asignación previa**, que consiste en reservar un espacio de almacenamiento igual al máximo espacio que se estima va a ocupar el archivo. Por otro lado, nos encontramos con la **asignación dinámica**, la cual trata de ir otorgando espacio al archivo a medida que crece. Esta última es bastante más eficaz, puesto que evita desperdiciar espacio por sobreestimaciones previas.

Procedemos ahora sí a estudiar los principales métodos de asignación de espacio a archivos [1]. En la Figura 4 se pueden visualizar gráficamente.

- **Asignación contigua**: a un archivo recién creado se le otorga un conjunto de bloques contiguos. Se trata de un método de asignación previa y el conjunto de bloques puede tener cualquier longitud. En la tabla de asignación de archivos se ha de indicar el nombre del archivo con el bloque de comienzo y la longitud del conjunto de bloques asignado al archivo.

  Es el tipo de asignación que se utiliza para archivos secuenciales individuales. Tiene como principales inconvenientes la posibilidad de malgastar espacio al ser una asignación previa y que se producen fragmentaciones entre los conjuntos de bloques de

datos de distintos archivos (no se almacenan de forma contigua). En este último caso, habría que compactar de vez en cuando dichos conjuntos de bloques para liberar espacio.

- **Asignación encadenada**: se otorgan bloques de forma individual. Aunque se puede llevar a cabo una asignación previa, normalmente se realiza de forma dinámica a medida que el archivo requiere más espacio. Los bloques que se asignan no son contiguos, sino que pueden tener cualquier ubicación en el dispositivo de almacenamiento. Es por eso que es necesario que cada bloque use un puntero que señale al bloque que le sucede. En la FAT se debe indicar el nombre del archivo con el bloque de inicio y la longitud de bloques del archivo.

  La asignación encadenada es útil para tratar secuencialmente archivos secuenciales. Además, nos hay que preocuparse por una posible fragmentación como ocurría en la contigua. La princial deventaja es que si se desean coger varios bloques de un archivo, hace falta recorrerlos todos de forma secuencial, ya que no están ubicados de manera contigua.

- **Asignación indexada**: tiene como objetivo solventar los problemas de las asignaciones contigua y encadenada. La tabla de asignación de archivos contiene el nombre del archivo y un bloque índice. El bloque índice no guarda datos del archivo, sino que almacena otra tabla. Esta tabla contiene una lista de bloques de inicio con una respectiva longitud. Es decir, contiene los metadatos necesarios para acceder a los bloques y conjuntos de bloques contiguos que sí almacenan los datos del archivo.

  La asignación indexada es el método más utilizado, puesto que es adecuado para archivos de accesos secuencial y directo.

3.10.2 Gestión de espacio libre

Igual de importante que asignar espacio a archivos es gestionar el espacio libre disponible. Para ello, hace falta una propia tabla llamada **tabla de asignación de disco** para tener ubicados los bloques libres. A continuación se presetan algunos de los métodos de gestión de espacio libre más comunes [1].

- **Tabla de bits**: cuenta con un vector de bits en el que cada bit representa a un bloque del dispositivo de almacenamiento. Si el bit está 0, indica que el bloque está libre, y si está a 1, que el bloque está ocupado.

  Es compatible con todos los metodos de asignación de archivos previamente analizados. No obstante, requiere demasiada memoria principal para su utilización.

- **Encadenamiento de conjuntos de bloques libres**: consiste en encadenar conjuntos de bloques libres conociendo su longitud y haciendo uso de punteros.

  También es compatible con todos los metodos de asignación de archivos. El consumo de memoria que hace es muy pequeño porque no requiere tabla de asignación de disco. De todos modos, a medida que se va fragmentando el dispositivo de almacenamiento y van quedando bloques libres aislados, se ralentiza mucho el método.

- **Indexación**: considera a todos los bloques libres como un archivo y utiliza la tabla de asignación de disco del mismo modo que usaba el método de asignación indexada la tabla de asignación de archivos. Una vez más, es compatible con todos los metodos de asignación de archivos.

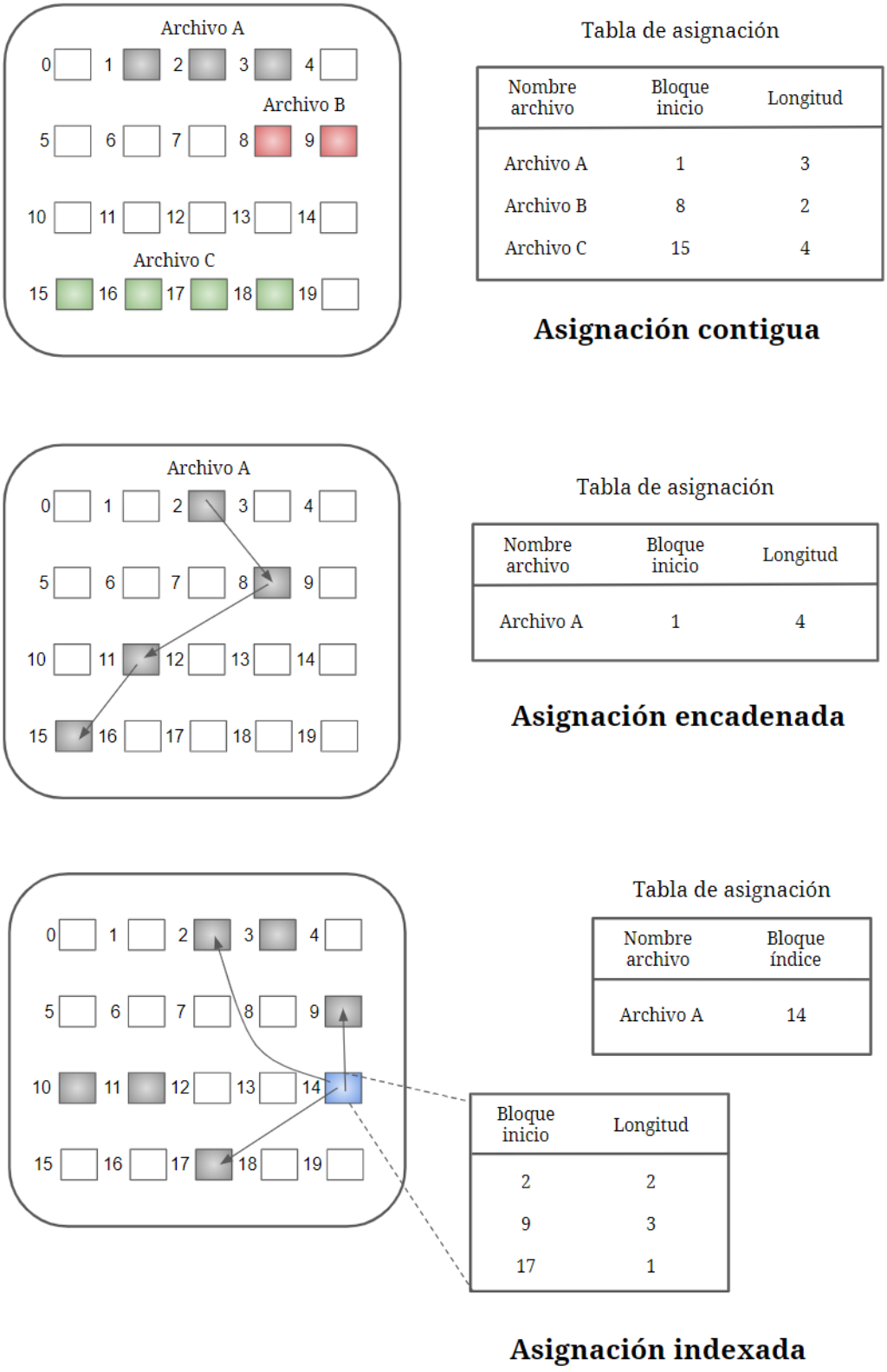

Figure 4: Asignación de espacio a archivos. Figura inspirada en Ref[1].

3.11 Servicios

Cuando se ha definido el sistema de archivos, se ha visto que incluía dos vertientes. Por un lado, la de gestión de archivos, que se ha analizado en profundidad hasta el momento. La otra era la referente a los **servicios** ofrecidos a los procesos para trabajar con archivos y directorios. En este apartado se presentan las operaciones que un sistema de archivo permite realizar a un proceso con archivos y directorios.

3.11.1 Operaciones con archivos

Las principales operaciones que se pueden llevar a cabo con archivos son las siguientes [3]:

- **Crear** un archivo sin datos.
- **Eliminar** un archivo cuando ya no es necesario para liberar espacio del dispositivo de almacenamiento.
- **Abrir** un archivo. Cuando se ejecuta se obtienen las direcciones del dispositivo de almacenamiento donde están almacenados los bloques del archivo.
- **Cerrar** un archivo cuando se deja de trabajar con él. Cuando se cierra el archivo, obligatoriamente se tiene que escribir el último bloque del mismo, independientemente de si está lleno o no.
- **Leer** datos de un archivo. Normalmente se parte de la posición actual y es necesario especificar cuántos datos se desean leer y utilizar un buffer para ello.
- **Escribir** datos en un archivo, generalmente también en la posición actual. Si dicha posición es el final del archivo, este aumenta de tamaño y, si está en medio del archivo, se sobreescriben los datos.
- **Añadir** datos al final de un archivo.
- **Buscar** datos en un archivo. Para ello, hay que tener el cuenta la estructura y método de acceso del archivo (sección 3.8).
- **Obtener atributos** de un archivo, como pueden ser el propietario o la fecha de creación.
- **Establecer atributos** de un archivo.
- **Renombrar** un archivo.

3.11.2 Operaciones con directorios

A continuación se listan las principales operaciones que un proceso puede realizar con directorios [3]:

- **Crear** un directorio vacío.
- **Eliminar** un directorio. Para ello, debe estar vacío.
- **Abrir** un directorio. Es un paso previo necesario para poder leer el directorio y, por ejemplo, obtener una lista de los archivos contenidos en el mismo.
- **Cerrar** un directorio una vez se ha leído.
- **Leer** un directorio una vez ha sido abierto.
- **Renombrar** un directorio.

- **Enlazar** un archivo a un directorio. Es posible que un archivo esté ligado a más de un directorio.
- **Desenlazar** un archivo de un directorio. Si el archivo solo estaba ligado a dicho directorio, entonces se elimina.

3.12 Administrador de volúmenes

Un **administrador de volúmenes** es un software que permite agrupar varios dispositivos físicos (discos duros por ejemplo) o particiones para lograr un volumen virtual que el sistema operativo interpreta como un solo dispositivo [6].

Muchos sistemas de archivos pueden residir únicamente en un dispositivo de almacenamiento, de manera que si se hace uso de más de un dispositivo, es necesario crear un sistema de archivos para cada uno de ellos. Para evitar este tipo de situaciones, un administrador de volúmenes resulta de gran utilidad.

Existen distintos tipos de soluciones para la gestión de volúmenes. Una de las más conocidas es la **tecnología RAID**, que además hace uso de la redundancia y paridad de datos con el objetivo de evitar perder datos por errores de hardware y mejorar el rendimiento. Está disponible para numerosos sistemas operativos.

Otro ejemplo es el **Logical Volume Manager (LVM)**, un software para la administración de volúmenes en sistemas operativos Linux. A diferencia de RAID, el LVM no cuenta con redundancia y paridad para el almacenamiento de datos.

Algunos sistemas de archivos, como es el caso de ZFS, tienen un administrador de volúmenes integrado. Esto se analizará más adelante.

3.13 Tipos de sistemas de archivos

A continuación se van a analizar algunos de los sistemas de archivos más importantes. Dichos sistemas de archivos son los que posteriormente se compararán con ZFS.

3.13.1 FAT

FAT (File Allocation Table o tabla de asignación de archivos) es un sistema de archivos que surgió en 1977 de la mano de Microsoft para el sistema operativo MS-DOS y que ha sido mejorado con el tiempo con las versiones FAT12, FAT16 y FAT32. No obstante, todas ellas han quedado un tanto anticuadas. FAT se utilizaba tradicionalmente con los sistemas operativos Windows hasta Windows ME inclusive [7].

El sistema de archivos FAT se caracteriza por su gran compatibilidad con cualquier sistema operativo. De esta manera, se facilita la transmisión de datos entre sistemas operativos del mismo tipo y distintos. Se utiliza para dispositivos de almacenamiento portables como pueden ser tarjetas de memoria o dispositivos USB. No obstante, cuenta con numerosas limitaciones, la mayoría de ellas relacionadas con el tamaño de almacenamiento y la integridad de los datos, que a día de hoy lo hacen quedar un tanto desfasado [8].

3.13.2 exFAT

exFAT (Extended File Allocation Table), lanzado en 2006 por Microsoft para el sistema operativo Windows XP, se considera el sucesor de los sistemas de archivos de la familia

FAT, concretamente del FAT32. Surge con el objetivo paliar las limitaciones de almacenamiento de su antecesor sin perder la compatibilidad con los sistemas operativos. El sistema de archivos exFAT se emplea fundamentalmente para dispositivos de memoria externa como son los USB [8].

3.13.3 NTFS

NTFS (New Technology File System) es un sistema de archivos desarrollado por Microsoft y que usa en sus sistemas operativos Windows desde 2001. En su momento surgió con la intención de acabar con las limitaciones de memoria del sistema de archivos FAT32 y mejorar el rendimiento. Entre las características clave de NTFS destacamos su seguridad y capacidad para detectar sectores defectuosos en discos y traspasar los datos a otros sectores [9].

3.13.4 APFS

APFS (Apple File System) es el sistema de archivos que utiliza la empresa Apple para sus sistemas operativos macOS, concretamente desde macOS 10.13. Es el sucesor del sistema de archivos HFS+. APFS se caracteriza por tener un administrador de volúmenes integrado que le permite crear espacios de almacenamiento virtuales (llamados contenedores) en los que pueden convivir varios sistemas de archivos. Además, es muy seguro en lo que a la encriptación de archivos respecta y cuenta con las ventajas que proporciona el copy-on-write (más adelante veremos en qué consiste) [10].

3.13.5 UFS

UFS (Unix File System) es el sistema de archivos comúnmente usado para sistemas operativos Unix y de tipo Unix, como pueden ser FreeBSD o Solaris. Algunos de los aspectos más remarcables de UFS son su capacidad de almacenar sistemas de archivos de gran tamaño y las banderas de estado que utiliza para conocer la situación de los sistemas de archivos sin necesidad de hacer un chequeo completo [11].

3.13.6 ext4

ext4 (fourth extended file system) es el sistema de archivos que utilizan los sistemas operativos Linux desde 2006 sucediendo a ext3. Las principales mejoras que presenta frente a sus predecesores es un menor consumo de CPU y un aumento en la velocidad de los procesos de lectura y escritura. Además, es capaz de llevar a cabo una gestión efectiva del espacio de almacenamiento libre [12].

3.13.7 BTRFS

BTRFS (B-Tree File System) es un sistema de archivos desarrollado por Oracle corporation para sistemas operativos Linux. Su objetivo es sustituir al ya mencionado ext4 mejorando alguno de sus puntos débiles, como es el caso del tamaño máximo por archivo permitido o la integridad de los datos. Es un sistema de archivos copy-on-write con administrador de volúmenes integrado [13]. Como veremos más adelante, es bastante similar a ZFS.

4 ZFS

Una vez conocidos los principios generales de los sistemas de archivos, se trata ahora de analizar ZFS y sus características más específicas.

4.1 ¿Qué es ZFS?

ZFS (Zettabyte File System) es un sistema de archivos copy-on-write (COW) y administrador de volúmenes que se caracteriza por tener un concepto de almacenamiento muy distinto al de otros sistemas de archivos más tradicionales. Tiene como principal objetivo velar por la integridad de los datos y protegerlos frente a posibles amenazas o fallos de hardware. Además, ofrece unas limitaciones de capacidad de almacenamiento que son inalcanzables en la vida real.

Los sistemas de archivos tradicionales pueden existir en un único dispositivo de almacenamiento, como puede ser un disco o una partición del mismo. En caso de utilizar dos dispositivos diferentes, es necesario hacer uso de dos sistemas de archivos diferentes. Para evitar esta situación, se puede emplear la tecnología RAID previamente mencionada. En cambio, ZFS, al contar con la gestión de volúmenes integrada, se olvida de la estructura de los dispositivos de almacenamiento físicos y trabaja con espacio virtual efectivo [14].

4.2 Historia

ZFS comenzó a desarrollarse por dos ingenieros de Sun Microsystems en 2001 para su sistema operativo Solaris. No obstante, no fue hasta octubre de 2005 cuando se integró dentro del mismo. Por aquel entonces, Solaris se trataba de un software propietario (no libre), pero dado que Sun Microsystems estaba emergiendo como desarrollador de software de código abierto, decidieron incluir en noviembre de 2005 el código fuente de ZFS dentro del proyecto de libre acceso OpenSolaris [15].

En sus primeros años, ZFS se podía utilizar por medio de ports en otros sistemas operativos como Linux, Mac OS X y FreeBSD. Sin embargo, en 2010 Oracle compró la empresa Sun Microsystems y decidió que ZFS fuera exclusivamente de código cerrado. En este contexto, nacieron proyectos independientes de código abierto de ZFS, entre los que destacamos OpenZFS, que fue anunciado en 2013. Gracias a esto, actualmente existen distribuciones de ZFS para numerosos sistemas operativos de tipo Unix, como es el caso de (aparte del ya mencionado Solaris) Linux, Illumos, sistemas de la familia BSD, OSv o macOS e incluso también con sistemas operativos Windows [15].

ZFS se utiliza de forma nativa en sistemas operativos basados en FreeBSD, como es el caso de TrueNAS o XigmaNAS. La empresa QNAP, que hasta hace poco tiempo utilizaba el sistema de archivos EXT4 para sus sistemas operativos, ha lanzado recientemente QuTS Hero, un sistema operativo basado en ZFS [16]. Otras compañías como Apple estudiaron la posibilidad de utilizar ZFS para sus servidores e incluso llegaron a hacer pruebas, pero finalmente no llegó a buen puerto [17].

4.3 Pools, vdevs y dispositivos de almacenamiento

ZFS propone un método de almacenamiento completamente diferente al de otros sistemas de archivos. Para poder entender correctamente su funcionamiento, es necesario conocer su **estructura**. En la Figura 5 se puede ver un esquema con un ejemplo que facilitará la comprensión durante la lectura.

4.3.1 Pools

El primer eslabón dentro de la jerarquía de almacenamiento es el llamado **pool**. Un pool es una unidad completa e independiente de almacenamiento. Puede existir más de un pool al mismo tiempo. El pool está compuesto por los denominados vdevs, que son dispositivos de almacenamiento virtuales. A su vez, estos vdevs se componen de dispositivos de almacenamiento reales [18].

4.3.2 vdevs

Los **vdevs** son dispositivos de almacenamiento virtuales que constituyen los pools. Un pool puede contener uno o más vdevs. Estos vdevs no son compartidos por los pools, puesto que solo pueden pertenecer a uno. En lo que a la gestión de espacio se refiere, los vdevs se van llenando en función del espacio libre del que disponen de manera que teóricamente se llenen todos al mismo tiempo. No obstante, en versiones más recientes de ZFS, se tiene en cuenta también el nivel de uso de los vdevs. Esto quiere decir que si un vdev tiene más espacio libre que los demás pero se encuentra más ocupado realizando operaciones, entonces se omite temporalmente en el proceso de escritura. De este modo, se mejora el rendimiento reduciendo la latencia [18].

Existen diferentes tipos de vdevs atendiendo a su función. La mayoría de ellos son **data vdevs** destinados al almacenamiento de datos. Todos los pools requieren la existencia de al menos un data vdev. Estos data vdevs pueden tener distintos diseños internos. A continuación se analizan los más importantes:

- **Stripe vdev**: se trata de uno o varios dispositivos de almacenamiento físicos sin ninguna redundancia de datos. La capacidad total de almacenamiento es igual a la suma de memoria de todos los dispositivos. Cuando solo hay un dispositivo, se le suele llamar **single device vdev** [18].

  Podemos catalogarla como una solución barata ya que, al no haber redundancia, se necesitan pocos dispositivos de almacenamiento. Además, los procesos de lectura y escritura son rápidos [19].

  Puede llegar a resultar peligroso, puesto que en caso de fallo de cualquiera de los dispositivos de almacenamiento, se perdería la información contenida. De hecho, no suele ser recomendable usarlo. El stripe vdev es similar al RAID0 [19].

- **Mirror vdev**: contiene un mínimo de 2 dispositivos de almacenamiento. Cada bloque de datos se almacena en todos los dispositivos, es decir, la información está replicada en todos los dispositivos, de ahí el término mirror. De este modo, solo se pierde información en caso de que fallaran todos los dispositivos de almacenamiento [18]. La capacidad máxima del mirror vdev será la del dispositivo físico de menor capacidad [16].

  El coste de hardware es muy alto debido a que la mitad del espacio total proporcionado por los dispositivos se destina a redundancia. De hecho, es el que tiene mayor coste de hardware, teniendo en cuenta el poco aprovechamiento del espacio que realiza. Sin embargo, los procesos de lectura son rápidos y, los de escritura, relativamente también [19].

  Lo más común es encontrar mirrors de 2 dispositivos, aunque también se pueden emplear tres o más para aumentar el rendimiento de lectura de datos y protección

en caso de fallo [18]. No obstante, la rapidez de escritura dismunuiría cuantos más dispositivos hubiera. Se puede decir que su funcionamiento es parecido al de un RAID1 tradicional [19].

- **RAIDz o RAIDz1 vdev**: se necesitan un mínimo de 3 dispositivos de almacenamiento. Del espacio total proporcionado por los dispositivos, el espacio correspondiente a 1 dispositivo se destina a la redundancia de datos, mientras que el resto del espacio es para almacenamiento. Es importante remarcar que es el espacio correspondiente a 1 dispositivo, no un dispositivo como tal. El RAIDz soporta el fallo de 1 dispositivo de almacenamiento sin pérdida de datos.

  El coste de hardware es relativamente alto, pero no mayor que en el mirror. Aunque hagan falta más dispositivos de almacenamiento con respecto al mirror vdev, se aprovecha más el espacio teniendo la misma redundancia. Los procesos de lectura y escritura son más lentos. El funcionamiento del RAIDz es similar al del RAID5 [19].

- **RAIDz2 vdev**: contiene un mínimo de 5 dispositivos de almacenamiento. El espacio de 2 dispositivos de almacenamiento se utiliza para redundancia de datos y, el resto del espacio, para almacenamiento. De este modo, se logra tolerancia al fallo de un máximo de 2 dispositivos de almacenamiento.

  El coste de hardware es también alto y, los procesos de lectura y escritura, lentos. El RAIDz2 es similar al RAID6 [19].

- **RAIDz3 vdev**: hacen falta como mínimo 7 dispositivos de almacenamiento. El espacio de 3 dispositivos de almacenamiento se destina a redundancia de datos y, el resto del espacio, se usa para almacenar. Pueden fallar un máximo de 3 dispositivos sin que se pierda información.

  Una vez más, el coste de hardware es alto y, los procesos de lectura y escritura, lentos [19].

Más allá de los data vdevs, hay otros tipos de vdevs que iremos viendo más adelante. Cuando se crea un pool, es necesario incluir al menos un data vdev. Este data vdev puede tener cualquiera de los diseños mencionados. Una vez creado el pool, es posible añadir otros data vdevs y aumentar el tamaño del mismo. Estos nuevos data vdevs han de tener el mismo diseño que los ya existentes en el pool. Por ejemplo, si tenemos un pool con un RAIDz vdev de tres dispositivos, podríamos añadir otro RAIDz vdev de tres dispositivos para formar un stripe de dos RAIDz vdevs. Si en vez de uno añadiéramos dos RAIDz vdevs de tres dispositivos, tendríamos un stripe de tres RAIDz vdevs. Y así sucesivamente para todos los diseños y siempre y cuando contemos con dispositivos de almacenamiento físicos suficientes [20].

4.3.3 Dispositivos de almacenamiento o devices

Los **dispositivos de almacenamiento** o **devices** son dispositivos físicos que disponen de acceso aleatorio a sus bloques y forman los vdevs. Normalmente suelen ser discos duros o sólidos o incluso sus particiones. Se suele decir que todo dispositivo reconocido en la carpeta /dev con acceso aleatorio a bloques funciona como device [18].

Una vez creado un vdev, de forma general no se le pueden añadir más dispositivos físicos para incrementar el espacio disponible. Tampoco es posible retirar dispositivos de

almacenamiento del vdev. No obstante, existen dos casos de excepción en los que sí se puede añadir. Por un lado, es posible agregar un dispositivo a un single device vdev para convertirlo en un mirror vdev. Por otro lado, si se añade un dispositivo a un mirror vdev, este se transforma en un RAIDz [21].

Conviene apuntar que en el proceso de escritura de datos, ZFS emplea lo que se conoce como striping. El **striping** es una técnica por la cual los datos a almacenar se dividen en bloques que posteriormente se distribuyen a lo largo de todos los dispositivos físicos del vdev de acuerdo a unas reglas.

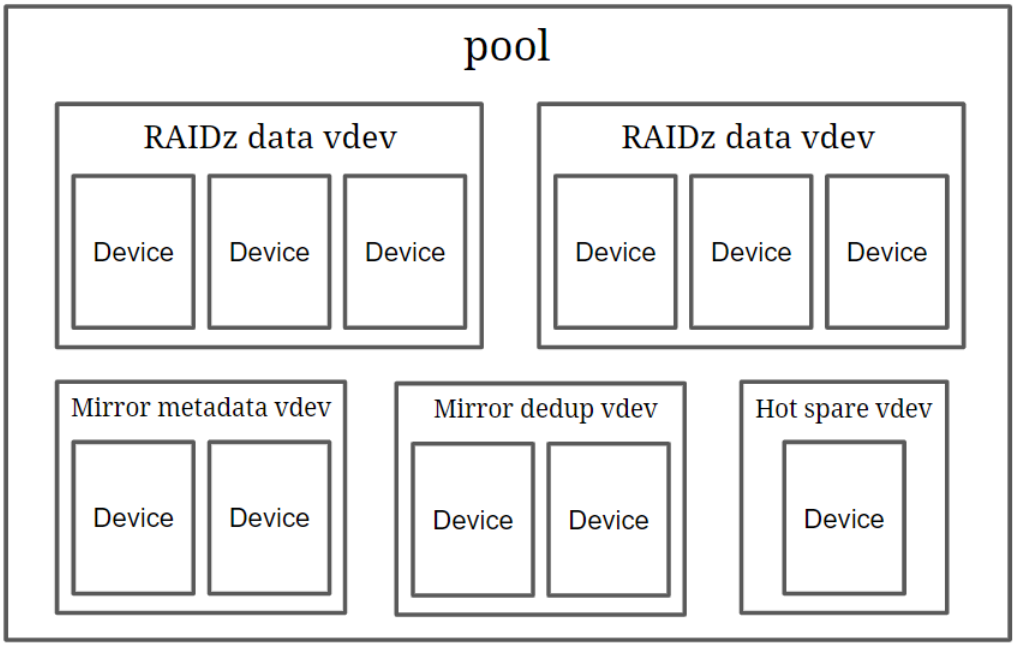

Figure 5: Ejemplo de un pool para visualizar la estructura de almacenamiento en ZFS.

4.3.4 Ejemplo

Vamos a crear a modo de ejemplo un pool llamado *poolex* y que cuenta con un data mirror vdev formado por 2 dispositivos de almacenamiento. Este pool lo vamos a seguir utilizando para el resto de ejemplos del trabajo. Para ello, hacemos uso del sistema operativo TrueNAS basado en FreeBSD.

```
root@truenas[~]# zpool create poolex mirror da1 da2
root@truenas[~]# zpool status poolex
  pool: poolex
 state: ONLINE
config:

        NAME        STATE     READ WRITE CKSUM
        poolex      ONLINE       0     0     0
          mirror-0  ONLINE       0     0     0
            da1     ONLINE       0     0     0
            da2     ONLINE       0     0     0

errors: No known data errors
```

Una vez creado el pool, podemos añadirle, por ejemplo, otro data mirror vdev formado por 2 dispositivos de almacenamiento. También podemos visualizar el estado del pool:

```
root@truenas[~]# zpool add poolex mirror da3 da4
root@truenas[~]# zpool status poolex
  pool: poolex
 state: ONLINE
config:

        NAME        STATE     READ WRITE CKSUM
        poolex      ONLINE       0     0     0
          mirror-0  ONLINE       0     0     0
            da1     ONLINE       0     0     0
            da2     ONLINE       0     0     0
          mirror-1  ONLINE       0     0     0
            da3     ONLINE       0     0     0
            da4     ONLINE       0     0     0

errors: No known data errors
```

4.4 Datasets, bloques y sectores

Una vez conocida la estructura básica de almacenamiento de ZFS, toca analizar cómo se organizan y almacenan los datos. Para ello, es necesario saber qué son los datasets, bloques y sectores.

4.4.1 Datasets

Un **dataset** es una forma génerica de referirse a un sistema de archivos o un zvol. Es decir, puede ser cualquiera de los dos elementos [22].

Un pool puede contener uno o más sistemas de archivos. Sin embargo, un sistema de archivos no puede pertenecer a dos pools distintos. Al crear un pool, automáticamente se crea un sistema de archivos con el mismo nombre y generalmente se monta debajo del directorio raíz / del sistema operativo. Los sistemas de archivos pueden aumentar su tamaño hasta completar toda la memoria restante en el pool. Cuando se trabaja con más de un sistema de archivos en un mismo pool, puede ser interesante utilizar reservas y cuotas de espacio para gestionar la memoria disponible [21].

Existe la posibilidad de crear y montar sistemas de archivos dentro de otros, de manera que un sistema de archivos padre puede tener un sistema de archivos hijo. A modo de ejemplo, un sistema de archivos padre se puede montar justo debajo del sistema de archivos del pool al que pertenece, /pool/padre, y el sistema de archivos hijo debajo del padre, /pool/padre/hijo.

Un zvol es un dataset que representa a un dispositivo de bloques. Es decir, se puede considerar como un espacio reservado para almacenamiento. Dentro de un zvol se pueden crear sistemas de archivos. La gran ventaja que presenta este tipo de dataset es que los sistemas de archivo que se crean en su interior no tienen por qué ser ZFS, pero sí se mantienen las características de ZFS. Por ejemplo, se podría crear un sistema de archivos ext4 dentro de un zvol y seguir conservando todas las propiedades de ZFS. Los zvol son una característica propia de ZFS [18].

Continuando con nuestro pool *poolex*, podemos crear dentro de él, por ejemplo, un sistema de archivos llamado *filesys1* y un zvol de 2GB llamado *zvol1*:

```
root@truenas[~]# zfs create poolex/filesys1
root@truenas[~]# zfs create -V 2gb poolex/zvol1
root@truenas[~]# zfs list -r poolex
NAME              USED  AVAIL     REFER  MOUNTPOINT
poolex           2.06G  6.65G       24K  /poolex
poolex/filesys1    24K  6.65G       24K  /poolex/filesys1
poolex/zvol1     2.06G  8.72G       12K  -
```

4.4.2 Bloques

Los **bloques** son las unidades básicas de almacenamiento en ZFS. Toda la información que se almacena, incluidos los metadatos, se divide en bloques. Los archivos se dividen en bloques. En un bloque solo puede haber datos de un archivo. Es decir, dos archivos con diferentes datos no pueden compartir un mismo bloque para almacenar [18].

Cada bloque se ubica en un solo vdev. La distribución de los bloques por los vdevs se lleva a cabo de manera que todos los vdevs teóricamente se llenen al mismo tiempo, como se ha explicado anteriormente. El tamaño de los bloques está definido por la propiedad **recordsize** del sistema de archivos al que pertenecen (para los zvols, esta propiedad se llama **volblocksize**). Por defecto, el tamaño suele ser de 128KB. No obstante, es posible modificarlo a valores entre 4KB y 1MB en caso de que fuera necesario almacenar archivos más pequeños o grandes. A la hora de cambiar el recordsize hay que tener en cuenta ciertos factores como puede ser el rendimiento. Si se desea almacenar un archivo de, por ejemplo, 3KB, lo más eficiente sería utilizar un bloque de 4KB, pero, a pesar de todo, habría 1KB que quedaría en desuso. En el caso en que un archivo por tamaño requiera más de un bloque, todos los bloques que lo componen serían del mismo tamaño (el dado por recordsize), incluso el último bloque en almacenarse, que podría tener mucho espacio libre sin emplear [18].

En nuestro ejemplo, si quisiéramos que, por ejemplo, el sistema de archivos *filesys1* tuviera un recordsize concreto, podríamos establecerlo al crearlo o a posteriori:

```
root@truenas[~]# zfs create -o recordsize=4kb poolex/filesys1
root@truenas[~]# zfs set recordsize=1mb poolex/filesys1
```

4.4.3 Sectores

Los **sectores** son la unidad física más pequeña que se puede leer o escribir de los dispositivos de almacenamiento reales. Los bloques se almacenan por sectores. Tradicionalmente, los discos duros han tenido sectores físicos de 512B, aunque hoy en día es más frecuente verlos de 4KB o incluso de mayor tamaño, como es el caso de los sectores de 8KB de los discos sólidos. Normalmente el tamaño de los sectores se mide por lo que se denomina **ashift**. Este ashift es el exponente del tamaño del sector expresado en potencia de 2 [18]:

$$\text{Tamaño} = 2^{\text{ashift}}$$

Es decir, si el ashift es igual a 9, el tamaño del sector es $2^9 = 512$B. En ZFS existe la posibilidad de configurar el ashift para adecuar el tamaño de los sectores lógicos a almacenar al tamaño de los sectores físicos del disco en que se van a escribir o leer. Dicha

configuración se lleva a cabo a nivel del vdev, no del pool. Una vez se establece un ashift determinado a un vdev y este se añade a un pool, no se puede revertir lo establecido. Es por eso que hay que tener cuidado, ya que una mala elección del ashift puede provocar el mal funcionamiento del pool y la necesidad de destruirlo.

Si se escoge un ashift demasiado bajo, el tamaño de los sectores de datos a leer o escribir es mucho menor que el tamaño de los sectores del dispositivo de almacenamiento. Esto provoca que, por ejemplo, en un proceso de escritura, sea necesario escribir los sectores lógicos de datos en los sectores del dispositivo en numerosos pasos, entre los cuales a su vez también se tendrían que leer los ya almacenados para gestionar el espacio. De este modo, se ralentiza enormemente la escritura y lectura de datos. Por el contrario, si se establece un ashift alto, no se aprecian penalizaciones en lo que al rendimiento se refiere [18].

Ejemplo de elección de un ashift determinado al añadir un data vdev al pool:

```
root@truenas[~]# zpool add -o ashift=12 poolex mirror da3 da4
```

4.5 Copy-on-write

El **copy-on-write (COW)** es una característica de ZFS que resulta fundamental para muchas operaciones. Cuando modificamos un archivo, realmente estamos alterando alguno de sus bloques. En ZFS, cuando queremos modificar los datos de un bloque, los datos nuevos no se reescriben en el mismo. Se crea una copia del bloque y ahí es donde se llevan a cabo los cambios. De ahí el nombre de copy-on-write. En la Figura 6 se puede visualizar cómo se crea una copia del bloque original para poder alterarlo. El bloque antiguo permanece en el vdev, pero se modifica la tabla de asignación de archivo para desvincularlo del archivo y vincular el nuevo. Una vez el bloque antiguo no es necesario, podría eliminarse en caso de requerir espacio libre [21].

El proceso de desvinculación de un bloque del archivo para la vinculación del nuevo se lleva a cabo en una sola operación. De este modo, se evita que si por cualquier motivo se apaga el sistema operativo entre la desvinculación de uno y la vinculación del otro, no se pierda la información. De hecho, esto es posible porque las modificaciones se hacen en la copia del bloque, no en el original. En caso de fallo entre las dos operaciones se volvería al bloque original [18].

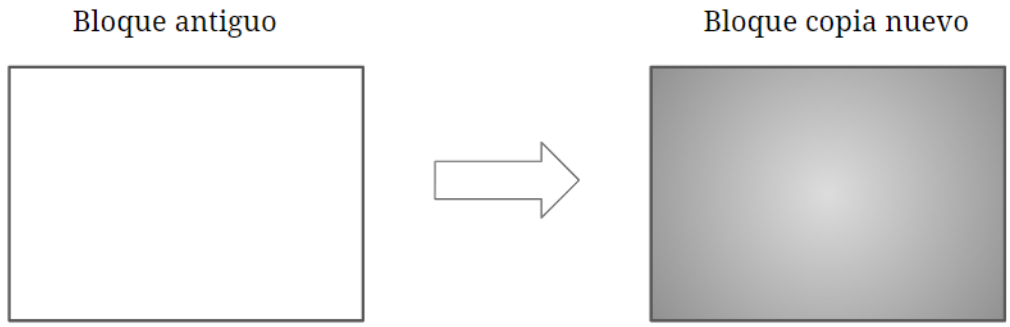

Figure 6: Copy-on-write al modificar un bloque de datos. Figura basada en Ref[21].

4.6 Snapshots

Un **snapshot** es una intantánea de un dataset, es decir, una copia del mismo en un instante determinado solo para lectura. El dataset puede ser el sistema de archivos del pool (el pool entero), un sistema de archivos dentro del pool o incluso un zvol [20]. Gran parte de las ventajas de los snapshots se deben al copy-on-write [21]. Cuando se hace un snapshot de un dataset, se crea una copia del dataset, pero esta no consume espacio extra, pues está referenciada a los mismos bloques que el dataset original. Si tras hacer el snapshot se modifica el bloque de algún archivo, de acuerdo al copy-on-write los cambios se llevan a cabo en un bloque copia y el bloque original se desvincula del archivo para vincular el nuevo. La diferencia en este caso es que los bloques antiguos no se eliminan, puesto que están referenciados al snapshot que se había creado. En este escenario, el snapshot empezaría a consumir espacio, que se correspondería con el tamaño de los bloques antiguos desvinculados del archivo del dataset.

Existe la posibilidad de realizar snapshots de forma recursiva. Esto quiere decir que se puede hacer un snapshot de un dataset que incluya a todos sus datasets hijos. También es interesante hacer snapshots de forma periódica. ZFS otorga la posibilidad de programar snapshots cada cierto tiempo. Esto puede ser útil en tareas de larga duración si se desea llevar un registro de los cambios realizados en el dataset. Se puede establecer un tiempo de vida para los snapshots e incluso pedir a ZFS que realice un snapshot a pesar de que no haya habido cambios en el dataset desde el último, lo cual no consumiría espacio extra [20].

En nuestro ejemplo, vamos a realizar un snapshot del sistema de archivos *filesys1*. Para ello, debemos escribir el nombre del dataset seguido de un @ y el nombre del snapshot:

```
root@truenas[~]# zfs snapshot poolex/filesys1@snap
root@truenas[~]# zfs list -t snapshot -r poolex
NAME                  USED  AVAIL     REFER  MOUNTPOINT
poolex/filesys1@snap     0B      -       24K  -
```

4.6.1 Clone y promote

Una característica relevante de ZFS asociada a los snapshots es el **clone**, es decir, el clon. Un clone es una copia del snapshot de lectura en forma de dataset en el que también se puede escribir. Es importante remarcar que el clone no es un snapshot, sino que es un dataset montado en cualquier punto que se desee dentro del pool. No obstante, mantiene una dependencia respecto al snapshot del que proviene, el cual debe existir durante toda la vida del clone. De hecho, para poder destruir un snapshot, primero hay que destruir todos sus clones [23].

En un principio, el clone está ligado a los mismos bloques de datos que el snapshot del que proviene, por lo que no ocupa espacio extra. Si se llevan a cabo modificaciones en el clone, los bloques nuevos de datos son los que consumen espacio de almacenamiento. También existe la posibilidad de realizar snapshots del clone.

Otro aspecto importante de ZFS es la capacidad de hacer un **promote** a un clone, es decir, promocionarlo. Esto consiste en cambiar los papeles entre el clone dataset y el dataset original del que proviene. Dicho de otro modo, el clone pasa a ser el dataset original y el dataset que en un principio era el original se convierte en el clone. Una vez hecho el promote, el dataset inicial se puede destruir, ya que no tiene un clone que dependa de él, sino que él mismo es el clone [24].

Siguiendo con nuestro ejemplo, podemos hacer ahora un clone a partir del snapshot previamente realizado. Debemos elegir un nombre para el clone y dónde montarlo:

```
root@truenas[~]# zfs clone poolex/filesys1@snap poolex/filesys1clone
root@truenas[~]# zfs list -r poolex
NAME                   USED  AVAIL     REFER  MOUNTPOINT
poolex                2.06G  6.65G       24K  /poolex
poolex/filesys1         24K  6.65G       24K  /poolex/filesys1
poolex/filesys1clone     0B  6.65G       24K  /poolex/filesys1clone
poolex/zvol1          2.06G  8.72G       12K  -
```

Una vez tenemos el clone, podemos hacer un promote:

```
root@truenas[~]# zfs promote poolex/filesys1clone
root@truenas[~]# zfs list -r poolex
NAME                   USED  AVAIL     REFER  MOUNTPOINT
poolex                2.06G  6.65G       24K  /poolex
poolex/filesys1          0B  6.65G       24K  /poolex/filesys1
poolex/filesys1clone    24K  6.65G       24K  /poolex/filesys1clone
poolex/zvol1          2.06G  8.72G       12K  -
root@truenas[~]# zfs list -t snapshot -r poolex
NAME                         USED  AVAIL     REFER  MOUNTPOINT
poolex/filesys1clone@snap      0B      -       24K  -
```

Como podemos observar, el archivo que ahora ocupa espacio es el clone (*filesys1clone*). Además, el snapshot que habíamos realizado de *filesys1* ahora pasa a ser un snapshot de *filesys1clone*. No obstante, para el resto del proyecto vamos a considerar que no hemos llevado a cabo el promote y que el sistema de archivos original sigue siendo *filesys1*.

4.6.2 Rollback

Otra capacidad de ZFS asociada a los snapshots es poder hacer un **rollback**. Un rollback de un snapshot permite restaurar el dataset en el punto en que se hizo el snapshot. Es decir, se deshacen todas las modificaciones realizadas en el dataset desde el snapshot.

Tan solo se puede realizar un rollback del último snapshot realizado al dataset. Si se desea regresar a un snapshot anterior, es necesario destruir todos los snapshots intermedios. Hay que tener cuidado porque puede que alguno de estos snapshots intermedios tenga algún clone, en cuyo caso deberían ser destruidos antes que el snapshot [25].

A continuación se presenta el código necesario para hacer un rollback con el snapshot del sistema de archivos *filesys1* en nuestro ejemplo:

```
root@truenas[~]# zfs rollback poolex/filesys1@snap
```

4.6.3 Entornos de arranque

En los sistemas operativos TrueNAS existen los llamados entornos de arranque. Los **entornos de arranque** (en inglés, boot environments) son sistemas de archivos ZFS destinados a iniciar el sistema. Originalmente, ZFS tiene un entorno de arranque por defecto. Cuando se instala por primera vez ZFS, se crea automáticamente un pool de arranque llamado boot-pool. En dicho pool se almacenan los entornos de arranque.

Un caso de especial y que resulta de gran utilidad, es crear entornos de arranque con snapshots de otros entornos de arranque. Esto suele ser conveniente hacerlo cuando se va

a llevar a cabo una actualización del sistema operativo. Cuando termina la actualización, automáticamente se crea un entorno de arranque. No obstante, puede que el resultado de la actualización no sea el deseado. En caso de que se hubiera hecho un snapshot previamente, se podría arrancar el sistema en tal punto y deshacer los cambios de la actualización [20].

4.7 Replicación

ZFS permite realizar copias de seguridad (en inglés, backups) de datasets con distintos fines. Este fenómeno se conoce como **replicación**. Las copias de seguridad pueden ser locales o remotas. Los datasets pueden ser sistemas de archivos de pools enteros, sistemas de archivos dentro de pools o zvols. Además, se inclyen también los metadatos. Cuando se hace una copia de seguridad de un dataset, realmente se está almacenando un snapshot del mismo en otro lugar. Por ello, es necesario realizar previamente un snapshot. Esta copia de seguridad puede ser recursiva, es decir, que se incluyan los snapshots de los datasets hijos del dataset seleccionado [20].

Una **replicación local** tiene lugar dentro del propio sistema. Consiste en almacenar un snapshot de un dataset en otro pool distinto dentro del mismo sistema. Protege frente a una posible pérdida del pool inicial, pero no en caso de fallo de todo el sistema. Es por eso que es más recomendable realizar copias de seguridad remotas [20].

Una **replicación remota** permite almacenar snapshots de datasets en sistemas ajenos. El proceso se lleva a cabo por medio de un protocolo SSH [20]. Recordemos que SSH es un protocolo que proporciona una comunicación segura entre dos sistemas a modo cliente-servidor, en el cual el cliente accede al servidor de forma remota. Una vez se ha establecido la conexión entre los dos sistemas, hay que imponer una orden de envío (send) al sistema de origen y de recibo (receive) al sistema de destino [26]. ZFS otorga la posibilidad de encriptar la información enviada. Si bien la transmisión se ralentiza, se gana notablemente en seguridad, cosa que siempre es recomendable [20].

Resulta de gran interés programar replicaciones remotas periódicas. De este modo, se asegura que se actualicen las copias de seguridad en otro sistema cada cierto tiempo. Estas replicaciones normalmente tienen lugar de forma incremental. Esto quiere decir que cada vez que se hace un snapshot del dataset para enviarlo al sistema destino, tan solo se envían los datos diferentes respecto del último snapshot replicado. Se trata una forma de optimizar el uso de almacenamiento en el sistema receptor. A la hora de programar las replicaciones periódicas, hay que indicar cada cuánto tiempo se realizan y en qué momento del día. Puede ser cada hora, diariamente, semanalmente, mensualmente, o lo que hiciera falta. La hora del día es al gusto del programador, pero suele ser recomendable establecerla en un momento en que no se esté haciendo uso del almacenamiento. También es conveniente imponer a los snapshots que se envían un periodo de vida en el sistema remoto. De esta manera, se evita que permanezcan indefinidamente snapshots que han quedado obsoletos [20].

A continuación se enumeran las principales finalidades de la replicación remota [27]:

- Posibilidad de recuperar fácilmente los datos almacenados en caso de pérdida por fallo del sistema original. Se asegura la integridad de los datos ante hipotéticos errores de hardware o ataques informáticos.

- Distribución de datos por numerosos servidores remotos para que clientes de todo el mundo tengan acceso a la información.

- Trasladar datos de un sistema a otro si se va a continuar trabajando en este último y se desea mantener el hardware en el sistema inicial. Más tarde veremos otro modo de migrar datos que consiste en mover el hardware de un sistema a otro.

Hay que recordar que al sistema remoto se envían snapshots, por lo que si queremos manipular los datos en ese sistema habría que usar un clone como hemos visto anteriormente.

Vamos ahora a llevar a cabo una replicación local con nuestro ejemplo. Vamos a copiar el sistema de archivos *filesys1* a otro pool dentro del mismo sistema. Para ello, hemos de crear un nuevo pool llamado *newpool* y enviar el snapshot de *filesys1* usando el comando "pipe" | . El sistema de archivos *filesys1* se crea en el nuevo pool con el nombre *fs1*:

```
root@truenas[~]# zpool create newpool mirror da5 da6
root@truenas[~]# zfs send poolex/filesys1@snap | zfs receive newpool/fs1
root@truenas[~]# zfs list -r newpool
NAME          USED  AVAIL     REFER  MOUNTPOINT
newpool       130K  4.36G       24K  /newpool
newpool/fs1    24K  4.36G       24K  /newpool/fs1
```

Podemos hacer la misma replicación del sistema de archivos *filesys1* pero de forma remota a un sistema ajeno llamado *sys2*. Una vez más, hemos de usar el comando "pipe", así como una instrucción de protocolo SSH:

```
root@truenas[~]# zfs send poolex/filesys1@snap | ssh sys2 zfs recv newpool/fs1
```

Como se ha comentado anteriormente, la replicación remota puede ser incremental. En este caso, debemos acompañar la orden send con -i y en el comando de envío se incluyen dos variables: el último snapshot replicado (*filesys1@snap*) y el nuevo snapshot que se desea replicar (*filesys1@newsnap*). De este modo, tan solo se envía la diferencia de datos entre ambos snapshots:

```
root@truenas[~]# zfs snapshot poolex/filesys1@newsnap
root@truenas[~]# zfs send -i poolex/filesys1@snap poolex/filesys1@newsnap |
                 ssh sys2 zfs recv newpool/fs1
```

El sistema de archivos *fs1* debe existir antes del envío de la información. En los dos casos anteriores (replicaciones local y remota no incremental), el sistema de archivos *fs1* no era necesario que existiera antes de la replicación [26].

4.8 Exportar un pool

ZFS permite **exportar** un pool de un sistema a otro. Esta exportación consiste en trasladar los dispositivos de almacenamiento a otro sistema y tener exactamente la misma información del pool que en el sistema original. Para ello, se deben seguir los siguientes pasos [28]:

1. Enviar una orden de exportar del pool (zfs export) al sistema original. Esta orden es necesaria antes de desconectar los devices para evitar perder datos como consecuencia de que algún proceso de lectura o escritura estuviera en plena ejecución.

2. Desconectar los dispositivos de almacenamiento de dicho sistema.

3. Conectar los dispositivos al nuevo sistema. El orden de conexión no es trascendente, ya que con los metadatos ZFS es capaz de interpretar la información almacenada.

4. Enviar una orden de importar el pool (zfs import) al nuevo sistema.

4.9 Corrección de errores

4.9.1 Memoria ECC

ZFS cuenta con **memoria ECC** para la detección de errores en los datos almacenados. Cada bloque de datos tiene su propio ECC (Código de corrección de errores). Cuando se escribe un bloque, se genera y almacena su ECC. Después de cada proceso de lectura del bloque, ZFS compara el ECC generado en la lectura con el ECC de la última escritura. Si no coinciden, es que se ha producido un error, el cual se corrige automáticamente. Los errores, que pueden llegar a ser incluso de un solo bit, suelen estar originados por fallos de hardware o alimentación o por interferencias eléctricas y magnéticas dentro del sistema [29].

4.9.2 Scrub

Más alla de los chequeos automáticos con el ECC, ZFS permite realizar manualmente scrubs. Un **scrub** es un escaneo que es capaz de hacer ZFS en los bloques de datos y también metadatos de un pool para comprobar que no haya errores. Se suelen programar de forma periódica. En estos scrubs tan solo se comprueban los bloques que están en uso.

Realizar un scrub puede reducir el rendimiento del sistema mientras se está ejecutando. No obstante, puede ser necesario renunciar a rendimiento con tal de asegurar la integridad de los datos [30].

A modo de ejemplo, podemos hacer un scrub de nuestro pool *poolex*. Si consultamos el estado del pool, este nos indicará si el scrub está en proceso o la última fecha en que se realizó:

```
root@truenas[~]# zpool scrub poolex
root@truenas[~]# zpool status poolex
  pool: poolex
 state: ONLINE
  scan: scrub repaired 0B in 00:00:00 with 0 errors on Sun Jul 24 18:51:58 2022
config:

        NAME        STATE     READ WRITE CKSUM
        poolex      ONLINE       0     0     0
          mirror-0  ONLINE       0     0     0
            da1     ONLINE       0     0     0
            da2     ONLINE       0     0     0
          mirror-1  ONLINE       0     0     0
            da3     ONLINE       0     0     0
            da4     ONLINE       0     0     0

errors: No known data errors
```

4.10 Reparación

En este apartado tratamos de analizar los métodos de reparación de ZFS cuando se daña algún dispositivo de almacenamiento.

4.10.1 Hot spare

El **hot spare** es un vdev que contiene uno o varios dispositivos de almacenamiento reservados para ser insertados en un data vdev en caso de que alguna unidad de este falle. De este modo, se podría evitar una posible pérdida de datos. El uso del dispositivo del hot spare vdev en principio es temporal hasta que la unidad dañada del data vdev se reemplace, en cuyo caso volvería a su situación inicial de reserva. No obstante, si esta unidad defectuosa se retira sin reemplazo, el originalmente dispositivo del spare vdev pasa a formar parte del data vdev permanentemente [20]. En la Figura 5 se puede ver cómo el pool tiene un hot spare vdev.

Podemos añadir a modo de ejemplo un hot spare single device vdev a nuestro pool:

```
root@truenas[~]# zpool add poolex spare da7
root@truenas[~]# zpool status poolex
  pool: poolex
 state: ONLINE
  scan: scrub repaired 0B in 00:00:00 with 0 errors on Sun Jul 24 18:51:58 2022
config:

        NAME        STATE     READ WRITE CKSUM
        poolex      ONLINE       0     0     0
          mirror-0  ONLINE       0     0     0
            da1     ONLINE       0     0     0
            da2     ONLINE       0     0     0
          mirror-1  ONLINE       0     0     0
            da3     ONLINE       0     0     0
            da4     ONLINE       0     0     0
        spares
          da7       AVAIL

errors: No known data errors
```

4.10.2 Resilvering

El **resilvering** es el proceso por el cual ZFS recupera datos cuando un dispositivo de almacenamiento ha sido sustituido por otro. Es decir, reescribe los mismos datos del antiguo dispositivo en el nuevo [31].

Un ejemplo claro de resilvering sucede cuando un dispositivo físico de un mirror vdev falla y se copian en el dispositivo del hot spare los datos que deberían estar almacenados en el dispositivo fallido. Si se reemplaza el device dañado, el dispositivo del hot spare vuelve a su función inicial de reserva y se produce otro resilvering en el nuevo dispositivo utilizado.

El resilvering y el scrub son operaciones excluyentes. Dicho de otro modo, no se puede realizar un resilvering a la vez que un scrub en un mismo pool. El resilvering es de mayor prioridad. Si se está llevando a cabo un scrub y hace falta un resilvering, el scrub permanece en espera hasta que finalice el resilvering. No obstante, ambos procesos son de baja prioridad [31].

4.11 Deduplicación

La **deduplicación** es una característica de ZFS que consiste en no almacenar más de una vez en un pool información equivalente. Se puede tener activada o no. Cada vez que se escriben datos en un dispositivo de almacenamiento, ZFS comprueba que no son equivalentes a los bloques ya existentes. En caso de encontrar un bloque idéntico, no se almacena físicamente, pero sí se escriben los metadatos necesarios. Los metadatos en este escenario particular son una tabla llamada **Tabla de Deduplicación (DDT)**. La tabla relaciona los bloques de datos con todos los archivos a los que pertenecen. Su conservación es esencial, por lo que es recomendable almacenarla de forma redundante. Existe la posibilidad de añadir un **dedup vdev** a un pool, que es un vdev dedicado exclusivamente a la deduplicación. Este vdev puede tener diseño de stripe o mirror (Figura 5) [20].

A modo de ejemplo, podemos considerar tres archivos de un pool que tienen un mismo bloque en común. Dicho bloque tan solo se escribe una vez en el dispositivo de almacenamiento correspondiente y la DDT vincula el bloque a los tres archivos. De este modo, se ahorra mucho espacio de almacenamiento.

En un mismo pool pueden coexistir datos deduplicados y datos no deduplicados. Cuando la deduplicación está activa, los datos se escriben usando la DDT. Si está desactivada, no se presta atención a posibles repeticiones y no se usa la DDT, pero esta permanece intacta pues es necesaria para gestionar los datos almacenados cuando sí lo estaba. El único método para conseguir que toda la información esté o bien deduplicada o bien no deduplicada es crear una copia en el mismo sistema de archivos o en otro con la configuración deseada.

El principal inconveniente de la deduplicación es el gran consumo de memoria RAM y CPU que realiza. La Tabla de Deduplicación requiere un uso de RAM y caché proporcinal a su tamaño. Se estima que por cada 1TB de datos almacenados, de los cuales 2/3 se pueden deduplicar, se necesitan entre 1 y 3 GB de RAM. Además, la CPU debería tener entre entre 4 y 6 núcleos [20]. Por estos motivos, en ocasiones la deduplicación no es conveniente o directamente posible.

A continuación se presenta un ejemplo de activación de la deduplicación en nuestro pool *poolex*. Asimismo, se puede comprobar la proporción de datos deduplicados en el parámetro dedup:

```
root@truenas[~]# zfs set dedup=on poolex
root@truenas[~]# zpool list poolex
NAME     SIZE  ALLOC   FREE  CKPOINT  EXPANDSZ   FRAG    CAP  DEDUP    HEALTH  A
LTROOT
poolex     9G   198K  9.00G        -         -     0%     0%  1.00x    ONLINE  -
```

4.12 Compresión nativa

La **compresión** es un método de ahorro de espacio en los dispositivos de almacenamiento y aumento de la velocidad. Se lleva a cabo en los bloques de datos cuando se escriben en los devices y el tipo de compresión se establece a nivel del dataset. ZFS cuenta con ella de forma nativa. No obstante, por defecto, está desactivada. ZFS soporta distintos algoritmos de compresión de datos. A continuación se analizan algunos de ellos [18]:

- **LZ4**: se trata de un algoritmo con gran velocidad de compresión y descompresión que ofrece el mejor rendimiento en la mayoría de los casos. Hoy en día es el que usa ZFS por defecto.
- **LZJB**: es un algoritmo que antes se usaba mucho, pero que ha quedado un poco obsoleto y, por eso, es recomendable utilizar otros.
- **GZIP**: es un algoritmo muy común en sistemas Unix. Suele utilizarse para compresión de textos, pero hay que tener cuidado porque es relativamente habitual que provoque ralentizaciones en la CPU.
- **ZLE**: siglas de Zero Level Encoding. Se trata de un algoritmo que solo comprime largas secuencias de ceros. El resto de datos los deja inalterados. Es útil para archivos que no se pueden comprimir como tal, como es el caso de JPEG y MP4 u otros formatos ya comprimidos, ya que simplemente reduce el espacio desperdiciado por ceros.

En líneas generales, se recomienda utilizar LZ4 en casi todos los escenarios. Un parámetro interesante de la compresión es el llamado **ratio de compresión** [32]. En ZFS se llama **compressratio**. Para comprender su funcionamiento, veamos un ejemplo. Supongamos que queremos escribir un archivo por bloques en un disco SSD. Tenemos establecidos los siguientes parámetros: tamaño de bloque recordsize=128KB y tamaño de sectores ashift=8KB. Para escribir un bloque necesitamos 128/8=16 sectores del SSD. Sin embargo, si llevamos a cabo una compresión, dígase LZ4, con un ratio de compresión compressratio=1.2, el tamaño del boque se reduce a $128/1.2 \sim 107$KB, por lo que tan solo necesitaremos $107/8 \sim 14$ sectores para escribirlo.

En nuestro ejemplo, prodríamos crear un sistema de archivos llamado *filesys2* con cualquiera de las compresiones mencionadas o incluso establecer o cambiar el tipo de compresión una vez ya se ha creado el dataset:

```
root@truenas[~]# zfs create -o compression=lz4 poolex/filesys2
root@truenas[~]# zfs set compression=lzjb poolex/filesys2
```

4.13 Cifrado nativo

ZFS puede **cifrar** de forma nativa datasets para que solo el propietario tenga acceso a sus datos. Es importante remarcar que este cifrado nativo no es a nivel de un pool, sino de un dataset. Cuando se cifra un dataset, se establece una clave de acceso a sus datos. De este modo, se mejora la seguridad e integridad de los datos frente a posibles ataques [20].

La gran ventaja del cifrado nativo es que se pueden llevar a cabo acciones como snapshots, replicaciones o scrubs sin necesidad de descifrar los datasets con la clave. Además, existe la posibilidad de hacer replicaciones cifradas de snapshots de datasets no

cifrados y viceversa, es decir, replicaciones no cifradas de snapshots de datasets cifrados [33].

Siguiendo con nuestro ejemplo, podríamos crear un dataset (sistema de archivos en este caso) llamado *fs3* que esté cifrado. Para ello, es necesario especificar correctamente el formato de la clave con keyformat (puede consultarse Ref[34]) y establecer una clave de acceso:

```
root@truenas[~]# zfs create -o encryption=on -o keyformat=passphrase poolex/fs3
Enter new passphrase:
Re-enter new passphrase:
```

4.14 Control de acceso

ZFS cuenta con un modelo de **listas de control de acceso (ACL)** (sección 3.7.2) puro, en el cual todos los archivos tienen una ACL. Concretamente, usa la implementación NFSv4 ACL. Los permisos que se imponen en las entradas de la ACL son normalmente al propietario, grupo de usuarios o todos los usuarios. Este tipo de ACLs se llaman **ACLs triviales**. Existe también la posibilidad de modificar los permisos de un archivo fuera de la ACL, pero las modificaciones realizadas se actualizan asimismo en la ACL. A continuación se listan algunas de las normas más significativas de una ACL en lo que al acceso a un archivo respecta [35]:

- Las entradas de la ACL se procesan en el orden en que están escritas de arriba a abajo.
- Una vez se ha concedido un permiso, no se puede denegar en una entrada que se procesa más tarde.
- Solo se concede acceso al solicitante si está autorizado en alguna de las entradas de la ACL.
- El propietario del archivo tiene siempre permiso para escribir en la ACL aunque en alguna de sus entradas se le niegue.

Vamos a visualizar un ejemplo de ACL. Para ello, primero necesitamos crear un archivo llamado *arch* dentro del sistema de archivos *filesys1*. Una vez creado, utilizando el comando getfacl [36] podemos ver su ACL:

```
root@truenas[~]# getfacl -v /poolex/filesys1/arch
# file: /poolex/filesys1/arch
# owner: root
# group: wheel
           owner@:read_data/write_data/append_data/read_attributes/write_attrib
utes/read_xattr/write_xattr/read_acl/write_acl/write_owner/synchronize::allow
           group@:read_data/read_attributes/read_xattr/read_acl/synchronize::al
low
        everyone@:read_data/read_attributes/read_xattr/read_acl/synchronize::al
low
```

4.15 Reservas y cuotas de espacio

En sistemas de archivos tradicionales, tan solo se podía almacenar un sistema de archivos por partición del dispositivo de almacenamiento y dicho sistema de archivos podía llegar a ocupar todo el espacio disponible en la partición. Por el contrario, en ZFS creábamos un pool en el que podían convivir más de un sistema de archivos. Estos sistemas de archivos podían aumentar su tamaño siempre y cuando hubiera espacio libre en el pool. En este contexto, es donde juegan un papel importante las reservas y cuotas de espacio.

Una **reserva** de espacio en ZFS permite guardar un espacio concreto de un pool para que solo pueda ser utilizado por un sistema de archivos y sus descendientes. Esto quiere decir que se incluyen también los snapshots y clones. De este modo, se evita que otros sistemas de archivos acaparen todo el almacenamiento libre disponible en el pool.

Una **cuota** de espacio es un límite de espacio máximo que se le otorga a un sistema de archivos y sus descendientes dentro de un pool. Es una forma de asegurar que un sistema de archivos no consuma demasiado espacio dentro del pool.

Existen tambíen las denominadas **refreservas** y **refquotas**, que son similares a las reservas y quotas. La única diferencia es que guardan un espacio y límitan el mismo tan solo al sistema de archivos como tal, sin incluir descendientes [21].

En nuestro pool *poolex*, podríamos, por ejemplo, establecer una reserva de 1GB al dataset *filesys2* y una quota de 2GB a *fs3*:

```
root@truenas[~]# zfs set reservation=1gb poolex/filesys2
root@truenas[~]# zfs get reservation poolex/filesys2
NAME              PROPERTY     VALUE   SOURCE
poolex/filesys2  reservation  1G      local
root@truenas[~]# zfs set quota=2gb poolex/fs3
root@truenas[~]# zfs get quota poolex/fs3
NAME         PROPERTY  VALUE  SOURCE
poolex/fs3  quota     2G     local
```

Las refreservas y refquotas se realizan de forma análoga.

4.16 Mejoras de rendimiento

4.16.1 L2ARC

Para intentar agilizar los procesos de lectura, ZFS utiliza **ARC** (Adaptative Replacement Cache). ARC es un caché que hace uso de memoria RAM para almacenar los datos más frecuentes y recientemente empleados del pool. No obstante, puede que en algunos casos la memoria RAM destinada a caché se llene y ARC no sea capaz de atender todas las demandas. En esta situación, puede ser interesante añadir un caché vdev a un pool con el que se trabaje exhaustivamente. Dicho chaché vdev también se conoce como **L2ARC** (ARC de nivel 2), ya que se hace cargo de las peticiones de lectura que ARC no es capaz de llevar a cabo por colapso [20].

4.16.2 Log vdev

ZFS tiene una característica llamada **ZIL** (ZFS intent log) que juega un papel importante en los procesos de escritura síncrona. El ZIL es una parte de un dispositivo de almacenamiento reservada a ejercer de almacenamiento intermedio para los datos que se van a escribir de forma síncrona en el dispositivo. Su principal función es velar por la integridad de los datos. Si se produce un fallo en el sistema y este se apaga en medio de un proceso de

escritura, los datos en memoria principal se pierden. Sin embargo, en el ZIL permanecen intactos ya que se trata de memoria no volátil. De este modo, se podrían recuperar los datos gracias al ZIL. Además, también agiliza los procesos de escritura síncrona.

Si se desea aumentar aún más la velocidad de escritura síncrona, se podría dedicar un vdev en un pool exclusivamente a ejercer de ZIL. Este se conoce como **log vdev**. Se trata también de una forma de evitar destinar parte del espacio de los dispositivos de almacenamiento al ZIL. Se recomienda que este log vdev se cree con SSDs y que tenga diseño de mirror [20].

4.16.3 Metadata vdev

El **metadata vdev**, también denominado **special vdev**, es un tipo de vdev destinado a almacenar metadatos como direcciones de ficheros o tablas de asignación. En ocasiones también se puede emplear para guardar pequeños bloques de datos y así aumentar la velocidad de acceso. Se recomienda acompañar a un metadata vdev con al menos un spare vdev, ya que contiene información importante para el funcionamiento del pool. Es conveniente que el diseño del metadata vdev sea tipo mirror [20]. A modo de ejemplo, el pool de la Figura 5 contiene un metadata vdev con diseño de mirror.

4.16.4 Ejemplo

A modo de ejemplo, podemos añadir a nuestro pool *poolex* un caché single device vdev, un log mirror vdev y un special mirror vdev. Si consultamos el estado del pool, se observan todos los vdevs que lo componen:

```
root@truenas[~]# zpool add poolex cache da8
root@truenas[~]# zpool add poolex log mirror da9 da10
root@truenas[~]# zpool add poolex special mirror da11 da12
root@truenas[~]# zpool status poolex
```

```
  pool: poolex
 state: ONLINE
config:

        NAME        STATE     READ WRITE CKSUM
        poolex      ONLINE       0     0     0
          mirror-0  ONLINE       0     0     0
            da1     ONLINE       0     0     0
            da2     ONLINE       0     0     0
          mirror-1  ONLINE       0     0     0
            da3     ONLINE       0     0     0
            da4     ONLINE       0     0     0
        special
          mirror-3  ONLINE       0     0     0
            da11    ONLINE       0     0     0
            da12    ONLINE       0     0     0
        logs
          mirror-2  ONLINE       0     0     0
            da9     ONLINE       0     0     0
            da10    ONLINE       0     0     0
        cache
          da8       ONLINE       0     0     0
        spares
          da7       AVAIL
```

4.17 Memoria RAM

ZFS requiere mucha **memoria RAM** para su correcto funcionamiento en comparación con otros sistemas de archivos más convencionales. No es que sea un defecto, sino que está diseñado para consumir toda la RAM que pueda. Se estima que para el correcto funcionamiento de ZFS en casos de operaciones sencillas hacen falta por lo menos 8GB de RAM [20]. A medida que aumenta la cantidad de datos almacenados y la complejidad de las operaciones, la demanda de memoria RAM es mayor. Este es el principal inconveniente de ZFS para ser utilizado a nivel usuario. Para un ordenador de hogar, el consumo de RAM de ZFS es demasiado elevado y supondría una pérdida de rendimiento del mismo en otros aspectos. Es por eso que la gran mayoría de ordenadores de usuario usan otros sistemas de archivo. ZFS está más orientado a servidores donde el consumo de RAM no es un problema y sí se valora más la integridad de datos que te proporciona este sistema de archivos.

5 Comparación de sistemas de archivos

Se trata ahora de comparar algunas de las principales características de ZFS con otros sistemas operativos. En la Tabla 1 se puede visualizar dicha comparativa.

	ZFS	**exFAT**	**NTFS**	**APFS**	**UFS**	**ext4**	**BTRFS**
Administrador de volúmenes integrado	✓			✓			✓
Redundancia de datos	✓						✓
Tolerancia a fallos de devices	✓						✓
Copy-on-write	✓			✓			✓
Snapshots	✓			✓	✓		✓
Replicación	✓						✓
ECC de datos	✓						✓
Scrub	✓			✓			✓
Deduplicación	✓		✓	✓			✓
Compresión nativa	✓		✓	✓			✓
Cifrado nativo	✓		✓	✓		✓	
ACL	✓		✓	✓	✓	✓	✓
Consumo RAM	Alto	Bajo	Medio	Medio	Medio	Medio	Alto

Table 1: Comparación entre sistemas de archivos.

Como podemos ver, tan solo ZFS, APFS y BTRFS cuentan con administrador del volúmenes integrado. Esto quiere decir que en el resto de los sistemas de archivos solo puede haber un sistema de archivos por dispositivo de almacenamiento físico o partición del mismo, a no ser que utilizen un software específico para la administración de volúmenes.

En lo que a la integridad de los datos respecta, ZFS y BTRFS son claramente ganadores. En ambos casos se puede lograr redundancia de datos gracias a la administración de volúmenes y también hay tolerancia al fallo de un número concreto de devices. Como se ha visto anteriormente, en ZFS esto se lograba por medio de los mirror, RAIDz, RAIDz2 y RAIDz3 vdevs. No obstante, esto supone también la necesidad de disponer de un gran

número de dispositivos de almacenamiento.

El copy-on-write solo es propio de ZFS, APFS y BTRFS [37][13]. En los demás sistemas de archivos, al modificar los datos en un bloque, se llevan a cabo los cambios en el mismo bloque, no en una copia. Esto puede llegar a ser un problema porque en caso de fallo del sistema en medio de un proceso de manipulación del bloque, se podrían perder datos. En cambio, con el copy-on-write, hasta que no se termina de modificar el bloque copia y se desvincula el bloque antiguo para vincular el nuevo, no se considera la operación como terminada, asegurando así la integridad de los datos.

Se pueden realizar snapshots en ZFS, APFS y BTRFS [37][13] aprovechando el copy-on-write, pero tambíen es posible hacerlos en UFS a pesar de no tener copy-on-write [38]. Con estos snapshots luego se pueden hacer rollbacks y regresar a instantes anteriores. Esto puede ser interesante si se produce algún fallo y se desea volver al estado previo al mismo.

Las replicación como tal solo la tienen ZFS y BTRFS. Se trata de una herramienta fundamental para mantener un registro de los datos almacenados y tenerlos guardados también en otro lugar para poder recuperarlos en caso de fallo.

La corrección autómatica de errores de datos haciendo uso de memoria ECC es solo propia de ZFS y BTRFS [39]. Estos dos sistemas de archivos, además de APFS, permiten también realizar scrubs cada cierto para comprobar que todo está en orden [10][13]. Estas dos características evidentemente suponen una gran ventaja para evitar la corrupción de datos.

Por otro lado, la deduplicación es propia de ZFS, BTRFS, NTFS y APFS [39]. Se trata de una forma eficiente de optimizar el espacio de almacenamiento utilizado, pero supone un incremento en el consumo de memoria RAM. Estos mismos sistemas de archivos cuentan también con métodos de compresión de los bloques de datos. La compresión es una forma más de aprovechar al máximo el espacio de los dispositivos de almacenamiento.

Como podemos comprobar hasta el momento, ZFS y BTRFS tienen unas características bastante similares. No obstante, BTRFS no cuenta con el cifrado nativo de sistemas de archivos que ZFS sí tiene, lo cual supone una ventaja en cuanto a seguridad. Además, no solo ZFS posee dicho cifrado, sino que NTFS, APFS y ext4 también [39].

Las listas de control de acceso (ACL) son más habituales, puesto que todos los sistemas de archivos analizados cuentan con ellas salvo exFAT []39.

Para terminar, ZFS y BTRFS realizan un consumo de memoria RAM muy elevado en comparación con el resto de sistemas de archivos. Esto puede llegar a ser un problema porque puede que no se cuente con suficiente RAM para soportar el funcionamiento de uno de estos sistemas al mismo tiempo que se llevan a cabo otro tipo de operaciones.

En líneas generales, podemos decir que ZFS, al igual que BTRFS que es bastante similar pero para sistemas operativos Linux, presenta unas características muy ventajosas en términos de salvaguardar la integridad de datos y evitar la corrupción de los mismos. No obstante, ambos consumen una cantidad de memoria RAM demasiado elevada para poder ser utilizados, por ejemplo, en ordenadores a nivel usuario. Es por eso que ZFS y BTRFS son sistemas de archivos destinados más al almacenamiento masivo en servidores, donde el consumo de RAM y, probablemente, la necesidad de un gran número de dispositivos de almacenamiento para lograr redundancia de datos, no son un problema.

6 Conclusiones

En primer lugar, hemos definido lo que es un archivo. Hemos visto que un archivo podía ser un conjunto de registros o bien simplemente una secuencia de bytes. Una vez conocido el concepto de archivo, hemos procedido a analizar cómo se gestionan. Para ello, hemos definido el término sistema de archivos. Dicho término incluía dos vertientes: la relacionada con la gestión de archivos y la referente a los servicios ofrecidos a procesos para realizar operaciones con archivos y directorios.

Tras analizar el cometido, el funcionamiento y la arquitectura de un sistema de archivos, nos hemos adentrado en el estudio de la vertiente de la gestión de archivos. Primero hemos analizado la organización por directorios típica con estructura de árbol, así como las principales formas de control de acceso a archivos. Posteriormente, hemos profundizado en las distintas estructuras de archivo y métodos de acceso, considerando los archivos como un conjunto de registros. Los archivos se almacenan en los dispositivos físicos por bloques, por lo que también hemos tenido que estudiar cómo se pasa de registros a bloques. Asimismo, hemos analizado la gestión del espacio de almacenamiento, lo que incluye tanto la asignación de espacio a archivos como la gestión de espacio libre.

Por otro lado, hemos considerado la vertiente del sistema de archivos correspondiente a los servicios. Hemos visto las principales operaciones con archivos y directorios que un sistema de archivos proporciona a un proceso.

Para finalizar con la sección de sistemas de archivos, hemos visto el concepto de administrador de volúmenes y hemos mencionado algunos de los sistemas de archivos más utilizados por los principales sistemas operativos.

Una vez conocidas las características generales de los sistemas de archivos, nos hemos centrado en el estudio de uno muy concreto, ZFS. Hemos visto que ZFS es un sistema de archivos, pero también es un administrador de volúmenes. Primero hemos analizado su estructura basada en pools y vdevs. A continuación, hemos profundizado en cómo se almacenan y organizan los datos, viendo qué son los datasets y los bloques y sectores en que se guardan los datos.

Posteriormente, hemos considerado las principales características de ZFS, como son el copy-on-write o los snapshots. También hemos estudiado la replicación, que resulta de gran importancia para asegurar la integridad de los datos, así como un método para exportar un pool de un sistema a otro. El siguiente paso ha sido analizar las soluciones que tiene ZFS ante problemas. Hemos visto la corrección de errores automática con memoria ECC y los scrubs y también qué son el hot spare y el resilvering. Asimismo, hemos considerado otras características como la deduplicación, la compresión nativa o el cifrado nativo, y hemos propuesto posibles mejoras de rendimiento. Además, se ha explicado el alto consumo de memoria RAM que ZFS realiza.

Finalmente, para poder poner en valor las propiedades de ZFS, las hemos comparado con las de los sistemas de archivos más habituales. Hemos concluido que ZFS es un sistema de archivos muy potente en términos de integridad y seguridad de datos. No obstante, el consumo de RAM que realiza es demasiado elevado para ser utilizado a nivel usuario, por lo que ZFS está más destinado a su uso en servidores.

Referencias

[1] William Stallings, *Sistemas Operativos*, Prentice Hall (2001)

[2] Yair Amir, *Operating Systems, 600.418, The File System*, Department of Computer Science The Johns Hopkins University, URL: https://www.cs.jhu.edu/~yairamir/cs418/os7/sld001.htm

[3] Andrew S. Tanenbaum, Albert S. Woodhull, *Operating Systems: Design and Implementation*, Prentice Hall (2006)

[4] *Sistemas de archivos: qué son y cuáles son los más importantes*, Ionos, URL: https://www.ionos.es/digitalguide/servidores/know-how/sistemas-de-archivos/

[5] Daniel Grosshans, *File Systems: Design and Implementation*, Englewood Cliffs, NJ: Prentice Hall (1986)

[6] *Administración de volumenes*, Institut Puig Castellar, URL: https://elpuig.xeill.net/Members/vcarceler/c1/didactica/apuntes/ud4/na3

[7] *Tabla de asignación de archivos*, Wikipedia, URL: https://es.wikipedia.org/wiki/Tabla_de_asignaci%C3%B3n_de_archivos

[8] Yúbal Fernández, *FAT32, NTFS o exFAT: qué sistema de archivos elegir al formatear tu disco duro o USB*, Xataka, URL: https://www.xataka.com/basics/sistemas-de-archivo-como-saber-cual-elegir-al-formatear-tu-disco-duro-o-usb

[9] *NTFS: qué es y qué ventajas e inconvenientes tiene* , Axarnet, URL: https://axarnet.es/blog/ntfs

[10] Adam H. Leventhal, *A ZFS developer's analysis of the good and bad in Apple's new APFS file system*, Arstechnica, https://arstechnica.com/gadgets/2021/06/a-quick-start-guide-to-openzfs-native-encryption/

[11] *UFS File System*, Oracle Corporation, URL: https://docs.oracle.com/cd/E19253-01/817-5093/fsoverview-12343/index.html

[12] Sergio De Luz, *¿Qué sistema de archivos elegir para mi servidor NAS basado en Linux?*, Redes Zone, URL: https://www.redeszone.net/tutoriales/servidores/sistemas-archivos-ext4-btfs-zfs-elegir/

[13] BTRFS Wiki, URL: https://btrfs.wiki.kernel.org/index.php/Main_Page

[14] *The Z File System (ZFS)*, FreeBSD Handbook, URL: https://docs.freebsd.org/en/books/handbook/zfs/

[15] *ZFS History*, Wikipedia, URL: https://openzfs.org/wiki/History

[16] Sergio De Luz, *Cómo instalar ZFS, el mejor sistema de archivos para servidores*, Redes Zone, URL: https://www.redeszone.net/tutoriales/servidores/sistema-archivos-zfs-servidores/

[17] Adam H. Leventhal, *ZFS: The other new Apple file system that almost was until it wasn't*, Arstechnica, https://arstechnica.com/gadgets/2016/06/zfs-the-other-new-apple-file-system-that-almost-was-until-it-wasnt/

[18] Jim Salter, *ZFS 101-Understanding ZFS storage and performance*, Arstechnica, https://arstechnica.com/information-technology/2020/05/zfs-101-understanding-zfs-storage-and-performance/

[19] ReclaiMe Team, *RAIDZ types reference*, RAIDz calculator, URL: http://www.raidz-calculator.com/raidz-types-reference.aspx

[20] TrueNAS CORE and Enterprise Documentation, URL: https://www.truenas.com/docs/core/coretutorials/

[21] Damian Wojsław, *Introducing ZFS on Linux*, Apress (2017)

[22] *ZFS Terminology*, Oracle Solaris ZFS Administration Guide, URL: https://docs.oracle.com/cd/E18752_01/html/819-5461/ftyue.html

[23] *Overview of ZFS Clones*, Oracle Solaris ZFS Administration Guide, URL: https://docs.oracle.com/cd/E19253-01/819-5461/gbcxz/index.html

[24] *Replacing a ZFS File System With a ZFS Clone*, Oracle Solaris ZFS Administration Guide, URL: https://docs.oracle.com/cd/E19120-01/open.solaris/817-2271/gcvfl/index.html

[25] *Rolling Back a ZFS Snapshot*, Oracle Solaris ZFS Administration Guide, URL: https://docs.oracle.com/cd/E19253-01/819-5461/gbcxk/index.html

[26] *Sending and Receiving ZFS Data*, Oracle Solaris ZFS Administration Guide, URL: https://docs.oracle.com/cd/E18752_01/html/819-5461/gbchx.html

[27] *Replication Overview*, Oracle ZFS Storage Appliance Administration Guide, URL: https://docs.oracle.com/cd/E37831_01/html/E52872/gocuj.html

[28] *Migrating ZFS Storage Pools*, Oracle Solaris ZFS Administration Guide, URL: https://docs.oracle.com/cd/E18752_01/html/819-5461/gbchy.html

[29] *¿Qué es la memoria ECC?*, Crucial by Micron Technology, Inc., URL: https://www.crucial.es/articles/pc-builders/what-is-ecc-memory

[30] *Checking ZFS File System Integrity*, Oracle Solaris ZFS Administration Guide, URL: https://docs.oracle.com/cd/E18752_01/html/819-5461/gbbwa.html

[31] *ZFS Data Scrubbing and Resilvering*, Oracle Solaris ZFS Administration Guide, URL: https://docs.oracle.com/cd/E19253-01/819-5461/gbbya/index.html

[32] *OpenZFS: Understanding Transparent Compression*, Klara Inc., URL: https://klarasystems.com/articles/openzfs1-understanding-transparent-compression/

[33] *OpenZFS Native Encryption*, Klara Inc., URL: https://klarasystems.com/articles/openzfs-native-encryption/

[34] *Encrypting ZFS File Systems*, Oracle Solaris 11.1 Administration: ZFS File Systems, URL: https://docs.oracle.com/cd/E26502_01/html/E29007/gkkih.html,

[35] *Setting ACLs on ZFS Files*, Oracle Solaris ZFS Administration Guide, URL: https://docs.oracle.com/cd/E23823_01/html/819-5461/gbace.html

[36] *FreeBSD Manual Pages: getfacl*, URL: https://www.freebsd.org/cgi/man.cgi?query=getfacl&sektion=1

[37] *Apple File System Reference*, Apple Inc., URL: https://developer.apple.com/support/downloads/Apple-File-System-Reference.pdf

[38] *UFS Snapshots Overview*, System Administration Guide: Devices and File Systems, URL: https://docs.oracle.com/cd/E23823_01/html/817-5093/bkupsnapshot-11.html

[39] *Comparison of file systems*, Wikipedia, URL: https://en.wikipedia.org/wiki/Comparison_of_file_systems