Estrategias de optimización y análisis de performance en sistemas de almacenamiento distribuido

Javier Gonzalo Charne

Directores:

Hugo Dionisio Ramón

Francisco Javier Díaz

Tesis presentada para obtener el grado de Magister en Redes de Datos

Facultad de Informática - Universidad Nacional de La Plata

Abril 2021

Revisión noviembre 2021

A la UNNOBA, donde tuve y tengo el privilegio de trabajar.
A Hugo, quien hace años capitanea el barco y también toma su turno en los remos.
A Adrián y Diego, fieles y generosos compañeros de trinchera.
A Raquel, cuya amistad atesoro desde hace tantísimos años.
Y, fundamentalmente, a Elisa, mi mujer, y a Pedro y Francisca. Su presencia en mi vida modeló mi presente y llena de proyectos mi futuro.

ÍNDICE TEMÁTICO

1. INTRODUCCIÓN

La complejidad de los sistemas de almacenamiento es creciente debido a los esfuerzos por responder a requerimientos cada vez más estrictos: mayor cantidad y simultaneidad de clientes conectados, un número creciente de datos accedidos de manera concurrente, usuarios distribuidos geográficamente, tiempos de respuesta más acotados, el *throughput* que se amplía de forma exponencial, y todo esto con capacidad para recuperarse ante fallas que deben suponerse la regla más que la excepción.

A las unidades, archivos y carpetas que fueron durante mucho tiempo las interfaces con las que el usuario y las aplicaciones interactuaban con el sistema de almacenamiento se sumó la necesidad de acceder tanto a dispositivos de bloques como a objetos individuales, y todo esto en un sistema integrado y coherente de gestión, que posibilitara tanto la administración del almacenamiento como su monitoreo.

Respondiendo a estas necesidades, surgen soluciones de Almacenamiento Definido por Software (Software Defined Storage), en las que distintos dispositivos conectados a través de una red de datos forman un cluster que ofrece un conjunto de interfaces a las aplicaciones y clientes, y proveen un complejo sistema de gestión, mantenimiento y monitoreo de los distintos componentes.

Estas soluciones están orientadas a integrarse como un engranaje más en la gran maquinaria de infraestructura de servicios virtualizados (cómputo, conectividad, almacenamiento, réplicas, monitoreo, etc…) que poco a poco van poblando nuestros datacenters.

1.1. Situación y Motivación

El almacenamiento de los datos de la Universidad Nacional del Noroeste de la provincia de Buenos Aires (UNNOBA), donde se realiza el presente trabajo, y su gestión están diseminados por varios sistemas, cada uno de los cuales aporta una interfaz distinta, se adapta a una carga de trabajo diferente, y tienen anchos de banda y latencias dispares.

En el marco de la estrategia de gobierno electrónico de la Universidad, se guardan datos de antiguos alumnos, de docentes, de investigadores y de funcionarios; documentos de proyectos de extensión; registros de actividades realizadas a lo largo de la vida de la universidad: resoluciones de organismos de gobierno, expedientes de las distintas áreas de gestión, audios de entrevistas realizadas por la radio, producciones en videos, fotografías tomadas… documentos que -puestos a armar una taxonomía- podríamos identificar como históricos, con pocos accesos, pero que deben conservarse y son siempre crecientes.

Y también hay archivos accedidos diariamente a través de múltiples aplicaciones: web, bases de datos, nubes privadas, carpetas compartidas, correos, ftp, sistemas propios, microservicios… Estos archivos tienen poco tamaño y su número asciende a varios

millones. Datos que deben accederse con latencia mínima y preservarse frente a fallas de almacenamiento y errores humanos.

Hasta que la OMS declara pandemia la expansión de COVID-19 en el mundo, la mayor parte de la interacción directa con los sistemas de almacenamiento se daba dentro de la red de la universidad, con velocidades permitidas por la red LAN, y fallas que quedaban en el ámbito de gestión del área informática propia. Los accesos vía VPN eran pocos en relación a los usuarios internos, y las aplicaciones diseñadas para uso propio no contemplaban ni la velocidad, ni la latencia que las redes WAN introducen en la transferencia de datos.

Como todo el mundo, tuvimos que adaptarnos de una semana para otra, habilitando cientos de accesos vía VPN para que muchos docentes, investigadores y personal administrativo de la universidad pudieran continuar con sus tareas. La dispersión geográfica de los usuarios se incrementó considerablemente, y se invirtió el flujo de datos sobre los enlaces WAN, que pasó a ser mayoritariamente *upstream*.

El impacto que tuvo esta realidad sobre los sistemas de plataforma de educación a distancia fue importante. El porcentaje de materias que hacían uso activo del mismo pasó de una pequeña fracción a casi la totalidad. Y quienes antes usaban dichos entornos como meros repositorios de archivos, comenzaron a utilizar la amplia gama de servicios y actividades de interacción que proveen. La imagen titulada “Servicios Sustantivos” (Figura 1), que representa la cantidad de usuarios que interactúan con esos servicios, se incluyó en el informe del Prosecretario de TIC de la UNNOBA en su informe de gestión 2019-2020.

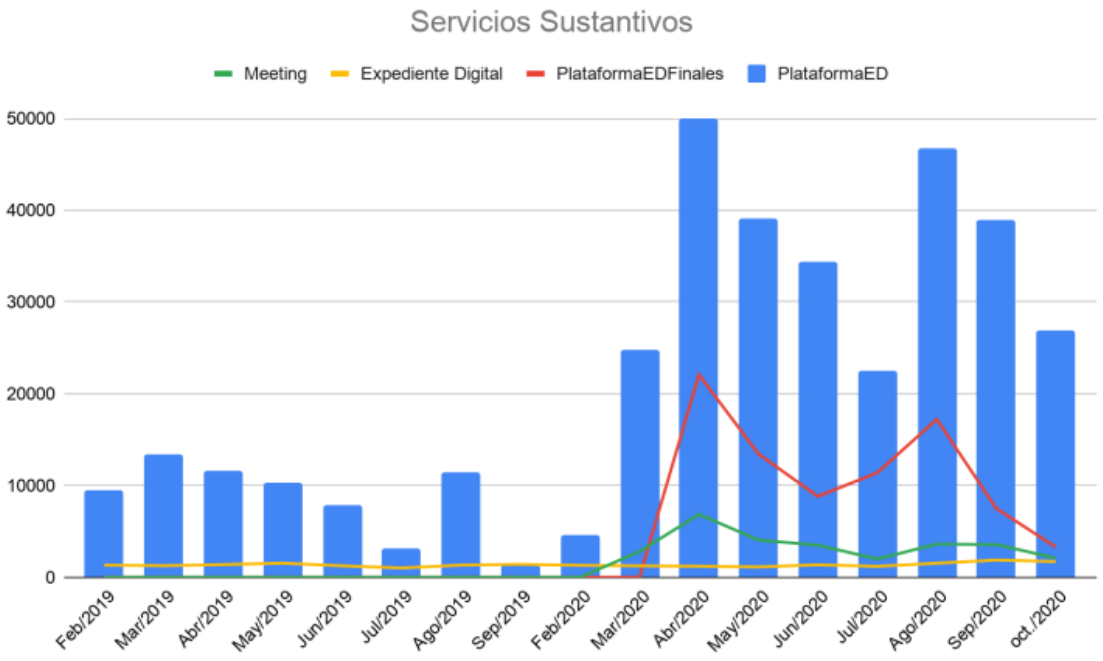


Figura 1 - Número de accesos registrados a los Servicios Sustantivos

El sistema de almacenamiento vio reflejado este incremento, y hubo que adecuar (mover, copiar, fragmentar, replicar…) los volúmenes de almacenamiento de uno a otro

sistema, en busca de mayor capacidad. Y con esto, re-adaptar los servicios de backups de la nueva información crítica que se estaba generando.

Además de los datos “de los usuarios”, los servidores de la universidad también tienen sus requisitos propios: volúmenes para almacenar las imágenes de las máquinas virtuales y los contenedores, y proveer al área de sistemas de los entornos necesarios de desarrollo, testing y producción. Clonar máquinas virtuales, pasar a testing una máquina que está en producción para que le apliquen los parches necesarios, hacer *snapshots* de máquinas previo a una actualización, hacer *rollback* o “pisar” una imagen con otra, son prácticas habituales en el día a día del área.

Los tradicionales dispositivos de almacenamiento, como discos individuales, unidades NAS (*Network Attached Storage*) y SAN (*Storage Area Network*) comenzaron a mostrar algunas desventajas:

- Escalabilidad, en términos de capacidad de almacenamiento, cantidad de usuarios concurrentes que pueden acceder al dispositivo con parámetros de latencia y throughput acotados.
- Funcionamiento sobre hardware específico del fabricante e incompatibilidad con el resto (chasis, memoria, discos, placas de red, etc.) El mantenimiento, la reparación y la actualización del equipo es costosa en términos económicos y limitada en tiempo, debido a políticas de end-of-life del mismo fabricante.
- Actualización de firmware (provisto exclusivamente por el fabricante) que implica poner offline el equipo y desconectar todas las unidades compartidas.
- Soporte técnico, base de conocimientos, manual de uso y troubleshooting son potestad exclusiva del fabricante, y se transforman en otro servicio más que debe comprarse para poder almacenar en estos dispositivos.
- Estos equipos representan un SPoF (*Single Point of Failure*), y la necesidad de replicación de la información para asegurar una continuidad de negocio ante incidentes (failover o backup de información histórica) obligan a invertir en más equipamiento, que no siempre trabajan en conjunto.

Se analizaron entonces diferentes soluciones de almacenamiento distribuido, y llegamos a Ceph, que en la documentación oficial enuncia:

- Una tecnología de almacenamiento distribuido basada en software libre que es neutral respecto de proveedores y fabricantes
- Puede escalar horizontalmente, sobre hardware genérico
- Es tolerante a fallas a nivel de discos, de nodos y de red de datos.
- Ofrece servicios de almacenamiento de objetos, de archivos y de dispositivos de bloques (*RBD, Rados Block Device*), que están implementados sobre un sistema de almacenamiento de objetos que es escalable, confiable y performante (*RADOS: Reliable Autonomic Distributed Object Store*)

1.2. Lograr el mejor desempeño

Es claro que la performance del cluster, medida en latencia de las operaciones de lectura y escritura y throughput total, depende del hardware subyacente, de la red de datos que lo conecta y de la configuración tanto de los nodos de almacenamiento, monitoreo y metadatos, como de los clientes que se conectan al cluster.

No obstante, una pregunta aparece en varios trabajos sobre estos sistemas: *¿Cómo verifico que mi cluster está corriendo en su máxima performance? (*Jiangang, 2013)

Algunos equipos de investigación en Big Data apuntan a verificar la escalabilidad de Ceph frente a grandes volúmenes de información (Yang, 2017). Otros buscan mejorar los niveles de caché (Shankar, 2017) para minimizar las latencias producidas por mecanismos de spinning (Lee, 2017) de discos rígidos tradicionales, y otros directamente apuntan a la implementación sobre discos de estado sólido exclusivamente (Ra, 2018), con los costos asociados a este tipo de dispositivos.

También están quienes analizan qué cambios realizar sobre las configuraciones del cluster para optimizar su rendimiento en escenarios de computación de alto desempeño, como en CERN (*Conseil Européen pour la Recherche Nucléaire, Consejo Europeo para la Investigación Nuclear*), donde la escala de almacenamiento son de cientos de Petabytes (Van Der Ster, 2014).

Otros, centran su atención en los clientes y en cómo cachear datos mejora la performance de lectura, aunque alertan sobre cómo esto afecta la consistencia de los datos en un ambiente distribuido. (Chamarty, 2020)

Así, dado un escenario concreto de uso, con el hardware *concretamente disponible* y una instalación base, motiva el presente trabajo la inquietud por definir estrategias para identificar parámetros de ajuste en cada nivel, tendientes a mejorar la performance integral del cluster de almacenamiento.

1.3. Objetivos

Los objetivos del presente trabajo son:

1. Analizar las diferentes soluciones de almacenamiento distribuido con desarrollo activo

2. Identificar el sistema de almacenamiento distribuido que más se adecúa a la infraestructura disponible, a las interfaces requeridas, y a la carga de trabajo y a los patrones de acceso utilizados en el ámbito de investigación e implementación de la presente tesis.

3. Establecer metodología de benchmarking y definir métricas a evaluar.

4. Analizar e Identificar qué configuraciones resultan óptimas
    a. en los dispositivos de almacenamiento por bloques (HDD y SSD)
    b. sobre la red de datos: a nivel físico y protocolos involucrados
    c. en el backend de almacenamiento de Ceph: OSDs, RADOS, Pools y dispositivos de bloque RBD, excluyendo objetos (RGW) y filesystem (CephFS)
    d. en la configuración de parámetros y módulos del kernel del servidor y de los clientes

5. Establecer valores de rendimiento base que puedan utilizarse a modo de referencia para realizar monitoreos continuos y evaluar los ajustes futuros sobre parámetros que afectan la performance.

2. MARCO TEÓRICO y ESTADO del ARTE

Los requerimientos sobre la tecnología son siempre crecientes: mayor capacidad, mayor velocidad, más cantidad de usuarios, mejor tiempo de respuesta y sin fallas. Se pide que un sistema mantenga su calidad de servicio aún cuando la cantidad de accesos concurrentes escalan día a día. Así las redes de datos ven saturada su capacidad en poco tiempo, el número y potencia de los procesadores nunca es suficiente, y el almacenamiento tampoco es ajeno a esta situación. Las arquitecturas tradicionales centralizadas no logran cumplir con esos requisitos de performance y las soluciones distribuidas parecen ser la respuesta a esos requerimientos.

Colouris, Tanenbaum, Silberschatz y Stallings, autores de la bibliografía tradicional sobre Sistemas Distribuidos, concuerdan en afirmar que estos sistemas deben poder escalar a nivel de tamaño, a nivel geográfico y a nivel administrativo (Coulouris, 1994) (Tanenbaum, 2016) (Silberschatz, 1990) (Stallings, 2018).

Aplicados estos conceptos al almacenamiento, se debe contemplar que la cantidad de clientes (usuarios y servidores) realizando operaciones de E/S aumenta, como así también el volumen de información y con esto, el throughput requerido para que el servicio no se degrade. Hace tiempo la magnitud de Terabytes/TibiBytes hacía presuponer que se había provisionado capacidad suficiente en la unidad de almacenamiento... Hoy sabemos que 8TB se llenan en pocos días.

Por supuesto todo depende de las características de la carga de trabajo sobre el sistema. La generación de video tiene patrones de acceso secuenciales: grandes archivos de muchos gigabytes, que se escriben de manera conjunta y que se leen de la misma manera, normalmente desde el inicio. En cambio, el almacenamiento de archivos de correo, o de un proxy web, es un ejemplo totalmente opuesto: millones de archivos de poco tamaño, que se acceden en un patrón completamente aleatorio. (Songbin, 2013)

Las soluciones tradicionales para proveer de almacenamiento a servidores consistía en montar volúmenes remotos en el directorio local a través de algún protocolo como NFS o CIFS (si se trataba de un NAS), o bien conectar a nivel de dispositivo, vía protocolos Fibre Channel o iSCSI (para acceder a SAN). Pero la centralización de estos modelos es un problema para la escalabilidad y la performance. Los discos o volúmenes están en un servidor que los exporta, y este servidor se convierte en un SPoF (Single Point of Failure. Punto único de falla), que puede provocar la caída completa del sistema. Además, están limitados por la capacidad de las interfaces de red, o la cantidad, tipo y velocidad de los discos. Estos dispositivos (NAS o SAN) permiten configurar múltiples discos en RAID, de manera de paralelizar las lecturas/escrituras y lograr mejor performance en momentos de alta concurrencia. Y también poseen capacidades de ampliación agregando más “cajas con discos” y conectándolos en cascada. No obstante, sigue habiendo un cuello de botella en las interfaces de red que conectan este equipo con los chasis tipo Blade o directamente a la red. ¡Y el equipo en sí mismo continúa representando un punto de falla crítico!

La actualización de firmware de estos dispositivos constituye una tarea también crítica que debe hacerse en modo off-line, desconectando a todos los usuarios y obligando

a detener todas las tareas de almacenamiento. El momento para realizar esta tarea y su costo, no es trivial. Como tampoco lo son los costos asociados a la renovación o reemplazo de partes de este hardware que suelen ser exclusivos del fabricante. O el costo del servicio técnico especializado cuando el equipo falla.

La poca rotación de hardware que impone nuestra realidad económica nos obliga a estirar la vida útil de estos equipos más allá de lo que el fabricante propone. ¿Cuántos de nuestros equipos los hemos comprado casi llegando a su etapa de “End-of-Sale” y a los pocos años, cuando necesitamos reemplazar algún componente, vemos que ya pasó su fecha de “End-Of-Life”?

Así, fue creciendo la inquietud por el desarrollo de sistemas de almacenamiento distribuido que no sea privativo de ninguna marca, que virtualice las unidades y volúmenes, y que carezca de las restricciones que recién enumeramos, a la vez que provea un servicio confiable, escalable, de alta performance, que se adapte a cargas de trabajo variables, tolerante a fallas y que se recupere rápidamente.

Para lograr estos objetivos, algunos paradigmas que impulsaron el desarrollo del SDS (Software Defined Storage) fueron:

- correr sobre hardware básico (commodity hardware), sin requerir de ninguna característica (feature) privada de ningún fabricante.
- escalabilidad horizontal: el agregado de nodos debe permitir un espacio casi ilimitado de almacenamiento y de usuarios.
- evitar la sobrecarga y la centralización, separando las operaciones sobre los datos (read, write), de las que se realizan sobre los metadatos (open, close, list, move, copy, rename, etc…)
- delegar las operaciones de bajo nivel.
- pensar en objetos en lugar de bloques.
- tener siempre en cuenta que los filesystems distribuidos son naturalmente dinámicos, donde la falla es la norma en lugar de la excepción, y la carga de trabajo es continuamente variable.

2.1. El camino recorrido

2.1.1. Los filesystems locales

El diseño de filesystems a lo largo de los años fue influenciado por el filesystem original de Unix y el Fast File System (McKusick,1984) de Unix BSD. Las interfaces y la semántica definidas en los primeros filesystems fueron la base sobre la que posteriormente se desarrolló el estándar POSIX: jerarquía del espacio de nombres, operaciones como open, close, create, remove, rename; permisos y marcas de tiempo…

Estos sistemas de archivos fueron diseñados para operar sobre discos conectados localmente que almacenaban información en bloques o sectores de tamaño fijo. Para minimizar la latencia generada por los movimientos del brazo necesarios para posicionar el

cabezal sobre la pista correcta, junto con la latencia rotacional de los platos, se buscaba asociar las pistas en cilindros y se implementaban distintos algoritmos que mejoraran la performance total.

En los filesystems que heredan de Unix sus diseños (Gooch, 1998), los metadatos se organizan en estructuras llamadas *Inodos*, identificados por un número, que contienen información sobre el tamaño del archivo y número de bloques que ocupa en el disco, en qué dispositivo está almacenado el archivo (DeviceID) , el número de enlaces al inodo, identificadores de usuario y de grupo, marcas de tiempo de creación, modificación, último acceso, y -finalmente- los punteros a los bloques de disco donde está almacenado el archivo. Esta información puede consultarse con el comando `stat <archivo>`. Como el tamaño del inodo es finito, si el número de bloques que ocupa el archivo a referenciar no alcanzan con la lista de punteros del inodo, se usan tablas de direccionamiento indirecto simples, dobles y triples.

Para armar la jerarquía de directorios utiliza otra estructura llamada *dentry* (Love, 2005), que -entre otras cosas- relaciona un nombre de archivo con el inodo correspondiente y un puntero al dentry padre.

Así, para acceder al contenido de un archivo, el sistema operativo debe:

- recorrer el árbol de directorios hasta encontrar la *dentry* correcta que contiene el nombre que el usuario requiere.
- leer el número de inodo.
- encontrar el inodo en el disco.
- leer la información del inodo.
- seguir las referencias a los bloques (directas e indirectas) donde están los datos del archivo.
- leer cada bloque.

Si estos bloques no están contiguos, los movimientos del brazo para ubicar el cabezal lector sobre cada posición se multiplican y la latencia aumenta.

Con el paso del tiempo, las velocidades de transferencia se incrementaron mucho más de lo que se redujeron las latencias de reposicionamiento y rotacional, y aparecieron soluciones como LFS (*Log-structured File System*) (Ousterhout, 1989) que evita los retrasos en las operaciones de escritura, organizando los datos en disco en un log secuencial, donde los metadatos se registran en el log periódicamente para permitir encontrar y leer los datos. Lo que más complicaba esta solución eran las operaciones de borrado, porque necesitaba de un proceso que reorganizara el log, esto degradaba la performance conseguida. Siguiendo estos diseños, desarrollos como DualFS (Piernas, 2002) buscan mejorar la performance general, separando los datos y metadatos en distintos dispositivos, usando en cada uno los diseños de LFS, y definiendo un archivo especial para mapear inodos a ubicaciones en el disco.

Como el tamaño de los archivos se incrementó considerablemente desde que estos sistemas de archivo se diseñaron, y el tamaño de bloques en disco continuó siendo de 4 KB, la cantidad de bloques requeridos para almacenarlo también creció y con ella la cantidad de indirecciones en los inodos. Para mitigar este problema, algunos filesystems

como ext4 (ext4, 2020) y XFS (Sweeney, 1996) reemplazan las listas de ubicación (allocation lists) con *extents*, que consisten en una tupla (inicio, longitud) utilizadas para referenciar grandes regiones del disco de manera compacta.

Otro avance importante para los filesystems locales fue la introducción de *journals* para mejorar la confiabilidad y la recuperación ante fallas. XFS (Silicon Graphics, 2020) implementa un archivo journal en disco que utiliza para registrar las actualizaciones que va a realizar sobre los metadatos. Ante una falla, se puede recorrer el journal para asegurarse de que las actualizaciones se aplicaron de manera completa y correcta, o -en el caso de que una operación no se terminó de completar- volver al punto en que los datos eran consistentes. Si bien esta característica conlleva una penalización sobre la performance, porque los discos deben reposicionar sus cabezales para realizar operaciones de lectura y escritura ahora sobre el archivo de journal y cada actualización sobre los metadatos es realizada dos veces (una sobre el journal, otra sobre el metadato en sí), evitan la necesidad de los costosos chequeos de consistencia que deben realizarse cuando ocurre una falla.

Y un par de características más que vale la pena mencionar en estos párrafos, son las *actualizaciones livianas (soft updates)* (McKusick, 1999) en las que las modificaciones se escriben ordenadas en regiones del disco sin ocupar, asegurando que el archivo en disco es consistente. El WAFL file system (*Write Anywhere File Layout, 2020)* implementa una forma de *copy-on-write* (COW) cuando actualiza las estructuras de metadatos, escribiendo los datos nuevos en zonas sin utilizar del disco, y esos cambios se *comitean* cambiando punteros.

Copy-on-write es una técnica en la cual, si un recurso (bloque, objeto…) debe ser duplicado pero no modificado, no necesariamente se crea un nuevo recurso, sino que el original es compartido por la referencia original y la copia. Si las modificaciones *necesariamente* deben generar una copia del recurso, entonces la operación se difiere hasta la primera operación de escritura. Esta característica reduce el espacio de almacenamiento utilizado por copias sin modificar de bloques, sectores u objetos, y agrega un pequeño overhead a la hora ejecutar una operación de modificación.

Btrfs (*btrfs Wiki*, 2020) y ZFS/openZFS implementan copy-on-write, que permite generar snapshots de archivos y directorios, compresión, soporte para múltiples discos (RAID) y comprobaciones on-line de consistencia, características todas que vamos a querer ver implementadas en los sistemas de almacenamiento distribuido.

2.1.2. Filesystems Cliente-Servidor

La necesidad de compartir los datos y el almacenamiento a través de la LAN, impulsaron el desarrollo de varios sistemas de archivos con arquitectura cliente-servidor. Los que vencieron la prueba del tiempo son, sin dudas, NFS (RFC 7530) y CIFS (Hertel, 2004). Ambos implementan un modelo en el que el servidor exporta una parte de la

jerarquía de su sistema de archivos, que el cliente puede mapear en su espacio de nombres local.

Como se mencionó, el almacenamiento centralizado facilitó la creación de hardware especializado llamado Network Attached Storage (NAS), aunque esta centralización de datos y de operaciones resultó ser un impedimento a la escalabilidad, obligando a los administradores a migrar grandes volúmenes de datos de uno a otro dispositivo, conforme la demanda crecía o el espacio asignado era subutilizado.

Los filesystems de red usualmente relajan la consistencia de la semántica para preservar la performance de la cachés en un ambiente distribuido. Por ejemplo, los clientes NFS graban los datos en el servidor de manera asíncrona, y si otro cliente accede al mismo archivo, no siempre va a obtener la última copia de los datos. De manera similar, los clientes cachean los metadatos de los archivos (*stat*) de manera de limitar la interacción y la carga generada en el servidor. Esta *relajación* en la consistencia del filesystem, puede causar problemas en algunas aplicaciones. (Ver referencia MySQL (2020) sobre advertencias con el uso de MySQL en directorios montados por NFS)

En las arquitecturas de acceso concurrente masivo a dispositivos de almacenamiento, se comprueba que la mayoría de los requerimientos de E/S son paralelos y sin relación unos con otros, y se opta por delegar la tarea de gestionar el filesystem en el usuario. Estas soluciones de almacenamiento llamadas SAN (Storage Area Network) exportan volúmenes a nivel de dispositivo de bloques, comúnmente llamados *luns*, permitiendo que los hosts conectados envíen comandos directamente a los discos, utilizando un protocolo distribuido para gestionar los bloqueos (Kronenberg,1986). Los protocolos más utilizados en estos ambientes son FibreChannel, iSCSI y AoE (ATA over Ethernet).

2.1.3. Filesystems Distribuidos

Los primeros sistemas de archivos distribuidos intentan abordar los desafíos fundamentales de balanceo de carga, escalabilidad y recuperación ante fallas, inherentes a los sistemas cliente-servidor.

Para poder balancear la carga de acceso a un recurso, algunos desarrollos implementan la generación de réplicas de dichos recursos en distintos servidores, distribuyendo el acceso de los clientes entre ellos y asegurando la disponibilidad cuando uno de los servidores falla. Otros modelos realizan la distribución del espacio de nombres (distribuyendo la jerarquía de directorios y archivos) en varios servidores, donde cada uno gestiona sólo una parte de esos archivos.

El aumento de escala de clientes accediendo a los archivos, generaba también situaciones de solicitudes concurrentes sobre el mismo recurso, que el filesystem distribuido debía gestionar.

Muchos de estos diseños iniciales están presentes -con sus adaptaciones- en los sistemas de almacenamiento definido por software (Software Defined Storage - SDS) actuales, y por eso vale la pena hacer una sucinta enumeración de las funcionalidades que, de una u otra manera, marcaron una línea en el diseño que muchos continuaron.

De la mano de la Carnegie Mellon University surgen Andrew File System (AFS) (Howard, 1988), y CODA (Satyanarayanan, 1990). Ambos utilizan un servidor central para coordinar los accesos al sistema de archivos, y otorgan *préstamos* sobre los datos y metadatos, con tiempo de caducidad específico (de modo de mantener consistencia en las cachés de los clientes) y también pueden solicitar *callbacks* de dichos préstamos frente a una situación de conflicto de permisos. En contraste con NFS (versión 3 y anteriores, que son *stateless* por diseño y que -como se comentó el los párrafos anteriores, sacrifican consistencia frente a accesos compartidos), AFS adopta un modelo de consistencia sobre las operaciones *open* y *close*, en lugar de hacerlas sobre las operaciones de *read/write*.

Las arquitecturas distribuidas frecuentemente replican los datos a través de múltiples servidores, para aumentar la confiabilidad y la disponibilidad en presencia de una falla. En el filesystem distribuido Harp (Liskov, 1991), los servidores emplean una replicación *primary-copy* en que las actualizaciones se realizan sobre la copia primaria del recurso y las modificaciones se trasladan posteriormente a las copias secundarias, y *write-ahead logs,* que cumplen la función del journal en los filesystems locales, para permitir la rápida recuperación frente a la falla de un nodo. En Petal (Thekkath, 1996) se provee a los clientes de una abstracción de discos virtuales que son gestionados por un cluster de servidores, y FAB (Federated Array of Bricks) (Saito, 2004), utiliza un protocolo de consistencia de réplicas basado en el voto de la mayoría.

Así como los sistemas de archivos locales descansan en tecnologías tipo RAID para obtener confiabilidad frente a fallas, sistemas distribuidos como Frangipani (Thekkath, 1997) implementaban una abstracción de la capa de almacenamiento de manera replicada, distribuida y confiable sobre la cual construían el filesystem.

2.2. Teorema de CAP

La presencia de la red de datos conectando a los distintos servidores del sistema de almacenamiento distribuido agrega un punto más de falla y un recurso más que debe optimizarse y monitorearse, porque no pocas veces es la responsable de la aparición de errores, de pérdida de datos y caídas en la performance.

La repentina desconexión de uno o varios servidores del cluster, es una situación que el sistema de archivos debe monitorear, detectar y sobre la que debe tomar decisiones.

Como se mencionó previamente, la replicación de datos es una estrategia común en los sistemas de almacenamiento distribuido para lograr fiabilidad y disponibilidad frente a una falla. No obstante, cuando una parte de los servidores se desconecta del resto, surgen problemas que caen dentro de lo que se dio en llamar “Fallas Bizantinas”, luego de que

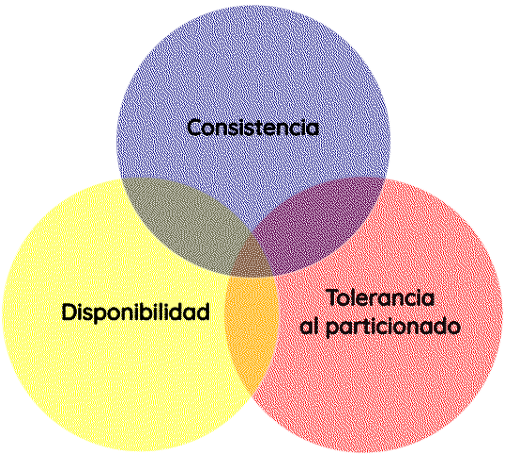
Leslie Lamport (1982) describiera la situación en su paper "*Problema de los Generales Bizantinos*" que puede enunciarse como una situación en la que una parte del sistema falla y no se cuenta con la información correcta para determinar qué parte del sistema es la que presenta la falla y cuál la que tiene la información correcta.

En nuestro caso, si dos servidores que contienen la misma réplica de un archivo se desconectan entre sí, y un archivo es modificado por distintos usuarios en cada uno de ellos, cuando vuelven a conectarse, ¿quién tiene la versión correcta del archivo? Cada filesystem distribuido implementa algún tipo de *protocolo de consenso* para coordinar las réplicas y resolver estas situaciones.

Fischer (1985) plantea en su artículo *"Impossibility of distributed consensus with one faulty process"* que la presencia de una falla en un sistema distribuido asíncrono, puede llevar a un estado de indeterminación cualquiera sea el algoritmo de consenso que se utilice.

Los sistemas de almacenamiento distribuido son esencialmente sincrónicos, y eso permite abordar el problema del consenso a través de la implementación del algoritmo PAXOS (Lamport, 2001 y 2005) un algoritmo para llegar a consensos en sistemas distribuidos con cierto grado de tolerancia a fallos, donde hay que ponerse de acuerdo sobre uno de los resultados entre un grupo de participantes.

Estas situaciones se enmarcan en el Teorema de CAP, también conocido como *conjetura o teorema de Brewer* (Gilbert, 2002), surge de una suposición del informático Eric Brewer, mencionada durante su conferencia en el Simposio sobre Principios de Computación Distribuida en el año 2000. En 2002, Seth Gilbert y Nancy Lynch, del MIT, demostraron su validez con evidencia axiomática.



Postulado inicialmente para bases de datos distribuidas, este teorema enuncia que un sistema distribuido que comparte datos a través de una red, sólo puede garantizar dos de estas tres propiedades: Consistencia, Disponibilidad y Tolerancia a las particiones (CAP = Consistency, Availability, and Partition Tolerance):

- Consistencia: Todos los nodos ven la misma información al mismo tiempo.
- Disponibilidad: Cada petición a un nodo recibe una confirmación (ACK), ya sea por operación exitosa o fallida.
- Tolerancia a las particiones: el sistema continúa funcionando normalmente cuando se pierden mensajes entre los nodos, o algunos directamente se desconectan de la red.

Este teorema, aplicado a los sistemas de almacenamiento distribuido, pone el foco en la *escala* del mismo y en que ningún sistema está exento de las fallas de la red de datos,

de manera que **la partición debe ser tolerada**: algunos mensajes van a perderse, algunos nodos van a desconectarse del resto.

En presencia de una partición, hay dos opciones:

- Limitar el servicio para preservar la consistencia de los datos,
- o continuar sirviendo los datos, a expensas de la consistencia.

Si el diseño del sistema opta por la CONSISTENCIA entonces se devuelve un error, o un time-out si no puede garantizar que la información que debe proveer está actualizada.

Si opta por la DISPONIBILIDAD, el sistema intenta devolver la versión disponible más reciente de la información solicitada, aunque no pueda garantizar su validez.

2.3. De la LAN a la WAN

Algunos sistemas llevan más allá de las redes LAN los desarrollos de sistemas de archivos, distribuyendo los componentes en la incipiente Internet. Esto trajo aparejado otro gran abanico de problemas que había que solucionar. La seguridad no era un requerimiento que quitara el sueño a los desarrolladores, al menos no tanto como la latencia que introducían las distintas redes WAN sobre las que los datos debían transportarse. Una aproximación común fue de cachear muchos datos en los clientes, y así minimizar la interacción con los servidores, aunque eso perjudicaba la consistencia de los archivos compartidos. xFS (Wang, 1993) utilizaba un protocolo de chequeo de consistencia de las cachés, permitiendo la actualización desde las cachés de otros clientes (Dahlin 1994), minimizando la comunicación a través de la WAN con el repositorio primario, con el costo de aumentar la complejidad de la sincronización.

OceanStore (Kubiatowicz, 2000) apunta a construir un servicio global de almacenamiento de archivos basado en *erasure codes*, un método de protección de datos en el que los datos se dividen en fragmentos, se codifican de manera de generar datos redundantes y se almacenan en un conjunto de diferentes medios de almacenamiento. Pangaea (Saito, 2002) apunta a un escenario similar, en el que los archivos son replicados agresivamente cada vez que son leídos, de manera de lograr una baja latencia de acceso y alta disponibilidad. Crea réplicas dinámicamente y construye un esquema aleatorio de réplicas para cada archivo para propagar las actualizaciones de manera eficiente. Utiliza una semántica de coherencia “optimista” pues relaja la consistencia entre las réplicas, de manera de minimizar el costo de las comunicaciones a través de la red WAN.

Siguiendo el diseño del filesystem Cedar de Microsoft (Gifford, 1988), que transformaba el problema de la consistencia en un problema de versionado de archivos compartidos marcándolos como “inmutables”, Venti (Quinlan, 2002) propone que *todos* los datos son inmutables (*write-once policy)* y se mantienen versiones de todos los archivos.

2.4. De Bloques a Objetos

Los desarrollos más recientes como Lustre (Wang, 2009), Panasas (Welch, 2008), Sorrento (Tang, 2004), Ursa Minor (Abd-El-Malek, 2005) y Kybos (Wong, 2005) adoptan arquitecturas basadas en *almacenamiento basado en objetos* (Azagury, 2003) en las que en lugar de realizar operaciones de E/S en pequeños bloques de tamaño fijo, los datos se almacenan en objetos con nombre, tamaño variable y otros metadatos (como permisos de acceso, cantidad de copias, estados, etc…)

De esta manera, el almacenamiento basado en objetos permite distribuir funcionalidades de bajo nivel en dispositivos semi-inteligentes, reduciendo la carga sobre los servidores de metadatos y mejorando la escalabilidad total del sistema. La conexión directa de los servidores del sistema de archivo con los discos rígidos se reemplazan por dispositivos inteligentes de almacenamiento de objetos (OSDs), que combinan procesamiento, conexión a la red de datos, cachés locales y -finalmente- uno o más discos. Los clientes se comunican con los servidores de metadatos (Metadata Servers, MDS) para operaciones que los involucran (open, move, rename, list…) y se comunican directamente con los OSDs para realizar las tareas de E/S sobre los archivos (read, write).

Exceptuando a Sorrento, todos los sistemas mencionados utilizan un servidor para mapear la ubicación en la que se almacenó cada objeto. Este servidor se ve involucrado en cada operación sobre los objetos, limitando la eficiencia y la escalabilidad.

2.5. Almacenamiento Definido por Software

De la mano del surgimiento de los protocolos como openFlow (ONF, 2015), que pusieron en el centro de las discusiones las Redes Definidas por Software, aparecieron también algunos trabajos (Thereska, 2013) que analizaban las operaciones de Entrada/Salida y los protocolos involucrados en esa misma arquitectura, en la que se separan por diseño el plano de control del plano de datos.

La virtualización del cómputo dio paso a la virtualización de la red de datos, y ésta a la virtualización del almacenamiento (Darabseh, 2015). El SDS es parte de un ecosistema más grande, que sintéticamente se puede definir como "todo definido por software", en donde el software está separado del hardware y donde las funcionalidades se programan, y el desafío consiste en definir con calidad y precisión “qué hacer” para dejar que el sistema provea el “cómo hacerlo”.

El almacenamiento definido por software es una arquitectura que separa el software de almacenamiento de su hardware (Wu, 2013). A diferencia del almacenamiento conectado a la red (NAS) tradicional o de los sistemas de red de área de almacenamiento (SAN), el SDS por lo general está diseñado para ejecutarse en cualquier arquitectura x86, y de esa manera el software no depende del hardware propietario.

Separar el software de almacenamiento de su hardware otorga flexibilidad y escalabilidad, porque permite utilizar hardware heterogéneo para expandir la capacidad de manera horizontal, o actualizar y reemplazar equipamiento, sin afectar el servicio.

¿Qué atributos definen al SDS? (RedHat, 2019)

- Automatización: Administración simplificada que mantiene bajos los costos.
- Interfaces estándares: Una interfaz de programación de aplicaciones (API) para la administración y el mantenimiento de los dispositivos y servicios de almacenamiento.
- Una ruta de datos virtualizada: Interfaces de objetos, archivos y bloques que son compatibles con las aplicaciones escritas para esas interfaces.
- Escalabilidad: La capacidad de escalar horizontalmente la infraestructura de almacenamiento sin que esto afecte el rendimiento.
- Transparencia: La capacidad de supervisar y administrar el uso del almacenamiento y, al mismo tiempo, saber qué recursos están disponibles y qué costos tienen.

Ya se mencionó que el almacenamiento tradicional es monolítico. Se vende como un conjunto de hardware (que suele ser el estándar del sector) y software propietario. Pero la utilidad del SDS radica en su independencia de cualquier hardware específico, pues no hace suposiciones acerca de la capacidad o utilidad del hardware subyacente.

El SDS es una capa de software entre el almacenamiento físico y las solicitudes de datos de los usuarios, que permite manipular la manera y el lugar donde se almacenan los datos y cómo estos se intercambian entre los servidores y clientes.

La flexibilidad mencionada se combina con la capacidad de programación de estos sistemas para permitir un almacenamiento que se adapte rápida y automáticamente a las demandas. La capacidad de programación incluye la gestión de recursos basada en políticas y el aprovisionamiento y la reasignación de la capacidad de almacenamiento de manera automática.

La naturaleza independiente del software de este modelo de implementación también facilita en gran medida los SLA (*Service Level Agreement*, Acuerdo de Nivel de Servicio) y la definición de QoS para diferentes tipos de volúmenes o conjunto de unidades, permitiendo definir distintos tipos de almacenamiento para distintos tipos de cargas de trabajo. Por supuesto, estos ajustes en la performance diferentes para cada “perfil” de trabajo, fue una de las motivaciones que impulsaron esta tesis.

2.6. El escenario actual

2.6.1. HDFS

Entre los sistemas de almacenamiento distribuido que continúan desarrollándose y que están en diferentes etapas de adopción por parte de las empresas e instituciones que requieren almacenamiento masivo, podemos mencionar a *Hadoop Distributed File System* (HDFS) (The Apache Software Foundation, 2015), basado en los desarrollos iniciales de Google File System (Ghemawat, 2003).

HDFS es el componente principal del ecosistema Apache Hadoop, que hace posible almacenar conjunto de datos masivos con tipos de datos estructurados, semi-estructurados y no estructurados. Es un sistema distribuido basado en Java que crea una capa de abstracción en la que los archivos se almacenan divididos en bloques de un mismo tamaño (128 MB) y estos se distribuyen en los nodos que forman el cluster.

Para conseguir una alta escalabilidad, HDFS usa almacenamiento local que escala horizontalmente. Para mantener la integridad de los datos, almacena por defecto 3 copias de cada bloque de datos. Esto significa que el espacio necesario en HDFS es el triple, por lo que el costo también aumenta.

La arquitectura de HDFS (Ver Figura 2) es de tipo maestro-esclavo y está basada en dos componentes principales: *NameNodes* y *DataNode*. Mientras que el DataNode almacena los datos, el NameNode es el nodo principal del sistema cuya función es gestionar el acceso a los datos y almacenar sus metadatos. Este componente es el único nodo que conoce la lista de archivos y directorios del cluster. El filesystem no se puede usar sin el NameNode. Al ser un punto de falla crítico, Hadoop 2 introduce el NameNode en alta disponibilidad.

Frente a cualquier operación sobre los metadatos (búsquedas, creación, eliminación, copia, movimiento, etc.) debe hacerse referencia al NameNode, limitando la escalabilidad y la confiabilidad. (Gudu, 2015) Esto sumado a que el tamaño de bloque de 128 MB hace que este filesystem distribuido no funcione de forma óptima con archivos pequeños debido a la fragmentación interna, invita a analizar las otras opciones disponibles.

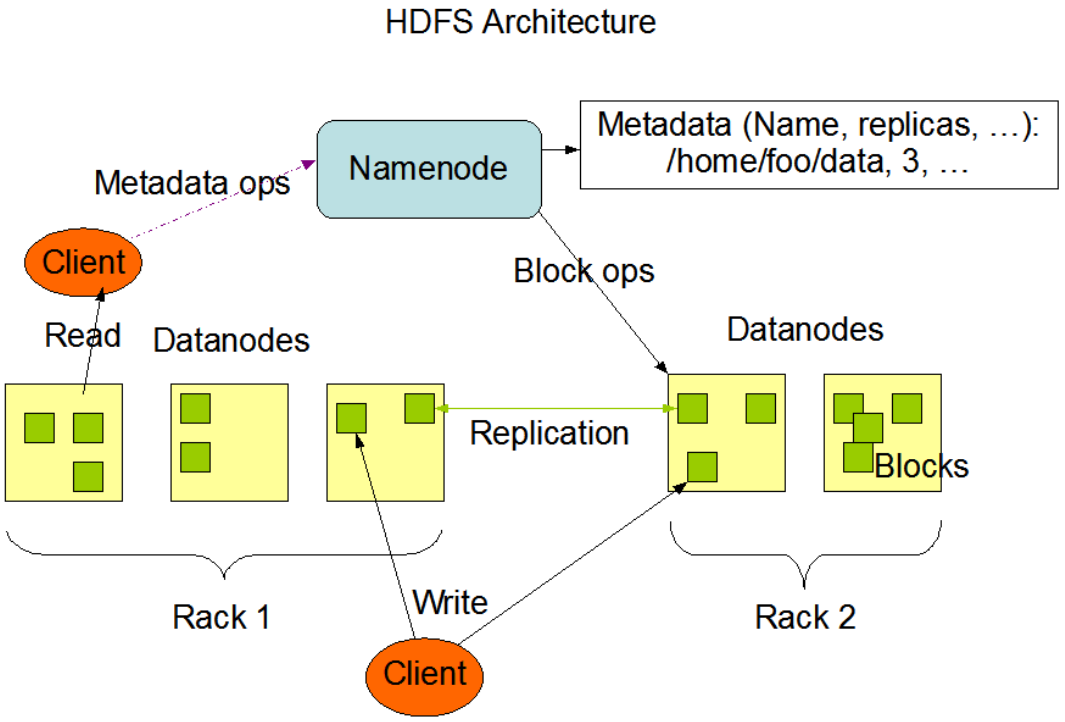


Figura 2. Arquitectura HDFS

2.6.2. GPFS

GPFS (General Parallel File System) es un sistema de archivos distribuido de alto rendimiento desarrollado por IBM (Barkes, 1998), ampliamente utilizado para proveer almacenamiento a clusters de alto desempeño con muy buena performance. No obstante, GPFS es un filesystem propietario, normalmente deployado sobre hardware propietario. Su alto costo y su dependencia del proveedor son argumentos que nos hacen desestimar su uso en nuestro contexto.

2.6.3. Lustre

Nacido en la Universidad Carnegie-Mellon, e impulsado luego por la *European Open File System*, surge Lustre (Lustre, 2020), una plataforma de software de código abierto que implementa un sistema de archivos paralelos distribuidos, diseñado para lograr escalabilidad, alta performance y alta disponibilidad (Figura 3). Provee un espacio de nombres POSIX-compliant que puede soportar cientos de Petabytes de almacenamiento y un throughput agregado de cientos de GB por segundo. Orientado a proveer almacenamiento a los clusters de cómputo de alto desempeño (HPC), sus componentes principales son los Servidores de Metadatos (MDS) , Metadata Targets (MDT), Servidores de Objetos (OSS), Object Server Targets (OST) y los clientes.

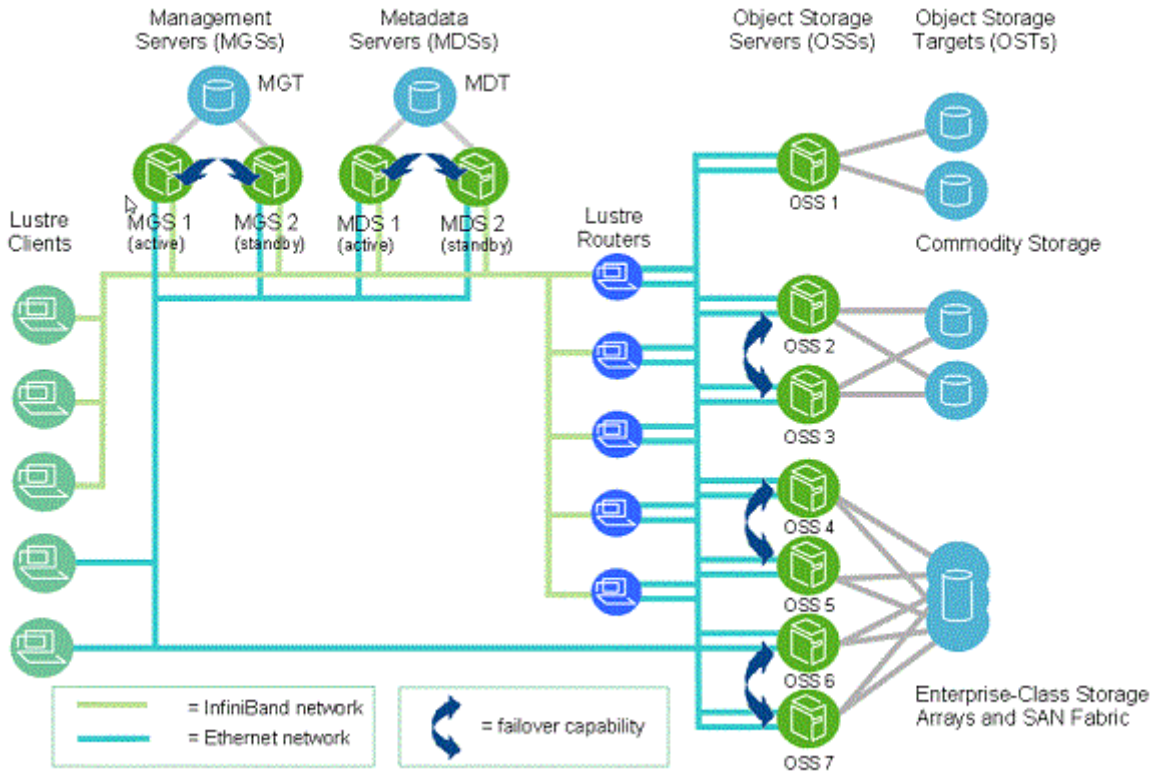


Figura 3 - Arquitectura de Lustre

Su principal característica es la capacidad de escalar exponencialmente, tanto en almacenamiento como en performance, y al estar orientado a clusters HPC, se adapta muy bien a cargas de trabajo en las que grandes archivos son accedidos de manera paralela por cientos de clientes de manera concurrente, aunque no tiene buen servicio para RESTFUL, S3 y Cloud Integration (Poat, 2017). Además, tanta potencia tiene sus costos: la implementación y gestión no es trivial, y no tiene la flexibilidad de uso que buscamos: ofrecer el acceso a nivel de archivos, objetos y volúmenes completos.

2.6.4. GlusterFS

GlusterFS (Gluster, 2016) es un sistema de archivos en red escalable y distribuido, utilizado en escenarios con uso intensivo de datos, como media streaming o cloud storage. Está diseñado para utilizarse en el espacio de usuario. Inicialmente desarrollado por Gluster y luego adquirido por RedHat, que lo renombra como “Red Hat Gluster Storage” (Red Hat, 2020)

El servidor GlusterFS es sencillo: exporta un sistema de archivos existente como un volumen, dejando en manos de los *traductores* que ejecutan en los clientes el armado de la estructura del almacenamiento. Los propios clientes se manejan independientemente, no se comunican directamente entre sí, y los traductores administran las consistencia de los datos

entre ellos. Para ubicar los datos, GlusterFS se basa en un algoritmo de hash en vez de utilizar un modelo de metadatos centralizados o distribuidos (Johari, 2014). Desde la versión 3.1, los volúmenes pueden ser agregados, eliminados o migrados en forma dinámica, esto ayuda a prever problemas de consistencia, y permite que el GlusterFS pueda ser escalado a varios petabytes sobre hardware de bajo coste, evitando así los cuellos de botella que normalmente afectan a muchos sistemas de archivos distribuidos con múltiple concurrencia.

Si bien cuenta entre sus características principales la simplicidad de instalación y operación, la flexibilidad y escalabilidad, no exporta los volúmenes como dispositivos de bloque.

2.6.5. Ceph

Concluímos entonces que la mejor opción disponible, la más robusta, cuyos requerimientos de hardware son notoriamente flexibles, y que más ha sido adoptada por empresas e instituciones de diferentes tamaños, desde organizaciones chicas hasta el CERN (Van Der Ster, 2014), es Ceph.

Ceph (Ceph, 2020) es un sistema de almacenamiento distribuido de código abierto que corre sobre hardware genérico, diseñado para proveer escalabilidad, confiabilidad y performance. Está basado en un servicio de almacenamiento de objetos distribuidos llamado RADOS (Reliable Autonomic Distributed Object Store) que maneja la distribución, replicación y migración de objetos de tamaño fijo. Sobre esa capa que abstrae de manera confiable el almacenamiento, Ceph implementa una serie de servicios para proveer dispositivos de bloque (RBD, Rados Block Device), un sistema de archivos distribuido (CephFS) y almacenamiento de objetos (RGW) similar a S3 de Amazon Web Services.

Ceph ofrece almacenamiento confiable sobre hardware inseguro, a través de esquemas de replicación de datos o de algoritmos erasure-code, un algoritmo matemático para reconstruir datos corrompidos o perdidos. Evita tener puntos de falla únicos (SPoF) y cuida que el servicio no se vea afectado por actualizaciones, reemplazo o agregado de discos, verificaciones de integridad o cualquier otra tarea administrativa.

Siendo Ceph nuestra elección, vamos a profundizar en su arquitectura, características y funcionamiento en el capítulo siguiente.

3. ARQUITECTURA de CEPH

Ceph es producto de la tesis de doctorado de Sage Weil en Ciencias de la Computación en la Universidad de California Santa Cruz (Weil, 2007), quien libera el código bajo licencias open-source. En 2010, el cliente de Ceph es incorporado al kernel de Linux y en 2012, Weil funda la empresa Inktank para ofrecer servicios de soporte profesional para Ceph. En 2014, Red Hat compra Inktank y en 2015 grandes empresas e instituciones que están comprometidas con el desarrollo de Ceph, como Canonical, CERN, Cisco, Fujitsu, Intel, SanDisk, SUSE, RedHat y otras forman el Ceph Community Advisory Board para asistir a la comunidad en la dirección del proyecto.

En su conferencia técnica *Intro to Ceph* (Weil, 2019), Sage presenta las principales características de manera sintética, diciendo que Ceph es un software de código abierto, que corre sobre hardware genérico tanto en referencia a los servidores y a la red de datos, como a los discos y que ofrece interfaces de almacenamiento a nivel de objetos, de archivos y de dispositivos de bloque. Ofrece almacenamiento confiable sobre hardware inseguro, a través de esquemas de replicación de datos o de algoritmos *erasure-code,* un algoritmo matemático para reconstruir datos corrompidos o perdidos. Evita tener puntos de falla únicos (SPoF) y cuida que el servicio no se vea afectado por actualizaciones, reemplazo o agregado de discos, verificaciones de integridad o cualquier otra tarea administrativa. Las imágenes que se muestran en este capítulo están basadas en dicha conferencia técnica.

3.1. Componentes

Ceph provee un almacenamiento unificado, basado en un servicio de almacenamiento de objetos distribuidos llamado RADOS (Reliable Autonomic Distributed Object Store) (Weil, 2008) que se encarga de la distribución, replicación, balanceo, reparación y migración de objetos de bajo nivel, y constituye una capa común de almacenamiento para los servicios de bloques, objetos y archivos.

Haciendo referencia al Teorema de CAP mencionado en el capítulo anterior, podemos decir que RADOS *privilegia la consistencia* en una situación de partición.

ceph-mon

ceph-mgr

ceph-osd

Los componentes iniciales de software de RADOS son 3:

- MON (Monitor)
- MGR (Manager)
- OSD (Object Storage Daemons)

Los MON son la autoridad central para la autenticación de usuarios, nodos y servicios. Mantienen una copia maestra del mapa del cluster, y organizan la ubicación de datos y las diferentes políticas de acceso, replicación, etc… Son

el punto de coordinación para los otros componentes del cluster. Siempre deben ser más de 2 (por alta disponibilidad) y número impar (la documentación recomienda 3 o 5), pues gestiona las situaciones críticas del cluster mediante consenso Paxos (Lamport, 2001). Los clientes, cuando se conectan al cluster, obtienen una copia del mapa a través de los monitores.

Los MGR recopilan métricas en tiempo real: throughput, uso de disco, latencias, operaciones de recuperación, fallas, etc… y suelen ejecutarse en el mismo servidor donde corre el MON.

Los OSD almacenan físicamente los datos en un HDD, SSD o NVMe, y atienden los requerimientos de E/S de los usuarios. Se coordinan para balancear y replicar los datos de manera colaborativa. Cada OSD chequea su estado y el de otros OSDs y lo informan al MON, de manera de tener un reporte continuo y consistente de la situación del cluster en cada momento. Cada OSD maneja un único dispositivo y toda la bibliografía recomienda que este dispositivo sea un disco nativo, no un RAID, no un volumen LVM, no un LUN exportado de un SAN, no una unidad remota montada.

3.2. CRUSH

Uno de los aportes más significativos de la tecnología de Ceph es el algoritmo CRUSH (Controlled Replication Under Scalable Hashing) (Weil, 2006), una función pseudodeterminística que permite **calcular**, tanto a los OSDs como a los clientes, la ubicación de los objetos dentro del cluster en base al nombre del objeto y al mapa del cluster, en lugar de buscarla en un servidor centralizado. Luego, las operaciones de E/S se realizan directamente contra los OSD. Esto promete, en teoría, una extrema capacidad de escalabilidad. (Figura 4)

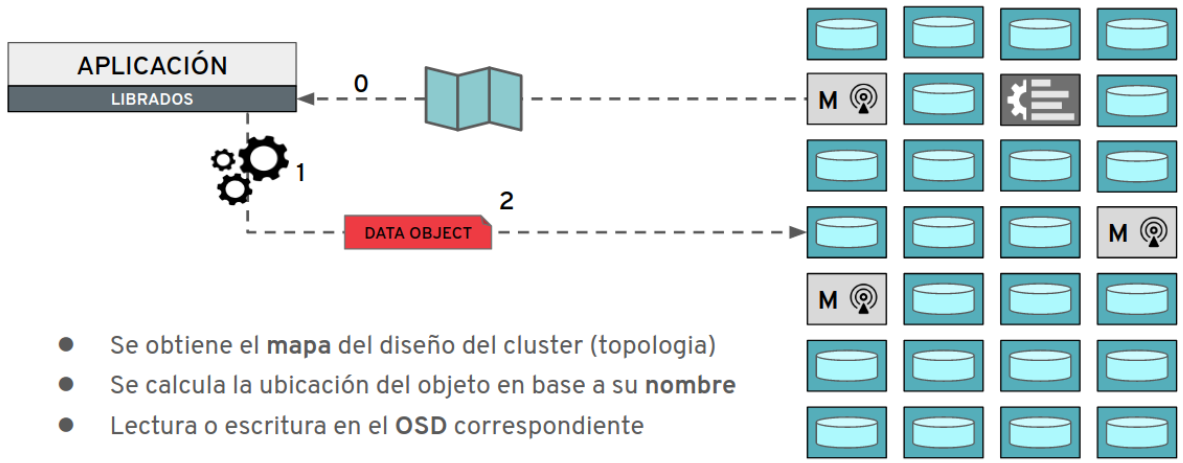


Figura 4. Cálculo de ubicación a través de función de mapeo

CRUSH también permite definir jerarquías de dominios de falla en los que ubicar las réplicas de los datos, en diferentes niveles: discos, hosts, racks, ...datacenters! Basados en esta jerarquía y en distintas reglas, CRUSH mapea varios objetos de bajo nivel que

pertenecen a un mismo *pool* de almacenamiento en una agregación de objetos llamada *placement groups* y éstos se mapean (y replican) en los OSDs.

Por ejemplo, si utilizamos una regla que indique que deseamos 3 réplicas de cada objeto, y configuramos nuestra jerarquía de modo que el dominio de falla sea el servidor (*host)*, entonces Ceph ubicará 3 copias de nuestros objetos, en 3 servidores diferentes. De esta manera, si sólo tenemos un servidor, con 3 discos asociados cada uno a un servicio OSD, Ceph no replicará los objetos. (ver Figura 5)

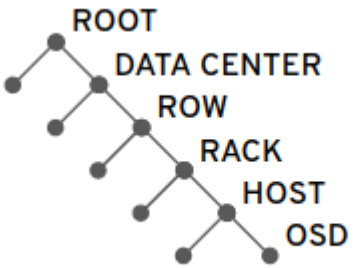


Figura 5: Jerarquía de Dominios de Falla

Cada “dato” (archivo, volumen u objeto) se divide en múltiples “*objetos rados”*, cada uno de los cuales tiene un tamaño fijo, que por defecto es de 4MiB. Estos objetos pertenecen a un pool de almacenamiento, que se divide en diferentes framentos (*stripes)* que constituyen los *Placement Groups (PGs)*. Un pool puede tener millones de objetos, pero sólo 128 PGs. El número de PGs en un pool juega un rol importante en la forma en que el cluster distribuye y balancea los datos, pues cada PG es mapeado a varios servidores, de acuerdo a la jerarquía y a las reglas de replicación. (Figura 6)

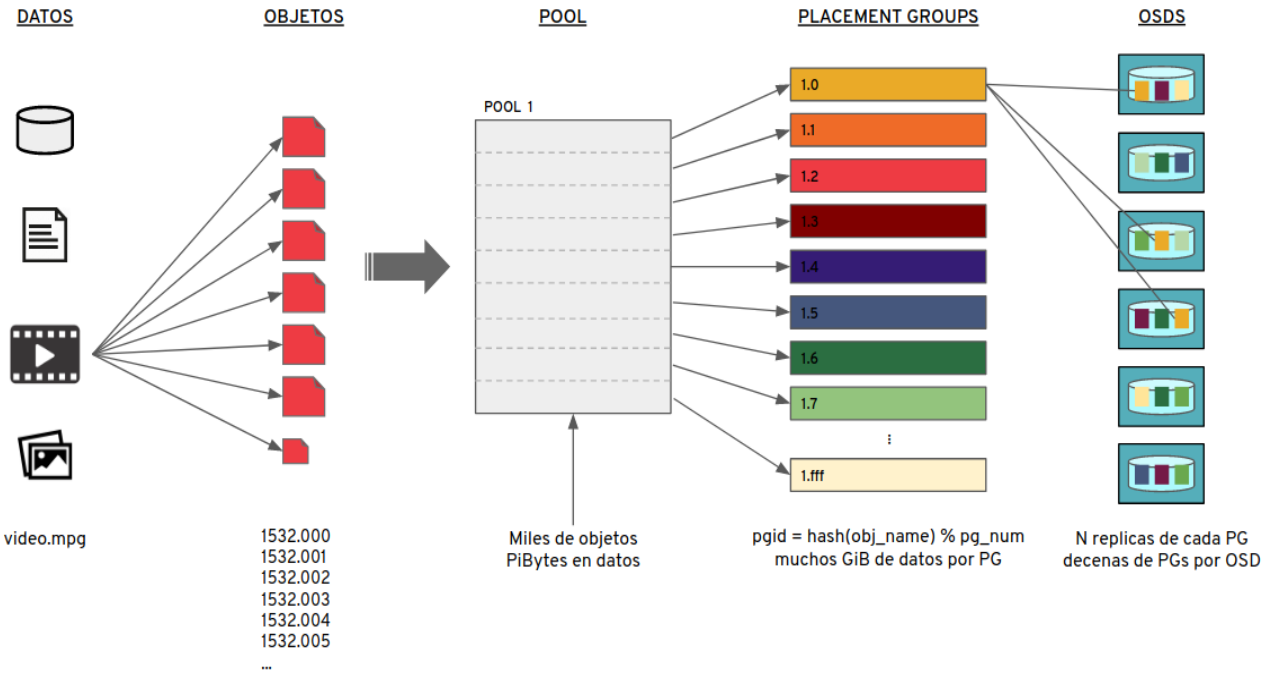


Figura 6. Mapeo de Datos a Objetos y Placement Groups

Para lograr que el almacenamiento sea confiable, seguro y resistente a fallas, Ceph dispone de dos técnicas que pueden configurarse a la hora de crear los pools de almacenamiento.

3.3. Replicación

Se crean copias idénticas disjuntas de cada *Placement Group*, y se almacenan en OSD separados. Los OSDs que almacenan un mismo PG, se llaman *acting set* del PG. Aunque el factor de replicación se puede variar, normalmente por cada PG se crean otras dos copias (1:3) y esto provoca un overhead del 200% o -visto de otra manera- sólo ⅓ del espacio de almacenamiento nativo (*raw)* está disponible. (Figura 7)

La ventaja es que frente a una pérdida de un PG, existen otras dos copias iguales, inmediatamente disponibles para ser utilizadas. Como dice Van der Ster, Ceph es una “solución orgánica”, debido a su habilidad para balancearse y curarse a sí misma, mientras vive en un ambiente heterogéneo de cambio continuo (van der Ster, 2014). En un cluster “saludable”, los clientes se comunican con la copia primaria del objeto, pero en un estado degradado, pueden acceder directamente con una réplica del objeto, en un nodo que registra las operaciones de escritura, de manera que la copia maestra pueda reproducir los cambios cuando se recupere.

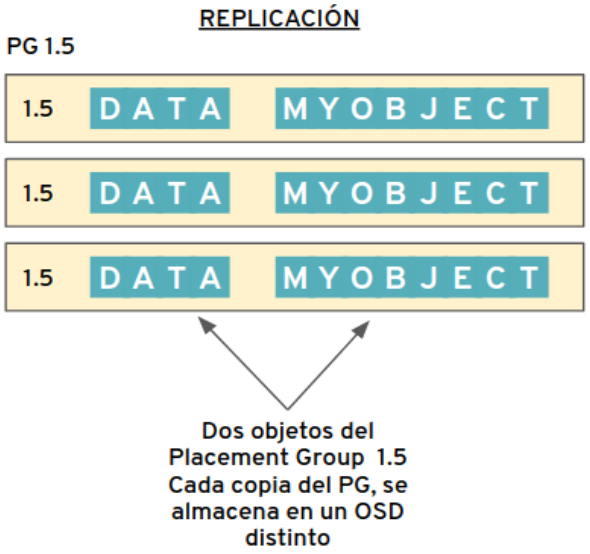


Figura 7: Replicación de PGs

3.4. Erasure Coding

A través de este algoritmo, los objetos se dividen en *k* fragmentos (*stripes*) que se almacenan en distintos bloques (llamados *chunks* o *shards,* dependiendo de la bibliografía*)* del Placement Group, de modo que cada *shard* contiene una porción de la información. A modo de redundancia y paridad, por cada **k** shards, se agregan **m** fragmentos más, en una relación típica 2:1, 5:2 o 5:3 (50% overhead aproximadamente). Estos parámetros (k+m) se establecen al momento de la creación del pool, y se recomienda el uso de *erasure coding* para almacenar objetos grandes y que rara vez se modifican.

De esta manera, se puede crear un pool configurado con *erasure coding* para almacenar archivos históricos, videos, backups, imágenes de volúmenes, etc… y crear otros pools para datos que tienen más movimiento con *replicación.* Los pools, divididos en placement groups, normalmente comparten OSDs con otros pools, aunque esto puede configurarse a través de políticas de ubicación de CRUSH (*CRUSH placement policies)* donde puede establecerse, por ejemplo, que un pool sólo debe almacenarse en dispositivos SSD para obtener mejor performance. Sobre este punto vamos a volver más adelante.

Los pools son escalables bajo demanda, ampliando su capacidad y número de placement groups a medida que más objetos se almacenan. Periódicamente monitorean su estado y, si se encuentran inconsistencias entre las réplicas del mismo PG, se reescribe la réplica fallada con alguna de las 2 copias existentes. Por eso el nivel de réplica es siempre impar y, aunque el parámetro es configurable, en toda la bibliografía se recomienda que sean 3 copias. (Figura 8)

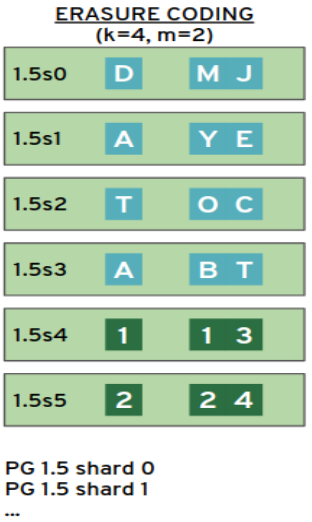


Figura 8: Erasure Coding

3.5. Tres servicios, un mismo cluster

Ceph ofrece almacenamiento para objetos a través de su servicio RGW (Rados Gateway) compatible con S3 de Amazon y Swift de OpenStack, accesible vía una API HTTPS/REST. Posee los mismos componentes que esos sistemas: *buckets, objetos y usuarios* cuyo acceso es controlado a través de permisos establecidos en ACLs. Vale aclarar una vez más que los objetos de RGW *no equivalen* a los objetos RADOS. Los primeros pueden tener tamaños variables de hasta GiB o TiB, mientras que los objetos RADOS son siempre del mismo tamaño (por defecto, 4 MiB). RGW fragmenta los datos en numerosos objetos RADOS, de acuerdo al mecanismo detallado previamente. Entre sus características, también podemos mencionar la capacidad de encriptación de los objetos, compresión, gestión del ciclo de vida, archivo, múltiples clases de almacenamiento (mapeo de clases a diferentes pools RADOS), y búsqueda de metadatos a través de Elasticsearch. (Figura 9)

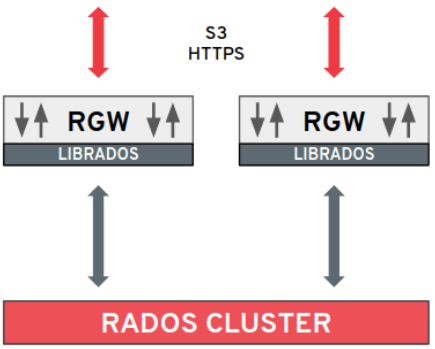


Figura 9: Rados Gateway

También provee servicios a nivel de archivos a través del Ceph File System (CEPHFS). El acceso concurrente y compartido de muchos clientes es gestionado para asegurar una consistencia fuerte y un cacheo coherente: las actualizaciones en un nodo son visibles por todos los otros de manera inmediata. El almacenamiento y gestión de los metadatos está separada de los datos: mientras que la capacidad de almacenamiento de los datos y el throughput de E/S escala con el agregado de OSDs, el espacio de nombres asociado al número de archivos escala con el agregado de servidores de metadatos (MDSs). (Figura 10)

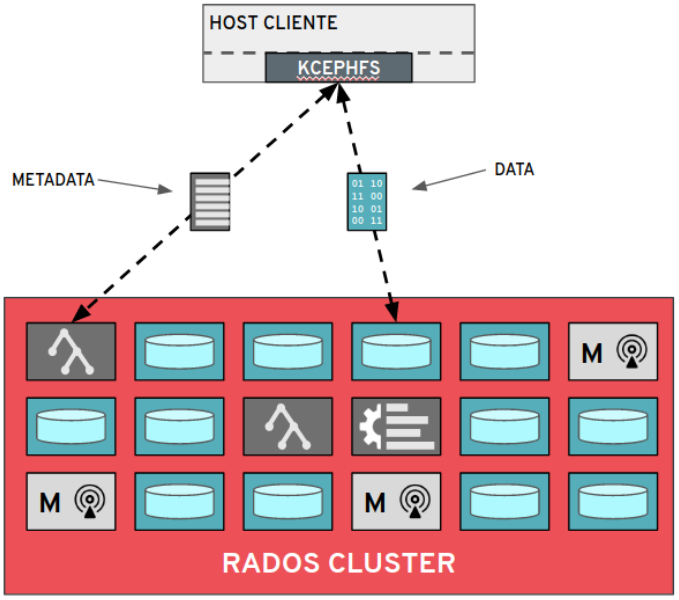
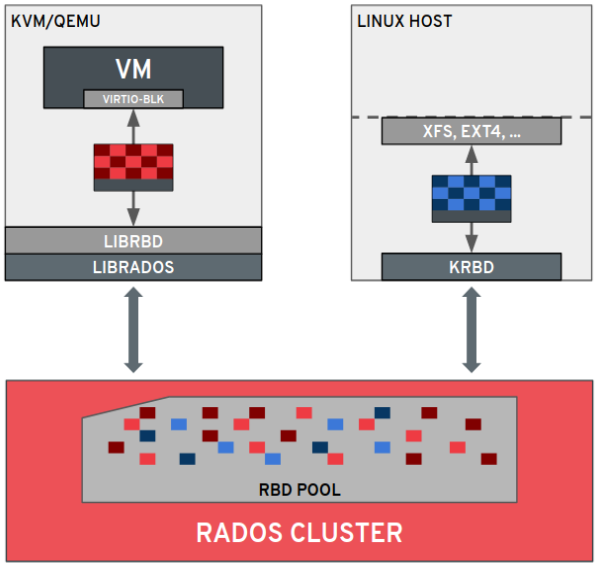


Figura 10. El sistema de archivos almacena datos y metadatos por separado

Los MDS se encargan de gestionar el espacio de nombres del sistema de archivos, y almacenan esa información (nombres, fechas, permisos, atributos extendidos, bloqueos, cuotas…) en objetos RADOS y particionan automáticamente la jerarquía dinámicamente en diferentes ramas basadas en la carga de trabajo, de manera de optimizar los accesos y escalar sin problemas. Pueden implementarse a través de clientes en el kernel, o bien por aplicaciones en espacio de usuario (ceph-fuse).



Finalmente, Ceph provee almacenamiento a nivel de dispositivos de bloque a través de su servicio RBD (Rados Block Devices) que almacena imágenes de discos dividiéndolas en múltiples objetos RADOS que, de manera similar a EBS de AWS (*Amazon Web Services*), desacopla el volumen del hipervisor y del host. Se provee de un cliente RBD implementado para el kernel de Linux y para KVM (*Kernel-based Virtual Machine*) (KVM, 2020) una tecnología para implementar full-virtualization con Linux, integrado con libvirt, Openstack, Kubernetes, Proxmox y otras grandes soluciones de cloud computing. (Figura 11)

Figura 11. Servicio RBD

RBD permite generar snapshots de las imágenes de manera de conservar instantáneas consistentes de sólo lectura, que a su vez pueden clonarse en nuevas

imágenes con tecnología copy-on-write, mediante la cual sólo se consume espacio de almacenamiento cuando los datos de la nueva imagen cambian respecto del snapshot del que se originaron.

Cada imagen RBD posee un nombre, tamaño, parámetros de fragmentación, datos de snapshots y permisos de bloqueo concedidos entre otros metadatos. El tamaño de objeto usualmente es de 4 MiB. Las operaciones sobre las imágenes RBD son registradas en un journal (Figura 12) para asegurar consistencia frente a posibles fallas en el almacenamiento, en la red de datos o en el cluster:

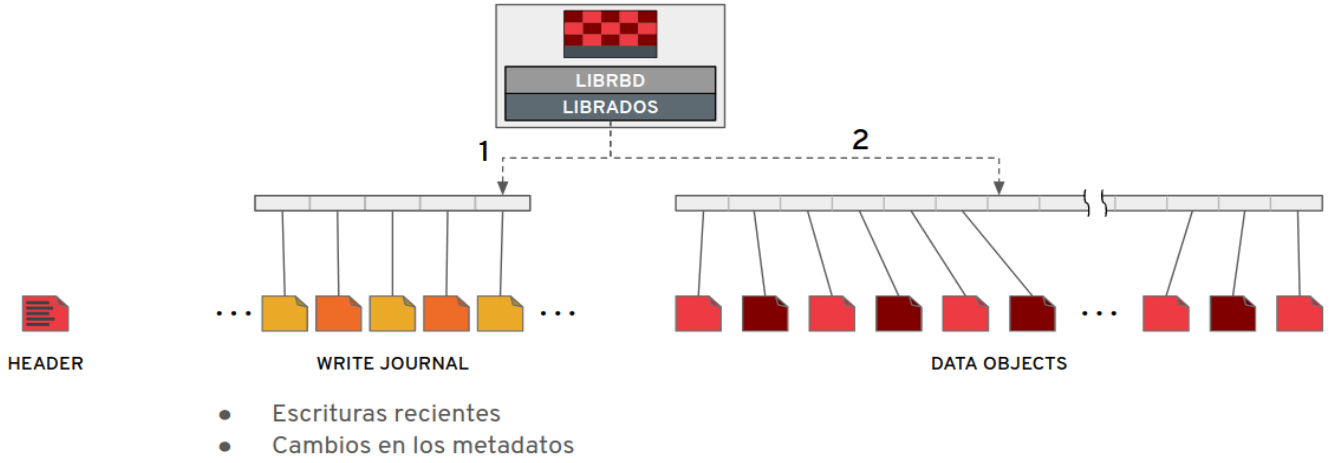


Figura 12. Operaciones en Journal y en Objetos de Datos

Entre sus otras características, RBD puede generar copias espejo de las imágenes en otro cluster Ceph de manera asíncrona, como protección ante fallas, backups o alta disponibilidad. Además, puede asignar cuotas a los usuarios que gestionan las imágenes, proveer aislamiento en el espacio de nombres, importar o exportar imágenes a otro tipo de almacenamiento, entre otras.

3.6. Monitoreo

Ceph tiene integrado un dashboard de monitoreo del cluster, que permite visualizar en tiempo real las métricas más importantes de su funcionamiento, performance, carga y estado general de todos los componentes. (Figura 13)

El módulo de monitoreo también permite gestionar el cluster, y realizar las operaciones más comunes a través de la interfaz web. (Figura 14)

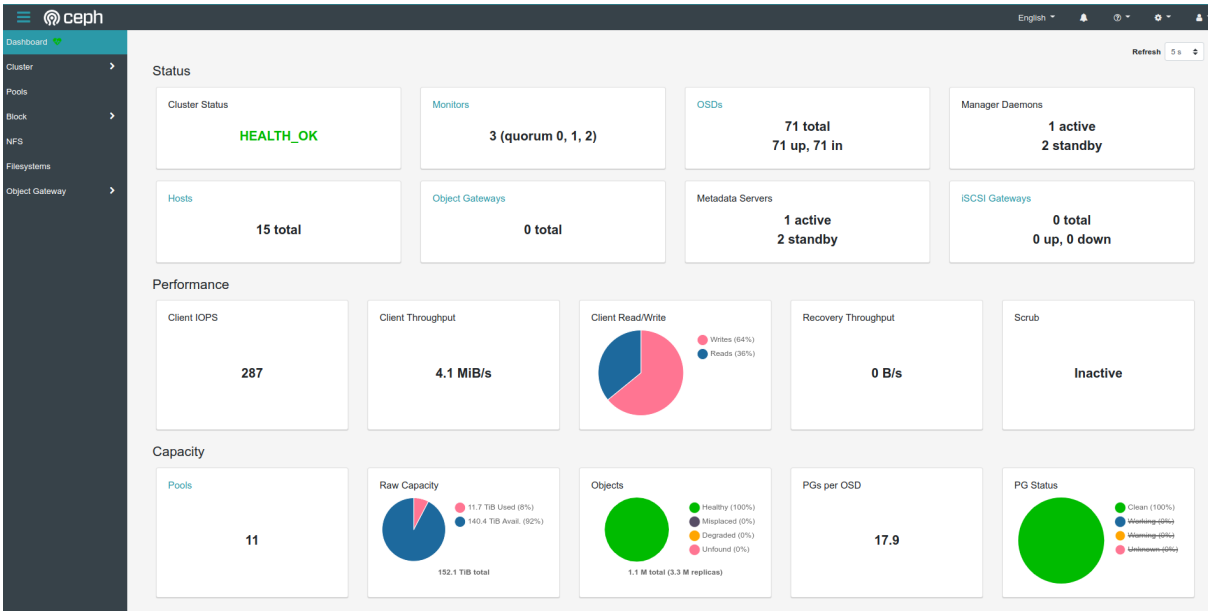


Figura 13. Monitoreo del estado del cluster

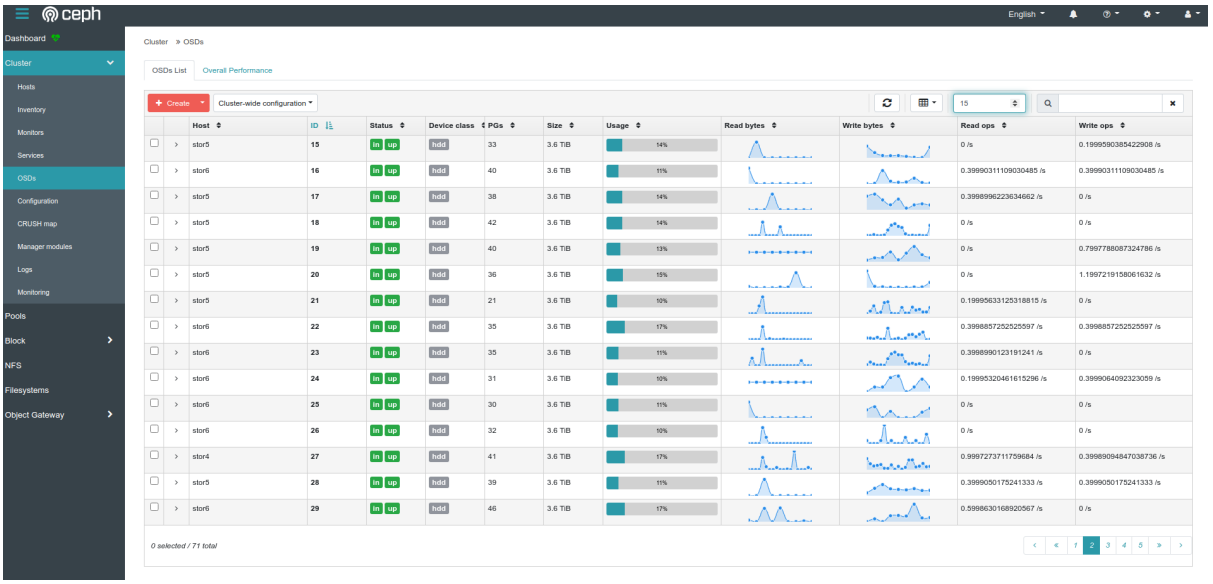


Figura 14. Monitoreo y configuración de los OSDs

4. METODOLOGÍA de ANÁLISIS

En el análisis de la performance del sistema de almacenamiento distribuido se utilizó una metodología bottom-up, comenzando las pruebas por los dispositivos de bloque disponibles (HDD y SSD), luego en los OSDs y continuando con los demás niveles de Ceph: RADOS, pools y RBD.

Se buscó establecer cuál es la máxima performance que puede obtenerse en el acceso al almacenamiento, pues a medida que los clientes aumentan y cambian sus patrones de acceso, y que el volumen de datos se incrementa, es necesario planificar cómo se va a mantener esa performance.

Además, se analizó la red de datos, pues es un factor decisivo a la hora de determinar cómo el throughput máximo que puede transmitir y la latencia introducida afectan el desempeño general del cluster.

Con estos datos, se establece una línea de rendimiento esperado, a modo de SLA (*Service Level Agreement, Acuerdo de Nivel de Servicio*) para el cluster, que luego se utiliza como referencia en el monitoreo continuo y actividades de troubleshooting.

Las operaciones que más afectan la performance son las escrituras, pues las lecturas los clientes las realizan sobre la copia primaria del objeto, en el OSD primario. En cambio, las escrituras afectan a todo el *acting set* del Placement Group (Placement Group Concepts, 2020), involucrando a todos los OSDs de acuerdo al número de réplicas configuradas.

Aunque en los clientes que intercambian datos con el sistema de almacenamiento distribuido los patrones de E/S pueden ser secuenciales, la E/S de cada cliente se distribuye en distintos OSDs que está atendiendo a la vez requerimientos de otros usuarios, y todas las operaciones concurrentes terminan generando en el OSD un patrón de acceso aleatorio en los discos afectados.

4.1. Métricas

Las métricas que se utilizan en la industria para caracterizar dispositivos o sistemas de almacenamiento son 3: throughput, latencia y IOPS.

El **throughput** o bandwidth registra cuántos MB/segundo es capaz de transferir el dispositivo. Esta métrica es usada en patrones de acceso secuenciales.

IOPS (*Input/Output Operations Per Second*) indica cuántas operaciones de Entrada/Salida pueden hacerse por segundo. Usada normalmente para medir accesos aleatorios, esta métrica no tiene sentido si no va acompañada de la latencia. No obstante,

es muy utilizada por los proveedores de servicios en la nube (Amazon EBS Volume Types, 2020) para tipificar sus distintos volúmenes de almacenamiento.

Una operación de E/S es una solicitud única de lectura/escritura que se envía al medio de almacenamiento y que tiene asociado un tamaño de datos. Si multiplicamos el número de IOPS por el tamaño de la solicitud, tendremos un número aproximado de throughput, o ancho de banda.

La **latencia**, usualmente medida en micro o milisegundos, refiere a la cantidad de tiempo que le toma al dispositivo completar la lectura o escritura de un bloque. Se registra normalmente la latencia promedio, aunque el valor de latencia máxima (*tail latency*) es importante porque la operación más lenta de un conjunto afecta la respuesta general.

El valor inicial de latencia se obtiene con un único proceso/thread y una cola (*queue-depth)* con profundidad de 1, lo que significa que una operación de E/S no se inicia hasta que no se completa la anterior. En este caso, IOPS=1/latencia y este número no escala aumentando los servidores, discos, procesos/threads. Como menciona Vitaliy Filippov (2020): "*Single-threaded IOPS and latency numbers only depend on how fast a single daemon is. Why is it important? It 's important because some of the applications can't use queue depth greater than 1 because their task isn't parallelizable. A notable example is any ACID DBMS because all of them write their WALs sequentially with fsync()s."*

A medida que aumenta el valor de *queue-depth* y con él las operaciones lanzadas de manera paralela, los IOPS también van a aumentar hasta los límites del dispositivo; y la latencia también va a ser mayor, porque recién se considera completa la tarea cuando se terminan todas las operaciones del conjunto. De acuerdo con distintas fuentes (Howard, 2015), queue-depth = 32 parecería ser el valor óptimo. Este valor debe verificarse con las pruebas a realizarse en los discos del cluster.

4.2. Herramientas utilizadas

Dado que no existe una herramienta unificada para realizar los testeos, en cada nivel se emplearán las herramientas comúnmente utilizadas para benchmarking:

- Para medir el máximo ancho de banda disponible en la red de datos, se utilizará **iperf3 (**Darabseh, 2015) (Dugan,2010)

- Para medir latencia, throughput y IOPS se utilizará **fio** (FIO, 2020) , específicamente diseñada por Jens Axboe para testear el subsistema de E/S y los schedulers de Linux. El uso de esta herramienta garantiza la independencia respecto de la plataforma, resultados reproducibles y facilita la comparación con otros tipos de almacenamiento. La herramienta **dd** también es muy utilizada para medir tiempos de respuesta de los dispositivos, pero no provee información de lectura y ejecuta en un

único testeo single thread y secuencial. Por eso, aunque se incluyan sus valores en algún testeo (sobre todo en la sección de benchmark de discos) se tomarán únicamente como referencia inicial.

- Para realizar las mediciones en los restantes niveles, se utilizarán las herramientas provistas por el mismo Ceph: **osd tell, rados bench y rbd bench.**

En cada capítulo, antes de utilizar la herramienta correspondiente, se introducirá una breve descripción de los principales parámetros con que se configura su comportamiento.

4.3. Unas palabras sobre benchmarking

Los testeos realizados sobre el hardware (discos, red) y sobre las diferentes capas de Ceph (OSDs, RADOS, RBD) van a arrojar resultados que difícilmente puedan ser los valores que uno obtenga en un sistema de almacenamiento distribuido en *producción,* con cargas de trabajo reales y patrones de acceso tan dispares como dispares son los clientes que lo utilizan.

Es muy difícil, sino imposible, lograr reproducir en un ambiente de testeo las cargas de trabajo de un sistema real, con todas sus variaciones. No obstante, sí podemos decir que si bien una métrica obtenida en un cluster de prueba tal vez no pueda alcanzarse en producción, un valor medido en un cluster en producción nunca va a ser *mejor* que uno obtenido en el de pruebas.

Así, los valores de las diferentes pruebas de benchmarking a realizar funcionan como una suerte de “línea base” o “máximo ideal” sobre las capacidades de throughput, latencia o IOPS del sistema testeado y pueden referenciarse en actividades de monitoreo.

4.4. TESTBED

Se describe a continuación el “banco de pruebas” sobre el que se realizará las mediciones, ofreciendo una sencilla descripción del hardware involucrado, la red de datos y la configuración básica realizada en el sistema de almacenamiento distribuido.

4.4.1. Servidores

El cluster sobre el que se va a realizar las mediciones (*testbed*) está conformado por 13 servidores físicos y 5 virtualizados, con las especificaciones detalladas en la Tabla 1.

CPU - Memoria - Red	Discos	Rol - Servicios
8 Intel Xeon CPU E5620 8 GB RAM 2 x 1Gbps Ethernet	1 x 500 GB HDD SAS 7.2k 1 x 1 TB HDD SAS 7,2k	MON MANAGER MDS (Active)
Virtualizado. 2 VCPUs, 4GB RAM	1 x 100 GB	MON MANAGER (StandBy)
Virtualizado 2 VCPUs, 4GB RAM	1 x 100 GB	MON MANAGER (StandBy)
Virtualizado 2 VCPUs, 2 GB RAM	1 x 10 GB	MDS (StandBy)
Virtualizado 2 VCPUs, 2 GB RAM	1 x 10 GB	MDS (StandBy)
16 Intel Xeon E5620 48 GB RAM 4 x 1 Gbps Ethernet	4 x 464 GB HDD SAS 7.2k	OSD
16 Intel Xeon E5520 36 GB RAM 4 x 1 Gbps Ethernet	4 x 464 GB HDD SAS 7.2k	OSD
16 Intel Xeon E5-2650 32 GB RAM 4 x 1 Gbps Ethernet	4 x 465 GB HDD SAS 7.2k	OSD
16 Intel Xeon Silver 4110 32 GB RAM 2 x 10 Gbps SFP+	1 x 1 TB HDD SATA 7,2k 7 x 4 TB HDD SATA 7.2k	OSD
16 Intel Xeon Silver 4110 32 GB RAM 2 x 10 Gbps SFP+	1 x 1 TB HDD SATA 7,2k 7 x 4 TB HDD SATA 7.2k	OSD
16 Intel Xeon Silver 4110 32 GB RAM 2 x 10 Gbps SFP+	1 x 1 TB HDD SATA 7,2k 7 x 4 TB HDD SATA 7.2k	OSD
16 Intel Xeon Silver 4208 64 GB RAM 2 x 10 Gbps SFP+	1 x 1 TB HDD SATA 7.2k 7 x 2 TB SSD SATA	OSD
16 Intel Xeon Silver 4208 64 GB RAM 2 x 10 Gbps SFP+	1 x 1 TB HDD SATA 7.2k 7 x 2 TB SSD SATA	OSD
16 Intel Xeon Silver 4208 64 GB RAM 2 x 10 Gbps SFP+	1 x 1 TB HDD SATA 7.2k 7 x 2 TB SSD SATA	OSD

16 Intel Xeon Silver 4208 64 GB RAM 2 x 10 Gbps SFP+	1 x 1 TB HDD SATA 7.2k 7 x 2 TB SSD SATA	OSD
16 Intel Xeon Silver 4208 64 GB RAM 2 x 10 Gbps SFP+	1 x 1 TB HDD SATA 7.2k 7 x 2 TB SSD SATA	OSD
16 Intel Xeon Silver 4208 64 GB RAM 2 x 10 Gbps SFP+	1 x 1 TB HDD SATA 7.2k 7 x 2 TB SSD SATA	OSD
Virtualizado 2 VCPUs, 2 GB RAM	-	Administración del Cluster.

Tabla 1. Especificaciones de hardware de los servidores

4.4.2. Sistema Operativo

En todos los nodos se instaló Debian 10 (Buster), con kernel 4.19.0-12-amd64, actualizados al 20/11/2020. Las recomendaciones presentes en la bibliografía (Fisk, 2019) (Hackett, 2019) respecto de la configuración del kernel, confluyen en recomendar estos valores para los nodos del cluster:

```
kernel.pid_max = 4194303
kernel.threads_max = 2097152
vm.max_map_count = 524288
vm.min_free_kbytes = 1048576
vm.vfs_cache_pressure = 10
vm.zone_reclaim_mode = 0
vm.dirty_ratio = 80
vm.dirty_background_ratio = 80
```

Una explicación detallada de cada uno de estos parámetros puede encontrarse en la documentación oficial del kernel de linux (Kernel, 2020).

4.4.3. Red de Datos

Los servidores están conectados a través de switches Huawei S6720-30C-EI-24S-AC (interfaces de 10 Gbps con módulos SFP+) y switches Cisco WS-C2960XR-24TD-I (interfaces de 1 Gbps).

El tráfico entre los nodos del cluster puede dividirse en 2: el tráfico que Ceph intercambia con los clientes y el tráfico interno del cluster. En algunos momentos, el tráfico interno provocado por el rebalanceo de *Placement Groups* en los OSDs, o por la replicación (*backfilling)* de los datos frente a la caída o el ingreso de un OSD, puede afectar el tráfico de los usuarios, aumentando la latencia. Por eso, mucha de la documentación técnica de Ceph recomienda instalar dos redes físicamente separadas para uno y otro tráfico.

No obstante, en la instalación del cluster sobre el cual se desarrolla esta tesis se optó por conectar las múltiples interfaces de los servidores (ya sea GigaEthernet o TenGigabitEthernet) en modo *bonding*, contra interfaces PortChannel o LAG (*Link Aggregation Group*) de acuerdo al switch sobre el que se conectaba el server, configuradas con protocolo LACP.

4.4.4. Ceph

La versión de Ceph utilizada es "*octopus*", lanzada en marzo de 2020, versión 15.2.5. (Figura 15)

ACTIVE STABLE RELEASES

Version	Initial release	Latest	End of life (estimated)
octopus	Mar 2020	15.2.5	2022-06-01
nautilus	Mar 2019	14.2.13	2021-06-01

Figura 15. Versiones estables de Ceph (Nov/2020)

Está instalada en todos los nodos que conforman el cluster de almacenamiento (Figura 16)

Hostname	Services	Version
ceph-mds3	mds.mds3	15.2.5
mds2.ceph.unnoba.edu.ar	mds.mds2	15.2.5
mon1	mds.mds1, mgr.mon1, mon.mon1	15.2.5
mon2.ceph.unnoba.edu.ar	mgr.mon2, mon.mon2	15.2.5
mon3.ceph.unnoba.edu.ar	mgr.mon3, mon.mon3	15.2.5
stor1	osd.0, osd.1, osd.2	15.2.5
stor10	osd.51, osd.52, osd.53, osd.54, osd.55, osd.56, osd.57	15.2.5
stor11	osd.58, osd.59, osd.60, osd.61, osd.62, osd.63, osd.64	15.2.5
stor12	osd.65, osd.66, osd.67, osd.68, osd.69, osd.70	15.2.5
stor2	osd.3, osd.4, osd.5	15.2.5
stor3	osd.6, osd.7, osd.8	15.2.5
stor4	osd.10, osd.11, osd.12, osd.13, osd.14, osd.27, osd.9	15.2.5
stor5	osd.15, osd.17, osd.18, osd.19, osd.20, osd.21, osd.28	15.2.5
stor6	osd.16, osd.22, osd.23, osd.24, osd.25, osd.26, osd.29	15.2.5
stor7	osd.30, osd.31, osd.32, osd.33, osd.34, osd.35, osd.42	15.2.5
stor8	osd.36, osd.37, osd.38, osd.39, osd.40, osd.41, osd.43	15.2.5
stor9	osd.44, osd.45, osd.46, osd.47, osd.48, osd.49, osd.50	15.2.5

Figura 16. Versión de Ceph instalada en todos los nodos.

4.4.5. Mapa y reglas CRUSH

Como mencionamos en el capítulo anterior, el mapa CRUSH consiste en una jerarquía que describe la topología física del cluster y un conjunto de reglas que definen la política de ubicación de los datos. Su estructura jerárquica tiene por hojas los OSDs, y las ramas son otros dispositivos o agrupamientos: hosts, racks, rows (filas de racks), datacenters, etc… Las reglas describen cómo se ubican las réplicas en términos de esa jerarquía: *tres réplicas en diferentes racks.*

La estrategia aquí es poder identificar los puntos de falla del hardware y configurar el mapa de manera que una falla en un equipo, en una PDU (*Power Distribution Unit*, una "zapatilla inteligente") o en una UPS, no afecte demasiados nodos del cluster. Para eso, se configuran las reglas de manera que la replicación de datos tenga en cuenta esta organización, y la redundancia se exprese en esos términos.

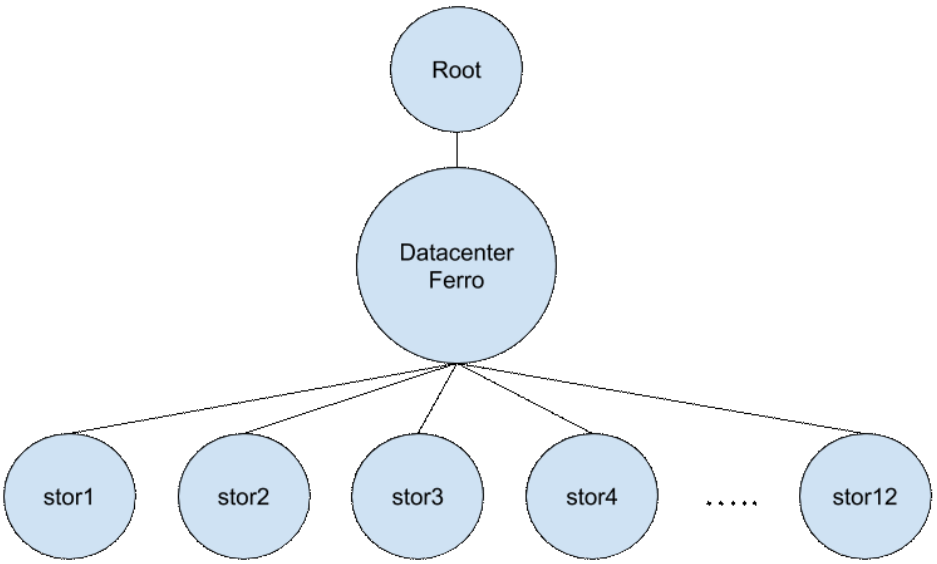


Figura 17. Mapa CRUSH del Cluster

El **mapa** CRUSH (Figura 17) implementado en el cluster de este trabajo es sencillo: todos los nodos se ubican bajo una agrupación llamada "Datacenter Ferro", donde *stor1* a *stor12* son los equipos destinados a almacenamiento (OSDs) descriptos en la tabla del punto 4.4.1.

Las **reglas** de CRUSH (Weil, 2006. Section 3.2) definidas para el presente trabajo son 4: una para replicación, otra para erasure-code, otra para que la replicación se efectúe únicamente sobre discos HDD y otra para que use únicamente discos SSD (Ver #fig_445rules), de manera de poder comparar la performance entre una y otra configuración del almacenamiento.

```
# ceph osd crush rule ls
replicated_rule
```

```
erasure-code
hddpool
ssdpool
```

Figura 18. Listado de reglas definidas en el cluster

4.4.6. Pools

Para poder realizar las pruebas de performance sobre el cluster, se definen inicialmente 3 pools. (Figura 19)

Nombre	Regla CRUSH	Número de PGs
testHDD	hddpool: replicated, n=3, sin compresión	32 (auto)
testSSD	ssdpool: replicated, n=3, sin compresión	32 (auto)
testEC	erasure-code, k=5, m=3, sobre discos hdd, sin compresión	32 (auto)

Figura 19. Pools definidos en el cluster

La creación de los pools puede hacerse vía intérprete de línea de comandos, o por la interfaz web del Ceph Manager, como se muestra en la Figura 20

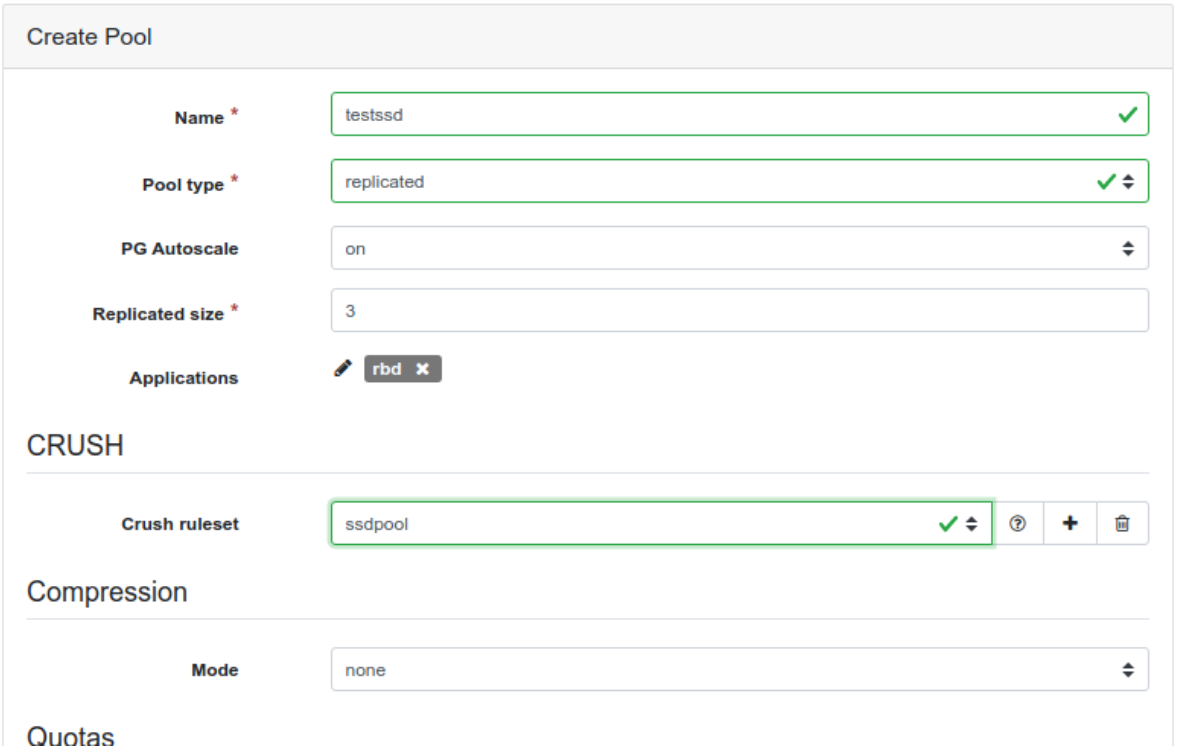


Figura 20. Interfaz web para creación de pool replicado.

Para la creación del pool de testeo de erasure-code (Figura 21), se crea también un perfil que establece que cada objeto se divide en 5 fragmentos (k=5) y que se deben generar 3 fragmentos más de redundancia/paridad (m=3). Como ya se dijo en el punto anterior, erasure-code es un esquema que funciona bien para datos de gran tamaño y con pocos cambios, de manera que se van a utilizar los discos HDD en las pruebas de performance. Por esto, se configura que la clase de dispositivo (*crush-device-class*) sea hdd. Además, se establece que el dominio de falla sea de OSD. Si hubiéramos elegido como dominio de falla el *host*, entonces Ceph necesitaría 8 servidores distintos con procesos OSD sirviendo discos HDD para ubicar los fragmentos de objetos.

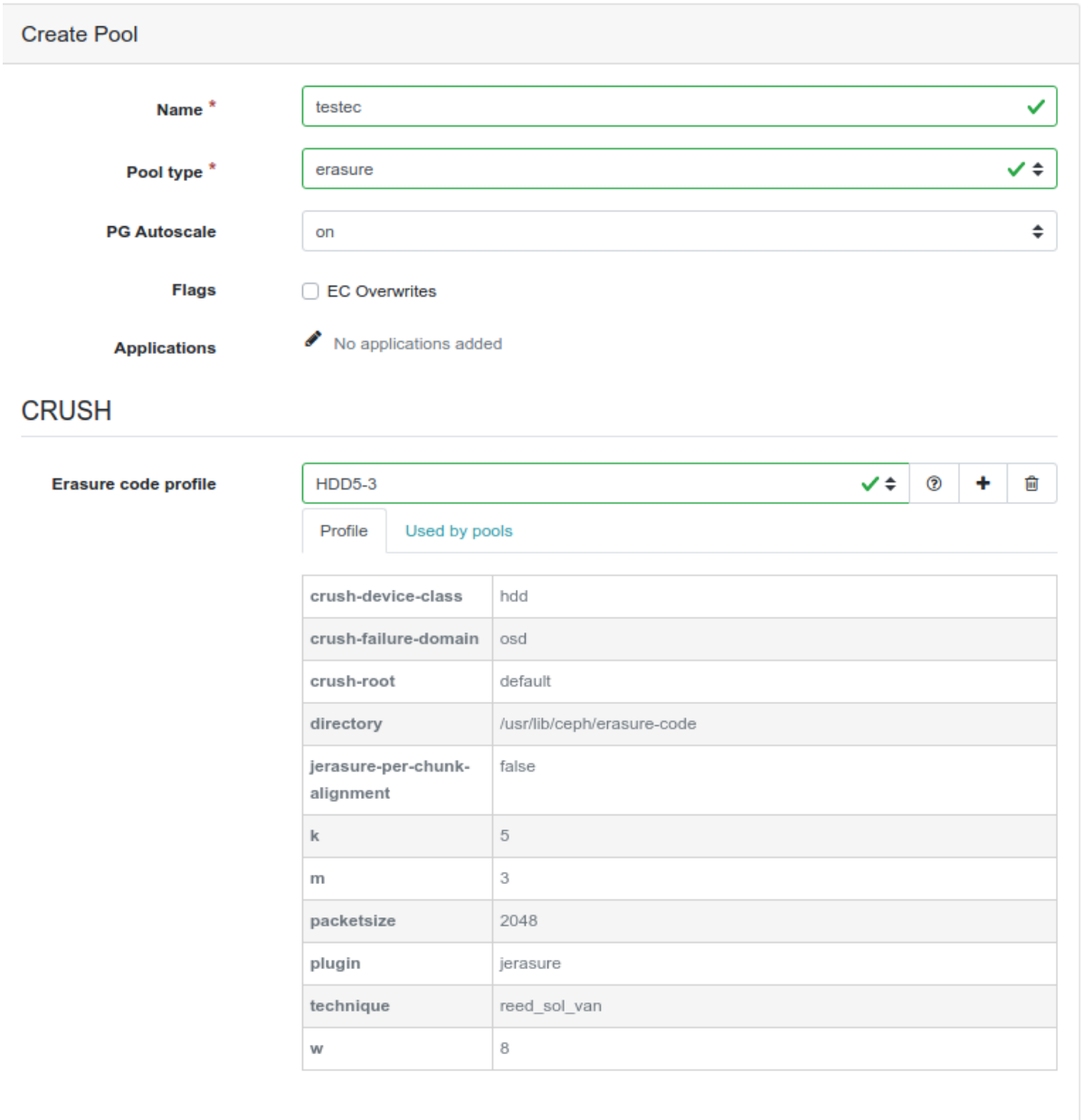


Figura 21. Interfaz web para creación de pools erasure-code

4.4.7. Placement Groups

Un *placement group* agrega objetos de un pool porque calcular la ubicación de datos y metadatos con una granularidad a nivel de objetos individuales es computacionalmente muy costoso. Los clientes Ceph calculan en qué PG un objeto se almacenó, aplicando un hash al nombre del objeto y realizando una operación basada en el número de PGs definidos en el pool y el ID del pool.

La instalación actual se hizo con el módulo *pg_autoscale* activado. Esto hace que el número de *placement groups* sea calculado y escalado automáticamente, de acuerdo a la cantidad de objetos que el pool contiene, y al número de OSDs disponibles. (Placement Groups, 2020).

5. OPTIMIZACIÓN de la RED de DATOS

Optimizar una red de datos para obtener un throughput y una latencia óptimas es un proceso complejo, con muchos factores a considerar. Estos factores incluyen las características y capacidades de la NIC (*Network Interface Card*), las funcionalidades del driver, la memoria RAM disponible, el número de CPUs, las configuraciones realizadas sobre el kernel del sistema operativo instalado y las cargas de trabajo que la interfaz de red debe manejar. Como menciona Jamie Bainbridge en su trabajo “Red Hat Enterprise Linux Network Performance Tuning Guide” (Bainbridge, 2014): no hay una configuración genérica que pueda ser aplicada en cualquier sistema, pues estos factores son siempre diferentes.

5.1. Las interfaces de red y sus drivers

La NIC y su driver comparten un buffer de recepción - *RX ring buffer* - que está organizado como un anillo circular, donde un overflow implica que los nuevos datos sobreescriben los datos más antiguos. En este buffer se almacenan los paquetes entrantes hasta que puedan ser procesados por el driver del dispositivo.

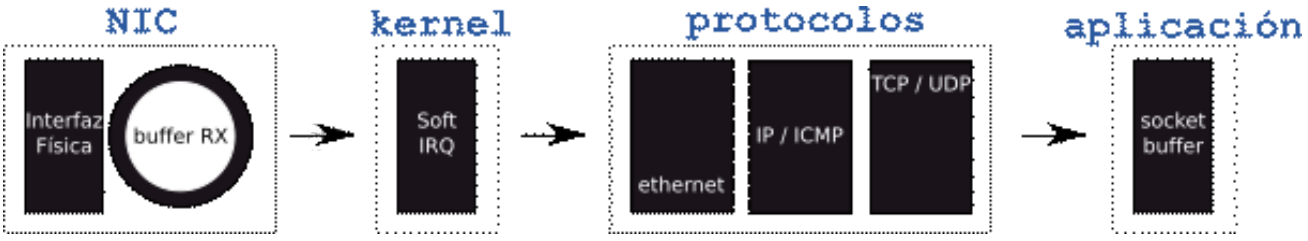


Figura 22. Flujo de datos que ingresan en la interfaz de red.

Cuando una NIC recibe datos, copia los datos al buffer del kernel a través de DMA y le notifica levantando una interrupción, que es atendida por el *handler.* Estas interrupciones, costosas a nivel de CPU porque interrumpen la ejecución de otro proceso y no pueden ser interrumpidas, hacen un mínimo trabajo y dejan el procesamiento del paquete que ingresó a una interrupción por software (*SoftIRQ*) cuyo proceso ingresa en la cola que gestiona el planificador de CPU. (Ver Figura 22)

Estas interrupciones pueden verse en **/proc/interrupts** (Figura 23), donde cada interfaz de red tiene asociado un interruptor en el vector de interrupciones de cada CPU disponible.

```
# cat /proc/interrupts | grep eno
           CPU0       CPU1       CPU2       CPU3       CPU4       CPU5       CPU6       CPU7
[...]
  44:          0          0          0          0          0   13872797          0          0   PCI-MSI 5767168-edge
eno1-0
  45:          0          0          0          0          0          0   22777544          0   PCI-MSI 5767169-edge
eno1-1
  46:          0          0          0          0          0          0          0   22455433   PCI-MSI 5767170-edge
eno1-2
  47:   19660699          0          0          0          0          0          0          0   PCI-MSI 5767171-edge
eno1-3
```

```
  48:          0   22332683          0          0          0          0          0          0   PCI-MSI 5767172-edge
eno1-4
  49:          0          0   25663487          0          0          0          0          0   PCI-MSI 5767173-edge
eno1-5
  50:          0          0          0   24539797          0          0          0          0   PCI-MSI 5767174-edge
eno1-6
  51:          0          0          0          0   17746035          0          0          0   PCI-MSI 5767175-edge
eno1-7
  53:          0          0          0          0          0   10652033          0          0   PCI-MSI 5769216-edge
eno2-0
  54:          0          0          0          0          0          0   37399115          0   PCI-MSI 5769217-edge
eno2-1
  55:          0          0          0          0          0          0          0   47986486   PCI-MSI 5769218-edge
eno2-2
  56:   42755461          0          0          0          0          0          0          0   PCI-MSI 5769219-edge
eno2-3
  57:          0   73869355          0          0          0          0          0          0   PCI-MSI 5769220-edge
eno2-4
  58:          0          0   29271710          0          0          0          0          0   PCI-MSI 5769221-edge
eno2-5
  59:          0          0          0   15706073          0          0          0          0   PCI-MSI 5769222-edge
eno2-6
  60:          0          0          0          0   42444394          0          0          0   PCI-MSI 5769223-edge
eno2-7
```

Figura 23. Cada interfaz tiene una interrupción asociada a cada CPU

El driver vacía el buffer de recepción a través de SoftIRQs, que colocan los paquetes entrantes en una estructura de datos del kernel llamada *sk_buff* o *skb* para luego ser procesada por la pila de protocolos.

Las SoftIRQ (ver Figura 24) son rutinas del kernel cuya ejecución se planifica cuando no interrumpen la ejecución de otros procesos. Estas rutinas ejecutan en la forma de procesos llamados *ksoftirqd/<número de cpu>* y pueden verse con el comando *ps*:

```
# ps -All | grep ksoftirqd
       1 S     0       9       2  0  80   0 -     0 -      ?         00:00:17 ksoftirqd/0
       1 S     0      16       2  0  80   0 -     0 -      ?         00:00:16 ksoftirqd/1
       1 S     0      21       2  0  80   0 -     0 -      ?         00:00:10 ksoftirqd/2
       1 S     0      26       2  0  80   0 -     0 -      ?         00:00:10 ksoftirqd/3
       1 S     0      31       2  0  80   0 -     0 -      ?         00:00:23 ksoftirqd/4
       1 S     0      36       2  0  80   0 -     0 -      ?         00:00:08 ksoftirqd/5
       1 S     0      41       2  0  80   0 -     0 -      ?         00:00:09 ksoftirqd/6
       1 S     0      46       2  0  80   0 -     0 -      ?         00:00:13 ksoftirqd/7
```

Figura 24: Procesos SoftIRQ corriendo en el server

Las softIRQs asociadas al buffer de recepción (ver Figura 25) pueden monitorearse en */proc/softirq*:

```
#cat /proc/softirqs | grep RX
          CPU0       CPU1       CPU2       CPU3       CPU4       CPU5       CPU6       CPU7
NET_RX:   64771376   98561586   57267446   42749521   62559836   26827224 62513112 72835328
```

Figura 25. Monitoreo de ejecución de las rutinas softIRQ

Estas rutinas van procesando los paquetes y los entregan a la pila de protocolos, que finalmente ubican el *payload* en el buffer de la aplicación, a la espera de que éstas lo procesen con *recv()*. Las softIRQ continúan haciendo un *NAPI Polling*. NAPI, o New API, fue escrita para hacer el procesamiento de paquetes más eficiente, porque permite al driver entrar en un modo de polling en vez de ser invocado a través de interrupciones cada vez que un paquete ingresa en la NIC.

En circunstancias normales, cuando llegan datos se levanta una interrupción inicial, seguida de un manejador softIRQ que sondea el buffer usando estas rutinas NAPI. Este proceso tiene asociado un *budget* que indica cuánto tiempo de CPU puede consumir en la

tarea de procesar paquetes ingresantes, para evitar que monopolice la CPU. Cuando se termina el tiempo asignado, el proceso finaliza y la operación arranca nuevamente si es necesario.

5.2. Monitoreo, análisis y ajuste de parámetros de red en el kernel

Los kernels de los sistemas operativos traen una configuración genérica que es necesario revisar y ajustar, de acuerdo al hardware que posee y a la funcionalidad que el equipo tiene en el cluster.

Cuando el kernel no puede vaciar el buffer de recepción de la NIC con suficiente velocidad, los nuevos paquetes sobreescriben los anteriores. A esta situación, las antiguas herramientas de la familia *net-tools* (ifconfig, netstat, etc…) la llamaban *overrun*. (Figura 26)

```
#ifconfig ens2f0
ens2f0: flags=6211<UP,BROADCAST,RUNNING,SLAVE,MULTICAST>  mtu 1500
        ether f2:2d:2d:80:f3:2c  txqueuelen 1000  (Ethernet)
        RX packets 9017440607  bytes 9017346605345 (8.2 TiB)
        RX errors 750466  dropped 0  overruns 750466  frame 0
        TX packets 9794464920  bytes 9065231576625 (8.2 TiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
        device interrupt 26  memory 0xe3800000-e3ffffff
```

Figura 26. Estadísticas de la interfaz de red

Los buffers de recepción tienen un tamaño fijo y cuando llegan más paquetes de los que la NIC puede almacenar, descarta los sobrantes, incrementando el contador de *discards*. (Ver Figura 27)

Cualquiera de estos errores implica una baja en la performance en el throughput de la red.

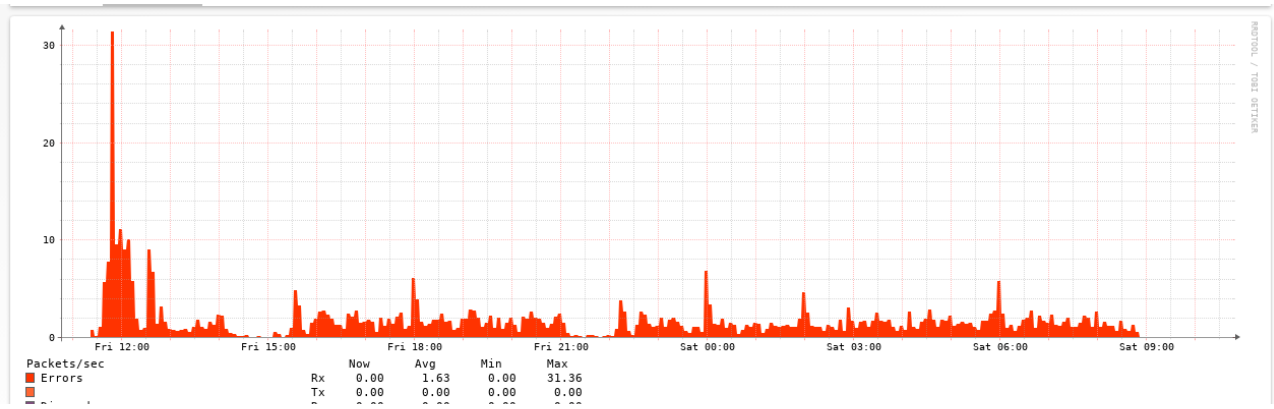


Figura 27. Monitoreo de paquetes descartados en la interfaz

La mayoría de los reportes técnicos analizados recomienda el uso de la herramienta *ethtool (*Figura 28*)* para poder consultar las estadísticas de las interfaces de red. La información que brinda esta herramienta es mucho más completa y es necesario filtrarla:

```
#ethtool --statistics ens2f0 | grep rx_discards
     rx_discards: 750466
```

Figura 28. Informe de paquetes descartados

¿Cuál puede ser la causa de estos errores? El equipo sobre el que se realizó el análisis anterior es uno de los servidores destinado a almacenamiento (OSDs), con 2 interfaces de 10 Gbps. Y la salida fue similar en los demás equipos con interfaces del mismo tipo. Pero no así en los servers con interfaces de 1Gbps.

Las razones más habituales pueden ser:

- Manejo pobre de las interrupciones
- El kernel no está pudiendo sacar los datos con suficiente velocidad
- Control de flujo de ethernet (*Pause frames*)
- Desborde del buffer de recepción

Si el manejo de interrupciones no está balanceado correctamente entre todas las CPU del equipo, puede darse la situación de la #fig_desbalance:

```
# cat /proc/interrupts | grep eno
           CPU0       CPU1       CPU2       CPU3       CPU4       CPU5       CPU6       CPU7
[...]
  44:   13872797          0          0          0          0          0          0          0   PCI-MSI 5767168-edge
eno1-0
  45:   22777544          0          0          0          0          0          0          0   PCI-MSI 5767169-edge
eno1-1
  46:   22455433          0          0          0          0          0          0          0   PCI-MSI 5767170-edge
eno1-2
  47:   19660699          0          0          0          0          0          0          0   PCI-MSI 5767171-edge
eno1-3
  48:   22332683          0          0          0          0          0          0          0   PCI-MSI 5767172-edge
eno1-4
  49:   25663487          0          0          0          0          0          0          0   PCI-MSI 5767173-edge
eno1-5
  50:   24539797          0          0          0          0          0          0          0   PCI-MSI 5767174-edge
eno1-6
  51:   17746035          0          0          0          0          0          0          0   PCI-MSI 5767175-edge
eno1-7
 [...]
```

Figura 29. Desbalanceo en la asignación de las interrupciones

Esto indica que el servicio **irqbalance** no está activo o no está funcionando correctamente. La carga de las interrupciones debería estar repartida en todos los cores disponibles (comparar con la tabla anterior de interrupciones). irqbalance es un servicio que puede balancear automáticamente las interrupciones entre los cores de las CPU, en base a las condiciones del sistema medidas en tiempo real.

No es el caso de nuestro equipo, entonces se continúa investigando qué provoca los errores encontrados.

Si SoftIRQ no ejecuta el tiempo suficiente, la tasa de paquetes que ingresan excede la capacidad del kernel para vaciar el buffer, que termina haciendo overflow y las tramas se pierden. Este tiempo se especifica en **net.core.netdev_budget** (Figura 30), que por defecto tiene un valor de 300, o sea que la rutina va a procesar 300 mensajes antes de dejar la CPU.

```
# sysctl net.core.netdev_budget
net.core.netdev_budget = 300
```

Figura 30. Tiempo por defecto para ejecución de rutinas SoftIRQ

Para ver si softIRQ está ejecutando el tiempo suficiente, debe analizarse la tercera columna de **/proc/net/softnet_stat**, llamada “time squeeze” (Figura 31), que representa el número de veces que la rutina terminó porque se le venció el tiempo, pero le quedaron paquetes en el buffer por procesar. (SoftNetStats, 2020)

```
# cat /proc/net/softnet_stat
00575188 00000000 000046ad 00000000 00000000 00000000 00000000 00000000 ...
26fb897b 00000000 002874c6 00000000 00000000 00000000 00000000 00000000 ...
25cc3479 00000000 00211e50 00000000 00000000 00000000 00000000 00000000 ...
2932702d 00000000 00271c2d 00000000 00000000 00000000 00000000 00000000 ...
004c229d 00000000 00003f02 00000000 00000000 00000000 00000000 00000000 ...
2a86a0f2 00000000 002722e1 00000000 00000000 00000000 00000000 00000000 ...
45096c52 00000000 00460092 00000000 00000000 00000000 00000000 00000000 ...
3bfb728c 00000000 00315de3 00000000 00000000 00000000 00000000 00000000 ...
{...}
```

Figura 31. Contador de veces en que SoftIRQ no completó el trabajo

Cada fila representa una CPU y si la tercera columna tiene valores altos, significa que **softIRQ no ejecuta el tiempo suficiente para vaciar el buffer de recepción de la NIC**, y necesita incrementarse el valor del budget con el comando de la Figura 32:

```
# sysctl -w net.core.netdev_budget=600
```

Figura 32. Reconfiguración del tiempo límite para ejecución de SoftIRQ

Una vez configurado este valor y agregado en /etc/sysctl.conf para que permanezca luego del reinicio, se debe reiniciar el server, dejarlo que trabaje algunos días y verificar nuevamente ambos valores: el de tramas descartadas y el contador de SoftIRQ.

5.2.1. Pause Frames

Otro aspecto a verificar es el control de flujo entre el adaptador de red y el puerto del switch, implementado en un mecanismo llamado “*pause frames”*. El adaptador va a enviar *pause frames* cuando los buffers de RX o TX se llenan. El switch deja de enviar paquetes por algunos milisegundos, tiempo suficiente para que el kernel vacíe los buffers. Los paquetes quedan encolados en la memoria del switch durante la “pausa”.

Este mecanismo debe habilitarse tanto en el switch como en las interfaces de red, y puede verificarse con el comando de la Figura 33

```
# ethtool -a ens1f0
Pause parameters for ens1f0:
Autonegotiate:  off
RX:         on
TX:         on
```

Figura 33. Verificación de Pause Frames

En este caso, el mecanismo está habilitado en la interfaz. Para habilitarlo en el switch, hay que ver el manual del equipo que provee el fabricante. En nuestra topología, el switch Huawei utilizado (Figura 34) reporta las tramas del mecanismo flow-control:

```
Eth-Trunk6 current state : UP
```

```
Line protocol current state : UP
Description:STOR12.CEPH
Switch Port, Link-type : access(configured),
PVID :   15, Hash arithmetic : According to SIP-XOR-DIP,Maximal BW: 20G,
Current BW: 20G, The Maximum Frame Length is 9216
IP Sending Frames' Format is PKTFMT_ETHNT_2, Hardware address is
d0c6-5bb3-1f50
Current system time: 2020-11-08 11:32:33-03:00
Last 300 seconds input rate 6447752 bits/sec, 1017 packets/sec
Last 300 seconds output rate 15120536 bits/sec, 1684 packets/sec
Input:  3472536258 packets, 3107918895711 bytes
 Unicast:                3472102893,  Multicast:                    432306
 Broadcast:                   1005,  Jumbo:                           0
 Discard:                        0,  Pause:                        54
 Frames:                         0

 Total Error:                    0
 CRC:                            0,  Giants:                        0
 Runts:                          0,  DropEvents:                    0
 Alignments:                     0,  Symbols:                       0
 Ignoreds:                       0
```

Figura 34. Estadísticas del puerto del switch conectado a un nodo del cluster

5.2.2. Interrupt Coalescing

Una configuración a verificar es la llamada "*INTERRUPT COALESCING"*, o *moderación de interrupciones,* que indica cuánto tráfico puede recibir una NIC antes de levantar la interrupción avisando al hardware (Interrupt coalescing, 2020*)*. Si la interrupción es muy frecuente, la performance se ve afectada; si tarda mucho, se pisan los datos en el buffer circular de recepción, provocando los *underruns* antes mencionados.

El *modo adaptativo* (Figura 35) modera el *Interrupt Coalescence* inspeccionando el patrón de tráfico y el buffer de recepción del kernel, y estima la configuración óptima en tiempo real. Un valor más alto favorece el ancho de banda por sobre la latencia, y tráficos como VoIP pueden verse afectados. En cambio, la transferencia de archivos se ve favorecida. Una configuración realista permite algunos paquetes en el buffer de la NIC, y algún tiempo de retraso, hasta que se interrumpe el kernel para que los procese.

```
# ethtool -c ens1f1
Coalesce parameters for ens1f1:
Adaptive RX: on  TX: on
stats-block-usecs: 0
sample-interval: 0
pkt-rate-low: 0
pkt-rate-high: 0

rx-usecs: 8
rx-frames: 128
rx-usecs-irq: 0
rx-frames-irq: 0
```

```
tx-usecs: 8
tx-frames: 128
tx-usecs-irq: 0
tx-frames-irq: 0
[...]
```

Figura 35: Configuración de interfaz de red con modo adaptativo

Los valores **-usecs* especifican cuántos microsegundos esperar después de que llegó un paquete y los valores **-frames* cuántos paquetes esperar antes de generar la interrupción, lo que ocurra primero de ambos.

Si bien esto busca maximizar el throughput o la tasa de paquetes, lo hace a expensas de la latencia que puede generar timeouts en protocolos como TCP por eso es bueno monitorear y verificar los valores configurados (Mellanox, 2019).

5.2.3. Colas de procesamiento

Cuando las tramas se sacan del buffer de la NIC, el kernel las ubica en una cola antes de ser procesadas por la pila de protocolos. Hay una cola por núcleo de la CPU, que puede crecer automáticamente hasta el máximo especificado por **netdev_max_backlog**, que por defecto tiene un valor de 1000 (Figura 36).

```
# sysctl net.core.netdev_max_backlog
net.core.netdev_max_backlog = 1000
```

Figura 36. Longitud por defecto de la cola

Si la **segunda** columna de **/proc/net/softnet_stat** (Figura 31) tiene valores distintos de 0, significa que hubo overflow en la cola de esa CPU y que hay que incrementar el valor de netdev_max_backlog.

Si después de haber configurado todos estos valores, en nuestro equipo continuamos teniendo reportes de frames descartados, es necesario analizar y configurar los buffers de recepción del adaptador de red: incrementando el tamaño del buffer de recepción, el kernel tiene más tiempo para procesar paquetes y evitar que se sobreescriban.

La configuración actual se consulta con el comando de la Figura 37.

```
#ethtool -g ens2f0
Ring parameters for ens2f0:
Pre-set maximums:
RX:         4078
RX Mini:    0
RX Jumbo:   0
TX:         4078
Current hardware settings:
RX:         453
```

```
RX Mini:    0
RX Jumbo:   0
TX:         4078
```

Figura 37: Tamaño de buffer de recepción de la NIC.

Acá verificamos que el buffer de recepción tiene un máximo posible de 4078, pero que está configurado en el driver en 453. Esto podría provocar el descarte de paquetes que estamos observando. Re-configuramos con el comando de la Figura 38.

```
# ethtool -G ens2f0 rx 4078
```

Figura 38. Configuración del máximo que permite el driver.

5.2.4. Adapter Offloading

Para reducir la carga de la CPU, algunos adaptadores de red tiene la capacidad de mover la carga de procesamiento a la NIC. Por ejemplo:

- El kernel manda un segmento de 64K TCP y la NIC lo segmenta en pequeños fragmentos del tamaño de la MTU (TSO: TCP Segmentation Offload).
- El cálculo de las tramas enviadas y la verificación del CRC de las tramas recibidas normalmente la hace la placa, no el kernel.

Estas características están habilitadas por defecto (Figura 39) y ,frente a algún problema, es bueno deshabilitarlas para hacer el procedimiento de depuración de errores.

```
# ethtool -k ens1f1
Features for ens1f1:
rx-checksumming: on
tx-checksumming: on
scatter-gather: on
tcp-segmentation-offload: on
tx-tcp-segmentation: on
udp-fragmentation-offload: off
generic-segmentation-offload: on
generic-receive-offload: on
large-receive-offload: off
rx-vlan-offload: on
tx-vlan-offload: on
ntuple-filters: off
receive-hashing: on
[...]
```

Figura 39. Características de Offloading habilitadas por defecto

5.2.5. Jumbo Frames

El tamaño de la trama estándar para IEEE 802.3 es de 1518 bytes (o 1522 bytes, incluyendo en el encabezado de la trama los campos para VLANS).

Como el encabezado ethernet utiliza 18 bytes, deja 1500 como payload máximo:

$$\frac{18}{1500} = \ 1.2\% \ de \ overhead$$

Jumbo Frames es una extensión a Ethernet en la que el payload se incrementa hasta 9000 bytes.

$$\frac{18}{9000} = \ 0.2\% \ de \ overhead$$

La eficiencia de esta configuración se nota cuando se transfieren grandes cantidades de datos contiguos, pero si se transfieren pequeños segmentos no hay gran mejora.

Para activar Jumbo Frames, **todas las interfaces** en **todos los equipos** conectados a un dominio de difusión deben soportar jumbo frames y tener habilitada la extensión del protocolo para permitir tramas con MTU de 9000 bytes.

Se puede chequear el valor de MTU configurado en la interfaz del server con el comando que se muestra en la Figura 40.

```
# ip link show dev ens1f0
4: ens1f0: <BROADCAST,MULTICAST,SLAVE,UP,LOWER_UP> mtu 1500 qdisc
mq master bond0 state UP mode DEFAULT group default qlen 1000
    link/ether 94:40:c9:95:2f:78 brd ff:ff:ff:ff:ff:ff
```

Figura 40. MTU Máximo

En este caso, la extensión de jumbo frames no está habilitada para las pruebas de esta tesis. El trabajo de Meyer y Morrison *Impact of single parameter changes on Ceph Cloud storage performance* (Meyer, 2016) concluye que el impacto de los jumbo frames en un cluster grande es poco significativo porque la carga de la replicación se distribuye y la dependencia del ancho de banda es menor; pero cuando el cluster es chico, el ancho de banda entre los nodos es un factor limitante.

5.2.6. TCP timestamps

Algunos autores (Umrao, 2017) (Fisk, 2019) (Hackett, 2019) recomiendan algunas configuraciones extra cuya incidencia en la performance no es determinante, pero que invitan a verificar en cada instalación, modificar su configuración y volver a probar para comparar resultados.

TCP Timestamps es una extensión de TCP (RFC 1323) que provee un contador de incremento monótono - en linux es el número de milisegundos desde el encendido - que se utiliza para estimar el RTT de una transmisión TCP y calcular con mejor precisión la ventana de TCP y el tamaño de los buffers. También provee una protección contra el reinicio de la secuencia de números que TCP define en su campo *sequence number* de 32 bits. El reinicio de la secuencia haría que el receptor descartara el segmento TCP con un flag RST porque tiene un número anterior al último.

En un enlace de 1Gbps, la secuencia de numeración TCP puede reiniciarse en 17 segundos. En un enlace de 10Gbps, puede suceder en 1.8 segundos. TCP timestamp provee un método alternativo y no-reiniciable para determinar el orden y la antigüedad de un segmento TCP. En Debian 10 viene habilitado por defecto. Verificar con el comando `sysctl net.ipv4.tcp_timestamps`

5.2.7. TCP Selective Acknowledgements

Definidos en la RFC 2018, esta extensión de TCP habilita al receptor a especificar qué bytes se perdieron y cuáles se recibieron, así el transmisor puede enviar sólo los bytes perdidos.

Pero el costo de tiempo de calcular el ACK selectivo es mayor al de reenviar los segmentos perdidos con el comportamiento habitual de TCP. Si el enlace es de alta velocidad, el overhead generado por la retransmisión es inferior al que genera calcular qué datos enviar y cuáles no.

Aquí la bibliografía se divide. Algunos recomiendan dejar tcp_sack apagado, otros recomiendan encenderlo. Por defecto, en Debian 10 viene habilitado. Verificar con `sysctl net.ipv4.tcp_sack`

5.2.8. Buffers de las Aplicaciones

Una vez que el tráfico es recibido por la NIC y procesado por el kernel, pasa por la pila de protocolos ethernet, IP y TCP, y queda a la espera de que la aplicación lo reciba. Si esto no ocurre, queda encolado en un *socket buffer* de la aplicación, que tiene 3 estructuras: una para recepción (sk_rmem_alloc), otra para transmisión (sk_wmem_allow) y otra para skbs fuera de orden (sk_omem_alloc).

sk_rcvbuf es una variable del kernel que contiene el límite en bytes que un socket puede almacenar. Si sk_rmem_alloc > sk_rcvbuf, la pila TCP llama a una rutina que “colapsa” la cola de recepción. Esta rutina es costosa en términos de CPU y si no se logra suficiente espacio, los datos se eliminan de memoria y el paquete se pierde.

La memoria de sockets tiene 3 valores: mínimo, default y máximo, expresados en bytes. En este servidor, vemos (Figura 41) que tiene una memoria entre 4 KB y 4 MB, estableciendo un buffer inicial de 128 KB.

```
#sysctl net.ipv4.tcp_rmem
net.ipv4.tcp_rmem = 4096   131072   4194304
```

Figura 41. Valores por defecto en la memoria de sockets

Si los valores del servidor son inferiores a estos, la bibliografía (Hackett, 2019) recomienda incrementarlos (Figura 42), aunque lograr una configuración óptima puede llevar más de una prueba. (The story of one latency spike, 2015)

```
net.ipv4.tcp_window_scaling = 1
net.ipv4.tcp_rmem = 4096 87380 4194304
net.ipv4.tcp_wmem = 4096 65536 4194304
net.core.rmem_max = 4194304
net.core.wmem_max = 4194304
```

Figura 42. Establecer valores óptimos para buffers

Las dos últimas líneas aplican a los demás protocolos que no son TCP, y como Ceph utiliza la interfaz BSD de sockets en modo SOCK_STREAM, no tienen efecto más que consolidar valores en la configuración.

5.3. Finalmente, benchmarking de la red

Habiendo optimizado todos los servidores que integran el cluster con las configuraciones descriptas, se inician pruebas sobre la red de datos, utilizando la aplicación *iperf3* en modo servidor en un nodo de almacenamiento con interfaces de 10Gbps y en modo cliente en el otro nodo con las mismas características. El RTT obtenido se muestra en la Figura 43:

```
min/avg/max/mdev = 0.139/0.163/0.213/0.021 ms
```

Figura 43. Round trip time obtenido al testear la red.

Se mide el ancho de banda máximo disponible (Figura 44) para una conexión TCP durante 20 segundos, entre ambos servidores.

```
# iperf3 -c stor11.ceph.unnoba.edu.ar -t 20
Connecting to host stor11.ceph.unnoba.edu.ar, port 5201
[...]
[  5]   0.00-20.04  sec  21.9 GBytes  9.40 Gbits/sec
```

Figura 44. Medición de Ancho de banda efectivo

Las pruebas (Figura 45 y Figura 46) entre nodos conectados a distintos switches, con interfaces de 10Gbps, arrojan como resultado:

```
Connecting to host stor12.ceph.unnoba.edu.ar, port 5201
      Cookie: ycistnjq5l3c7j2r3opf4bxu6pvuo5rmjbuk
      TCP MSS: 1448 (default)
Starting Test: protocol: TCP, 1 streams, 131072 byte blocks, omitting 0
seconds, 20 second test, tos 0
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  5]   0.00-1.00   sec  1.10 GBytes  9.42 Gbits/sec    0   1.34 MBytes
[  5]   1.00-2.00   sec  1.10 GBytes  9.41 Gbits/sec    0   1.72 MBytes
[  5]   2.00-3.00   sec  1.09 GBytes  9.41 Gbits/sec    0   2.29 MBytes
[  5]   3.00-4.00   sec  1.09 GBytes  9.40 Gbits/sec   37   1.25 MBytes
[  5]   4.00-5.00   sec  1.10 GBytes  9.42 Gbits/sec    0   1.42 MBytes
[  5]   5.00-6.00   sec  1.09 GBytes  9.39 Gbits/sec    0   1.73 MBytes
[  5]   6.00-7.00   sec  1.10 GBytes  9.41 Gbits/sec    0   1.91 MBytes
[  5]   7.00-8.00   sec  1.09 GBytes  9.40 Gbits/sec    0   2.05 MBytes
[  5]   8.00-9.00   sec  1.09 GBytes  9.41 Gbits/sec    0   2.16 MBytes
[  5]   9.00-10.00  sec  1.09 GBytes  9.40 Gbits/sec   32   1.69 MBytes
[  5]  10.00-11.00  sec  1.09 GBytes  9.40 Gbits/sec    0   1.72 MBytes
[  5]  11.00-12.00  sec  1.09 GBytes  9.41 Gbits/sec    0   1.91 MBytes
[  5]  12.00-13.00  sec  1.09 GBytes  9.40 Gbits/sec    0   1.95 MBytes
[  5]  13.00-14.00  sec  1.10 GBytes  9.42 Gbits/sec    0   2.06 MBytes
[  5]  14.00-15.00  sec  1.09 GBytes  9.40 Gbits/sec    0   2.13 MBytes
[  5]  15.00-16.00  sec  1.09 GBytes  9.41 Gbits/sec   11   1.75 MBytes
[  5]  16.00-17.00  sec  1.09 GBytes  9.40 Gbits/sec    0   1.95 MBytes
[  5]  17.00-18.00  sec  1.09 GBytes  9.41 Gbits/sec    0   2.11 MBytes
[  5]  18.00-19.00  sec  1.09 GBytes  9.40 Gbits/sec   39   1.68 MBytes
[  5]  19.00-20.00  sec  1.09 GBytes  9.40 Gbits/sec    0   1.81 MBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
Test Complete. Summary Results:
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-20.00  sec  21.9 GBytes  9.40 Gbits/sec  119             sender
```

Figura 45. Ancho de banda medido entre nodos conectados a distintos switches.

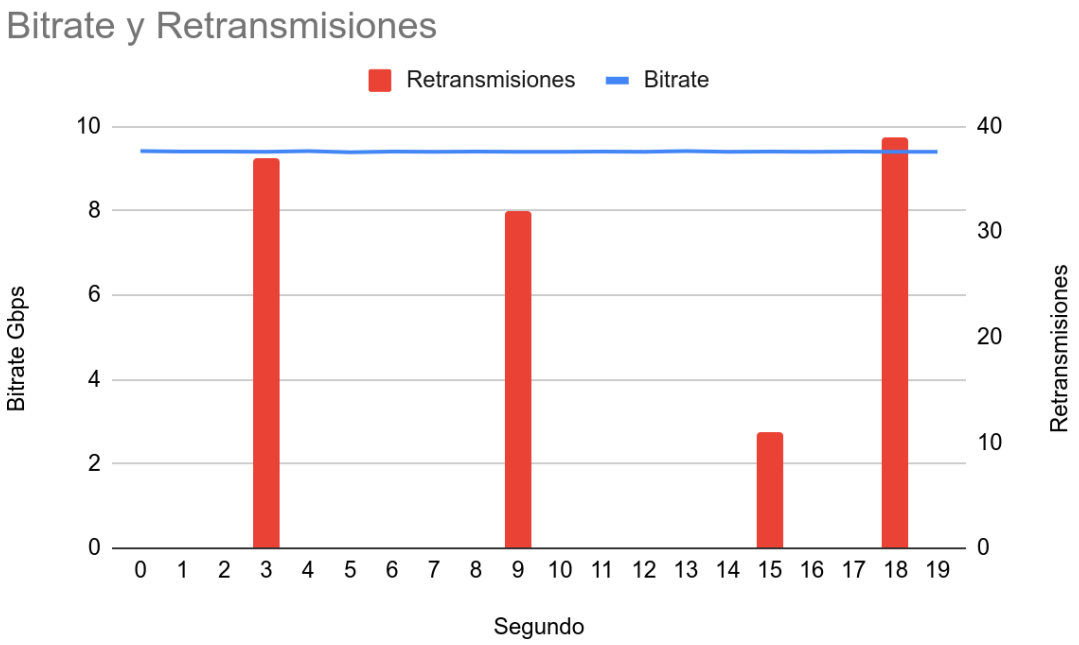


Figura 46. Ancho de Banda y Retransmisiones

Y las pruebas contra un nodo con interfaces de 1Gbps, arrojan resultados consistentes con el hardware (Figura 47).

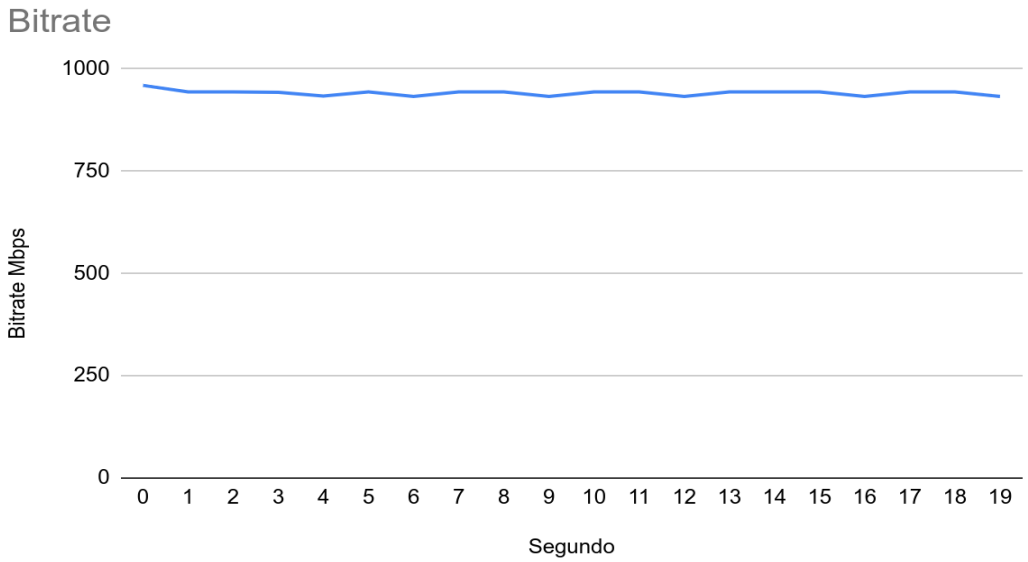


Figura 47. Pruebas en nodos con interfaces de 1Gbps.

Estos valores proveen un **límite máximo** respecto del rendimiento que la red del cluster impone al throughput entre los nodos, y entre los nodos y los clientes.

Algunas investigaciones como “Leveraging RDMA Technologies to Accelerate Ceph Storage Solutions” (Intel, 2018) directamente postulan que el stack TCP/IP es demasiado lento para Ceph (Figura 48) y proponen reemplazarlo por el protocolo iWARP RDMA.

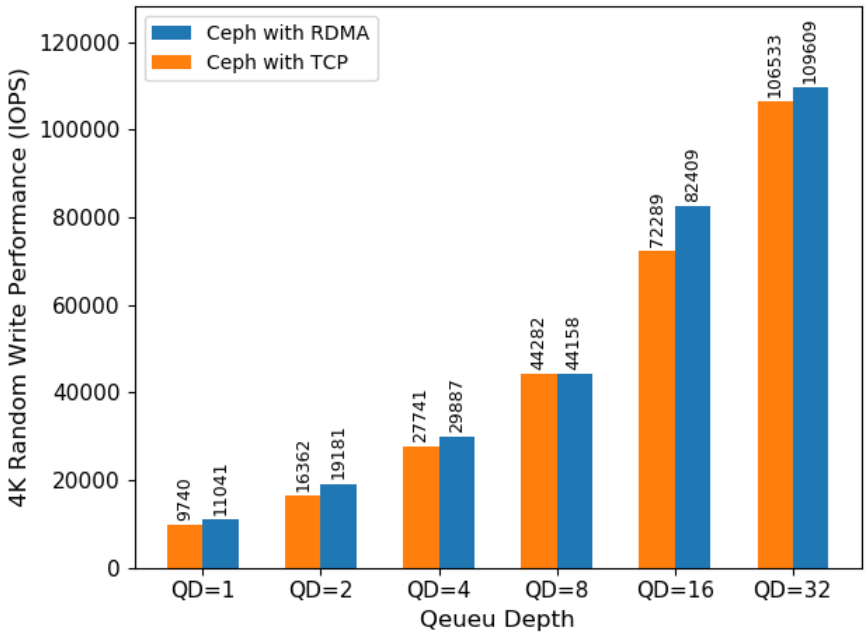


Figura 48. Comparativa de performance cambiando TCP por RDMA

El paso siguiente es analizar los discos de manera individual, antes de ser incluidos en el cluster para ser configurados como OSDs.

6. BENCHMARK de los DISCOS

Ceph va a trabajar tan rápido como el más lento de sus componentes, dice Poat en su trabajo “Achieving Cost/Performance Balance Ratio using Tiered Storage Caching Techniques” (Poat, 2017) Entonces, ¿Cómo vamos a identificar el cuello de botella? ¿Qué vamos a medir? Como se mencionó en el capítulo 4 al explicar la metodología de testeo utilizada, **el throughput** es una de las métricas más comunes para medir la performance de un dispositivo, registrando cuántos MB/segundo es capaz de transferir.

En los discos magnéticos, esta métrica está determinada inicialmente por la velocidad de rotación del disco y la cantidad de cabezales disponibles. En un disco de 7200 rpm, con un único cabezal, la tasa de transferencia está determinada por la cantidad de sectores que pasan por debajo del cabezal, además de las restricciones de ancho de banda que introducen el controlador del dispositivo y la tecnología de conexión: SATA3 opera en 3 y 6 Gbps, y SAS lo hace en 22 Gbps.

Aquí hay que tener en cuenta la diferencia de MB/s (MegaBytes por segundo) con Gbps (Gigabits por segundo). Además, mucha de la bibliografía y de las herramientas utilizadas hacen referencia a unidades definidas por “*IEEE 1541-2002 - IEEE Standard for Prefixes for Binary Multiples*” (IEEE, 2009) donde en lugar de utilizar potencias de 10, se utilizan potencias de 2 (prefijos binarios) (Figura 49)

bit (símbolo b), un dígito binario.
byte (símbolo B), conjunto de 8 bits adyacentes.

kibi- (símbolo Ki), 2^{10} = 1024 Bytes
mebi- (símbolo Mi), 2^{20} = 1048576 Bytes
gibi- (símbolo Gi), 2^{30} = 1073741824 Bytes
tebi- (símbolo Ti), 2^{40} = 1099511627776 Bytes
pebi- (símbolo Pi), 2^{50} = 1125899906842624 Bytes

Entonces,
1MB = 1.000.000 Bytes, pero 1MiB = 1.048.576 Bytes.

Multiples of bytes v·t·e

Decimal			Binary			
Value	Metric		Value	IEC		JEDEC
1	B	byte	1	B	byte	B byte
1000	kB	kilobyte	1024	KiB	kibibyte	KB kilobyte
1000^2	MB	megabyte	1024^2	MiB	**mebibyte**	MB megabyte
1000^3	GB	gigabyte	1024^3	GiB	gibibyte	GB gigabyte
1000^4	TB	terabyte	1024^4	TiB	tebibyte	–
1000^5	PB	petabyte	1024^5	PiB	pebibyte	–
1000^6	EB	exabyte	1024^6	EiB	exbibyte	–
1000^7	ZB	zettabyte	1024^7	ZiB	zebibyte	–
1000^8	YB	yottabyte	1024^8	YiB	yobibyte	–

Orders of magnitude of data

Figura 49. Estándar IEEE de prefijos para múltiplos binarios.

Así, la transferencia máxima de un bus SATA3 de 6 Gbps tendría un throughput máximo teórico de 6 Gbps = 6.000.000.000 bps = 750.000.000 Bytes por segundo.

Como 1 Mebibyte = 2^{20} = 1.048.576 bytes,
750.000.000 Bytes / 2^{20} = 715,25 MiB (Mebibytes)

Pero si los cálculos se hacen con potencias de 10:

Como 1 Megabyte = 10^6 = 1.000.000 bytes
750.000.000 Bytes = 750 MB (Megabytes)

Entonces, 750 MB = 715,25 MiB y esto se presta a confusión si las magnitudes no se miran con detenimiento.

6.1. Latencia

Mientras que el throughput hace referencia a cuántos bytes de datos por segundo se transfieren hacia o desde el disco, **la latencia** -usualmente medida en micro o milisegundos- refiere a la cantidad de tiempo que toma completar la lectura o escritura de un bloque.

En los discos magnéticos convencionales, este retardo está dado por el tiempo en que tarda el brazo mecánico en ubicar el cabezal sobre la pista donde está el sector a leer (o escribir) y el tiempo en que tarda el disco en girar y ubicar ese sector bajo el cabezal. En un disco que gira a 7200 rpm, la latencia rotacional promedio es de 0.5*(60/7200) ≈ 4,17 milisegundos (Hard Drive, 2020). Estas dos latencias, rotación y posicionamiento (rotational and seek latency), normalmente suman entre 15 y 25 milisegundos por acceso.

Estas latencias pueden observarse sobre todo en patrones de acceso *random*, donde el cabezal debe moverse entre distintas pistas, afectando el throughput. Cuando el acceso es secuencial, la tasa de transferencia aumenta considerablemente.

6.2. IOPS

IOPS son las siglas de “Input/Output Operations Per Second” y es la métrica de medida más utilizada en el benchmark de dispositivos de almacenamiento. ¿Cuántas operaciones de Entrada/Salida pueden hacerse por segundo?

Una I/O es una solicitud única de lectura/escritura que se envía al medio de almacenamiento y que tiene asociado un tamaño que pueden ser de algunos Kilobytes o varios Megabytes. Si multiplicamos el número de IOPS por el tamaño de la solicitud, tendremos un número aproximado de throughput, o ancho de banda.

Los IOPS pueden variar de acuerdo al balanceo de operaciones de lectura/escritura, la combinación de patrones de acceso aleatorios y secuenciales, el número de worker threads, la *queue_depth*, el tamaño de bloque de datos, la configuración general del sistema, los drivers de almacenamiento, la gestión que hace el sistema operativo de las operaciones de E/S, entre otros factores.

6.3. Throughput

En la bibliografía, throughput es utilizada en lecturas/escrituras secuenciales de grandes archivos. Pero -como se mencionó- esta métrica es afectada en accesos aleatorios

por la latencia rotacional y el tiempo de posicionamiento, para esos patrones de acceso con bloques de 4K, suele utilizarse IOPS como métrica.

Además, los discos de estado sólido (SSD) no tienen brazo mecánico para posicionar ni discos rotando, de manera que no sufren esas latencias. Sin embargo, son afectados por el acceso de bloques de 4K de manera aleatoria. Un disco SSD es, sintéticamente, un arreglo de memorias de acceso paralelo, con un controlador interno que coordina las lecturas/escrituras. El controlador trata de fragmentar los datos y realizar las escrituras sobre las memorias flash en múltiples canales internos y de forma paralela. Del mismo modo procede con las operaciones de lectura, y de esa manera logra maximizar el throughput.

Cuando un dispositivo SSD debe realizar una carga de trabajo de lecturas y escrituras de bloques aleatorios de 4K, si el controlador no puede *agregar* las operaciones, se va a encontrar con un cuello de botella determinado por la velocidad a la que una celda de memoria puede escribir/leer un único bloque.

Es importante repetir que los IOPS deben ir acompañados de la latencia porque el tiempo que toma completar cada una de esas solicitudes de E/S indica lo que en inglés se llama grado de *responsiveness* del dispositivo.

6.4. Nuevas interfaces

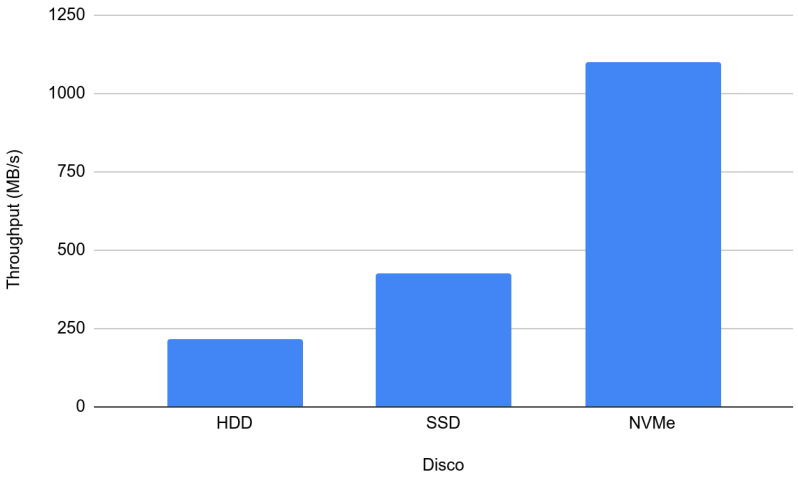
NVM Express (NVMe) es una especificación abierta de interfaz de dispositivo para acceder a medios de almacenamiento conectados a través del bus PCI Express (PCIe) en vez de las interfaces SATA. Está diseñado para aprovechar la baja latencia y el paralelismo interno de los dispositivos de almacenamiento de estado sólido. Por su diseño, NVMe reduce el overhead en las operaciones de E/S y ofrece varias mejoras de rendimiento: procesamiento de los comandos de E/S en múltiples colas y latencia reducida. Entonces, un disco NVMe tiene mejor performance que un SSD, y éste mejor que un HDD. ¿Pero cuánto mejor?

Un breve testeo inicial (Figura 50): escritura de un archivo de 5 Gigabytes. El disco HDD completa la tarea en 25 segundos, el SSD en 13 y el NVMe en 5. Es una escritura secuencial, continua, en un único proceso, sin restricciones respecto de cantidad de operaciones enviadas en simultáneo por el sistema operativo al dispositivo.

```
# dd if=/dev/zero of=archivo bs=1G count=5
```

Disco	Tiempo (seg)	Throughput (MB/s)
HDD	24,86	216
SSD	12,56	427
NVMe	4,93	1100

Figura 50. Testeo inicial para obtener valores de referencia.



Un trabajo interesante que en sus primeros párrafos dice "*Ceph es lento con NVMe*" puede leerse en la wiki de Vitalii Filipov (Filippov, 2019) quien realiza un estudio exhaustivo de Ceph en estos dispositivos, indicando que podrían realizarse mejoras en el código de Ceph para que se adapte mejor a la arquitectura y funcionamiento de estos dispositivos.

6.5. Una pincelada de realidad

Aunque la performance de los discos de estado sólido son claramente tentadoras y en los párrafos siguientes vamos a poner en números estas promesas que *a priori* nos seducen, el costo de los mismos, consultado a proveedor nacional del rubro en noviembre/2020 (Figura 51), también es un detalle a evaluar a la hora de implementar un sistema de almacenamiento distribuido:

1 disco HDD, de 1024 GB de capacidad, SATA, 7200 rpm	usd 152
1 disco SSD, de 960 GB de capacidad, SATA, enterprise level	usd 823
1 disco NVMe, 960 GB de capacidad, enterprise level	usd 1660

Figura 51. Precios del mercado argentino. Nov 2020.

Esto nos da un poco de perspectiva respecto de qué disponibilidad económica tenemos que tener a la hora de tentarnos con papers y bibliografía de otras geografías, donde la implementación del cluster en uno u otro dispositivo responde a requerimientos de performance y volumen de información, y lo económico -aunque es un punto que siempre se evalúa- no es necesariamente una restricción.

En el resto de este trabajo, sólo vamos a considerar los discos HDD y SSD, disponibles en el cluster de Ceph descripto en el capítulo 4.

6.6. Unas palabras sobre buffers y cachés

Hay que tener en cuenta que los sistemas operativos utilizan cachés de lecturas y buffers de escrituras para minimizar el número de operaciones seek necesarias y evitar la necesidad de releer bloques de datos usados frecuentemente.

Los buffers de escritura permiten al sistema operativo almacenar muchas pequeñas operaciones de E/S y realizarlas luego de manera conjunta sobre el disco. Un megabyte no parece ser mucha información y -sin embargo- terminan siendo 256 bloques de 4KB, que deberían ser escritos en operaciones individuales, y ocuparían la capacidad de trabajo del disco por 1 segundo completo.

Así, si se almacenan esos 256 bloques en un buffer de escritura y luego se hace un flush-out en una única operación, se evita casi toda la latencia de acceso al disco (seek time, rotational time…) y la misma cantidad de datos puede escribirse en una mínima fracción de segundo. Este mismo procedimiento beneficia a las lecturas posteriores, porque posiblemente los datos escritos de manera conjunta, deban ser leídos de la misma manera.

Las caché de lectura evitan que el sistema operativo reitere operaciones sobre el mismo bloque de datos, manteniendo en memoria aquellos que son accedidos frecuentemente.

En el contexto de este trabajo, como deseamos testear la performance nativa de los dispositivos estudiados, las operaciones de benchmarking se deben realizar deshabilitando tanto buffers como cachés del sistema operativo.

6.7. Benchmarking con FIO

El benchmarking de discos es un procedimiento complejo. Leer o escribir una gran cantidad de datos, buscando obtener cuántos MB/s se transfieren al o desde el disco, no es en realidad un número significativo, porque la carga de trabajo sobre el dispositivo rara vez se identifica con ese patrón de acceso y esa prueba no puede reproducir los muchos cuellos de botella que ralentizan el acceso a un disco en el trabajo diario.

La forma más realista de testear y realizar el benchmark sobre el dispositivo es usarlo y guardar estadísticas para analizar. Pero este proceso no es repetible, ni es breve, ni arroja datos simples para analizar. Por eso utilizamos FIO, una herramienta que permite simular el uso de dispositivos en distintos escenarios.

FIO es una herramienta open-source para generar tráfico con un patrón de E/S especificado por el usuario: lecturas secuenciales, escrituras random, sincrónicas o asincrónicas. Puede generar distintos subprocesos de manera de testear el acceso concurrente de distintos accesos al mismo recurso, dependiendo de la carga de E/S que se desea simular.

De las muchas opciones que FIO presenta, en nuestras pruebas vamos a utilizar algunas de ellas. (Figura 52)

```
fio --ioengine=libaio --direct=1 --invalidate=1
--name=fio_randrw_4k --bs=4k --iodepth=16 --size=1G
--numjobs=4 --readwrite=randrw --rwmixread=75
```

Figura 52. Ejemplo de comando fio.

`ioengine=libaio`

Define cómo los jobs envían las solicitudes de E/S al archivo de prueba. libaio=Linux Native Asynchronous I/O.

`name=fio_randrw_4k`

Nombre del archivo que crea para realizar la E/S solicitada.

`direct=1`

E/S sin buffer.

`invalidate=1`

Invalida la caché de páginas/buffer de los archivos que se utilizarán antes de iniciar la E/S, si la plataforma y el tipo de archivo lo admiten.

`bs=4k`

Tamaño de bloque, en bytes, usado para las solicitudes de E/S.

`size=1G`

Tamaño en bytes del archivo de pruebas utilizado por cada proceso de la tarea. FIO va a continuar trabajando hasta que este tamaño sea transferido, o hasta que se cumpla el tiempo especificado por --runtime.

`numjobs=4`

Clona 4 procesos concurrentes de la misma tarea.

`iodepth=32`

Número de solicitudes de E/S concurrentes que realiza cada proceso al Sistema Operativo (Aplicable sólo a patrones de acceso asincrónicos)

`readwrite=randrw`

Define el patrón de E/S.

Los valores pueden ser: read (lectura secuencial), write (escritura secuencial), randread (lectura aleatoria), randwrite (escritura aleatoria), randrw (lectura/escritura aleatorias)

`rwmixread=75`

Cuando se utiliza como patrón de acceso randrw, rwmixread establece qué porcentaje de los accesos al disco deben ser lecturas.

6.8. Patrones de acceso, tamaños de bloque y queue_depth

Los patrones de acceso a los dispositivos de almacenamiento pueden ser secuenciales o aleatorios. Un acceso secuencial puede ser copiar un archivo grande de un disco a otro, en la que se escriben una gran cantidad de bloques, muchas veces contiguos. Las tareas de backup, el render de archivos de video, la exportación de bases de datos, suelen tener este patrón de acceso que registra el throughput más alto, debido a que minimiza los movimientos de reposicionamiento del brazo (seek time). El acceso aleatorio (random) obliga al mecanismo del dispositivo a posicionarse en distintas ubicaciones antes de realizar las operaciones solicitadas. El tiempo de movimiento del brazo antes de iniciar la lectura o escritura aumenta la latencia en cada operación.

Las herramientas utilizadas para medir la performance de los discos, permite especificar el número de operaciones que pueden encolarse concurrentemente (*queue_depth*), y permitir así que el controlador del disco elija la mejor manera de procesarlas, a costo de aumentar la latencia.

El tamaño de bloque en la mayoría de las pruebas de lectura y escritura de bloques pequeños se establece en 4 KiB porque los sectores físicos de los discos HDD y las páginas de memoria virtual de Linux son de ese tamaño.

Se observará en algunas gráficas como los distintos parámetros afectan la performance general de los discos HDD y SSD, en términos de throughput, latencia e IOPS. Cuando el throughput no se menciona, tener en cuenta que las IOPS multiplicadas por el tamaño de bloque (de tamaño fijo), ofrecen la información del throughput máximo.

6.9. Hard Disk Drives

Los discos HDD analizados son discos HP de 4TB, 7200 rpm, SATA3 a 6 Gbps. (Figura 53)

```
Device Model:     MB4000GVYZK
Serial Number:    ZC1DF99E
LU WWN Device Id: 5 000c50 0c587c8c6
Firmware Version: HPG4
User Capacity:    4.000.787.030.016 bytes [4,00 TB]
Sector Size:      512 bytes logical/physical
Rotation Rate:    7200 rpm
Form Factor:      3.5 inches
ATA Version is:   ACS-3 T13/2161-D revision 5
SATA Version is:  SATA 3.1, 6.0 Gb/s (current: 6.0 Gb/s)
```

Figura 53. Descripción de los discos magnéticos utilizados.

Una primera prueba (Figura 54) sobre el disco, nos da una idea de su throughput máximo, solicitando que escriba un archivo de 5G, sin intervención de los buffers del kernel.

```
# dd if=/dev/zero of=/dev/sdc bs=1G count=5 oflag=direct
5368709120 bytes (5,4 GB, 5,0 GiB) copied, 27 s, 202 MB/s
```

Figura 54. Prueba inicial de desempeño del disco HDD.

Se analiza ahora con la herramienta *fio* la variación de IOPS, throughput y latencia variando como único parámetro el tamaño de bloque entre 4 KiB y 4 MiB, con 4 procesos concurrentes, cada uno con una queue_depht = 4, realizando operaciones por un total de 1 GiB de datos. (Figura 55)

Tamaño de Bloque	SECUENCIAL					
	ESCRITURA			LECTURA		
	IOPS	BW (MiB/s)	Latencia (mseg)	IOPS	BW (MiB/s)	Latencia (mseg)
4k	1062	4.15	15.06	53000	207.00	0.26
16k	1808	28.20	8.84	19800	309.00	0.70
64k	768	48.00	20.53	3306	207.00	4.28
128k	699	87.40	22.80	2082	260.00	7.82
256k	563	141.00	27.97	870	218.00	15.82
512k	353	177.00	40.80	397	199.00	36.63
1m	195	196.00	59.08	205	206.00	68.81
4m	47	188.00	320.00	48	194.00	304.00

Tamaño de Bloque	RANDOM					
	ESCRITURA			LECTURA		
	IOPS	BW (MiB/s)	Latencia (mseg)	IOPS	BW (MiB/s)	Latencia (mseg)
4k	144	0.56	47.85	431	1.68	21.02
16k	138	2.16	49.67	412	6.45	22.05
64k	121	7.57	55.19	357	22.30	25.75
128k	104	13.10	63.42	304	38.10	30.40
256k	84	21.00	74.92	240	60.20	39.44
512k	59	29.90	99.78	171	85.60	58.14
1m	35	35.00	150.83	101	101.00	102.76
4m	11	44.90	398.16	33	135.00	332.21

Figura 55. Variación de IOPS, throughput y latencia en discos HDD.

Analizamos ahora cómo la variación de *queue_depth* afecta la performance de las escrituras secuenciales (Ver Figura 56). Para un tamaño de bloque de 4KB, vemos como la latencia (expresada en milisegundos) disminuye y aumentan las IOPS conforme aumenta el tamaño de operaciones enviadas simultáneamente por el sistema operativo al dispositivo. Las imágenes fueron generadas utilizando la herramienta *fio-plot* (Lowrentius, 2020).

Para queue_depth=1, vemos que el dispositivo puede proveer de 120 operaciones, cada una con una latencia de 8.33 milisegundos.

Es interesante notar que el dispositivo tarda 8.41 milisegundos en completar 8 operaciones, mientras que tarda 5.25 en completar 16. Normalmente, estas variaciones responden a la arquitectura del disco y a cómo el controlador y el driver gestionan las colas de operaciones.

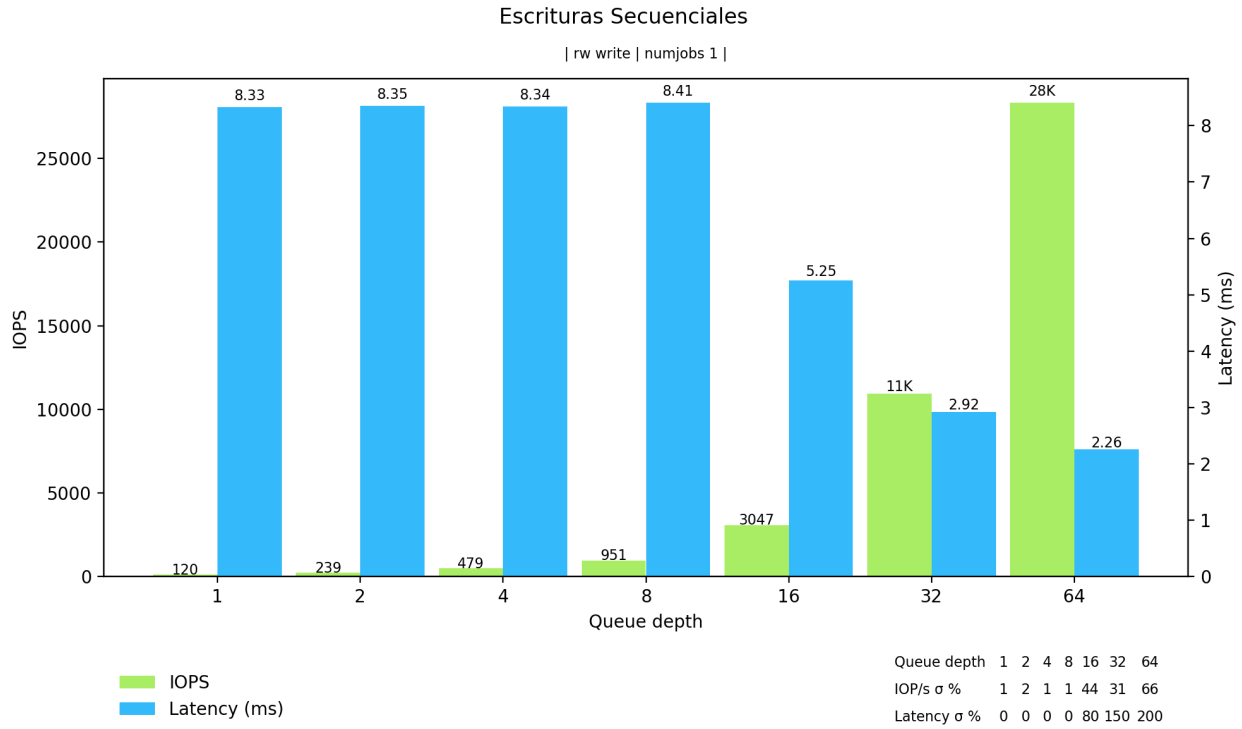


Figura 56. Escrituras Secuenciales sobre HDD.

En cambio, para las **lecturas secuenciales,** con más de 2 operaciones encoladas, la cantidad de IOPS toca su límite en 52000 y la latencia comienza a crecer (notar que está en microsegundos) . Para una queue_depth=1, vemos que el dispositivo puede proveer 27000 operaciones, y que cada lectura secuencial tiene una latencia de 37 microsegundos. (Figura 57)

Como mencionamos anteriormente, las operaciones sobre los dispositivos en un cluster de almacenamiento distribuido son **aleatorias,** pues se intercalan múltiples trabajos

de lectura y escritura de múltiples clientes, sobre distintos sectores del disco. Entonces, analizamos el comportamiento de escrituras aleatorias, con sectores de 4KB (Figura 58)

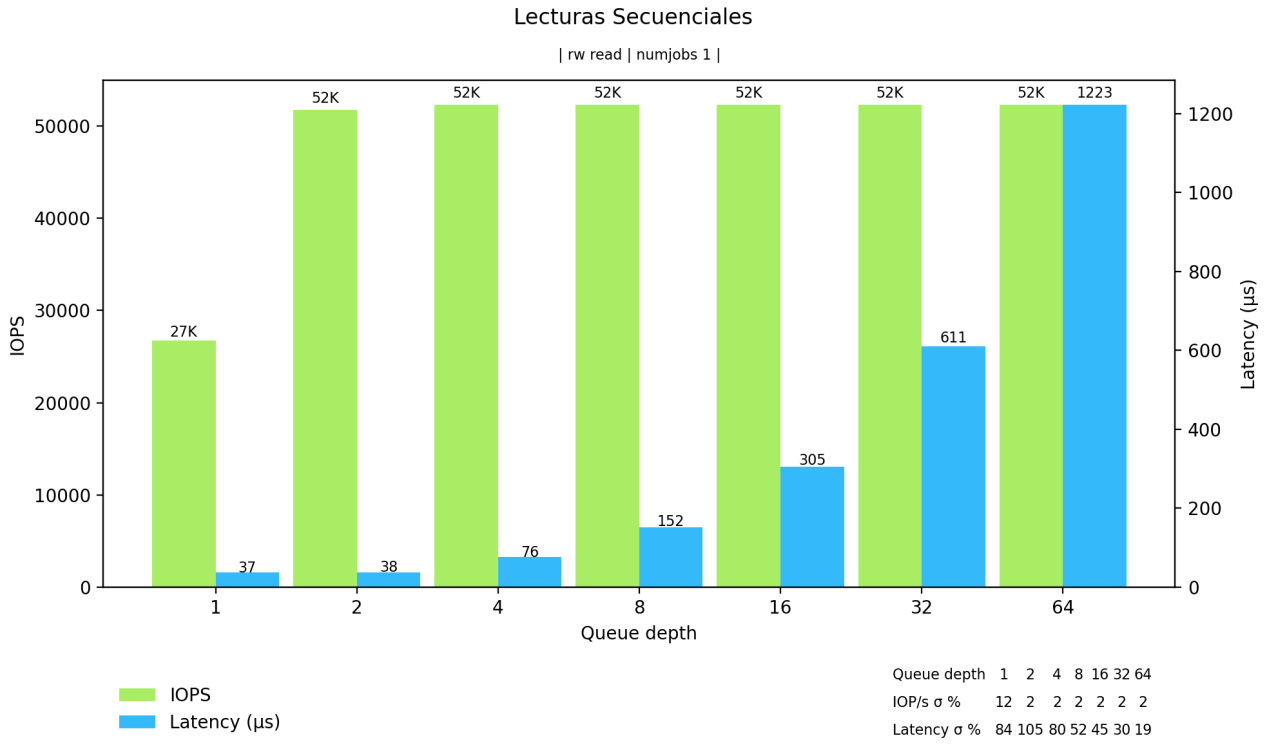


Figura 57. Lecturas Secuenciales sobre HDD.

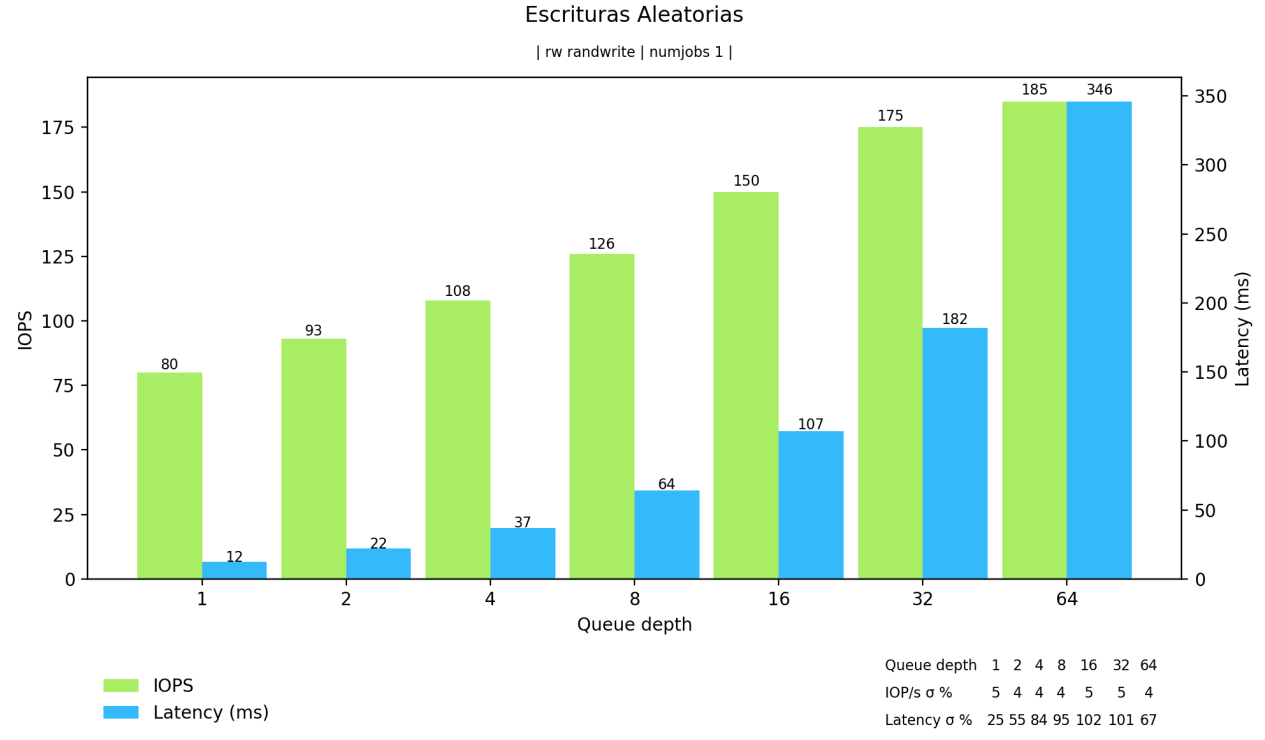


Figura 58. Escrituras Aleatorias sobre HDD.

Vemos que la cantidad de IOPS es notoria y obviamente menor que en las operaciones secuenciales y que su performance no aumenta significativamente con el aumento de *queue_depth*, como sí lo hace la latencia, y que el mismo patrón se repite en las lecturas aleatorias. (Figura 59)

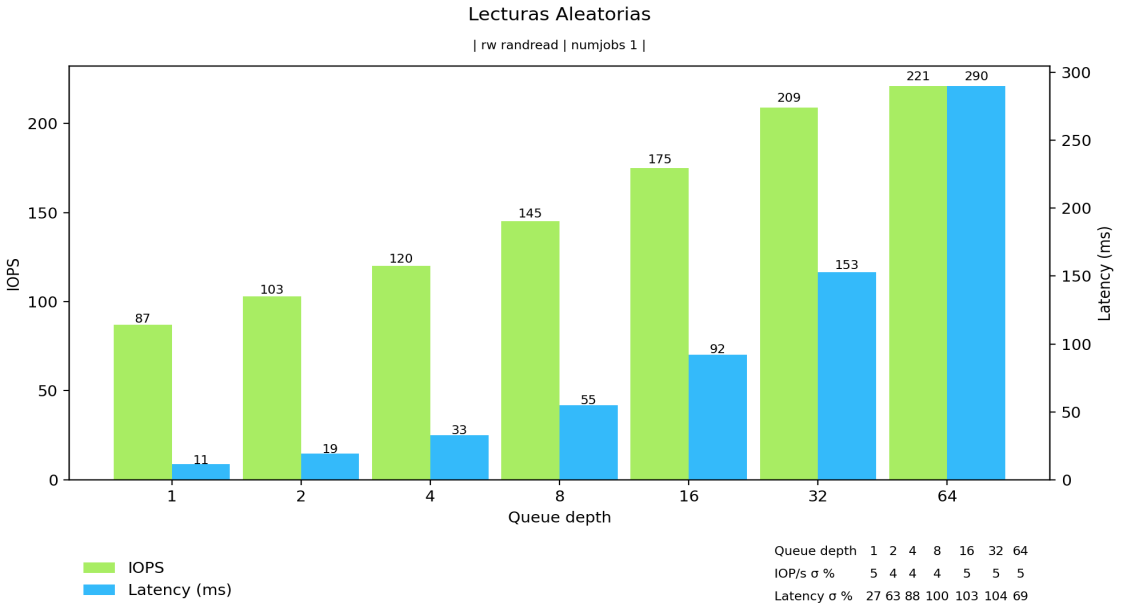


Figura 59. Lecturas Aleatorias sobre HDD.

Se prueba ahora cómo responde el disco en operaciones de escritura aleatoria, con una variación de queue_depth de 1 a 32, y sumando otra variable, el número de procesos que realizan operaciones concurrentes sobre el dispositivo. (Figura 60)

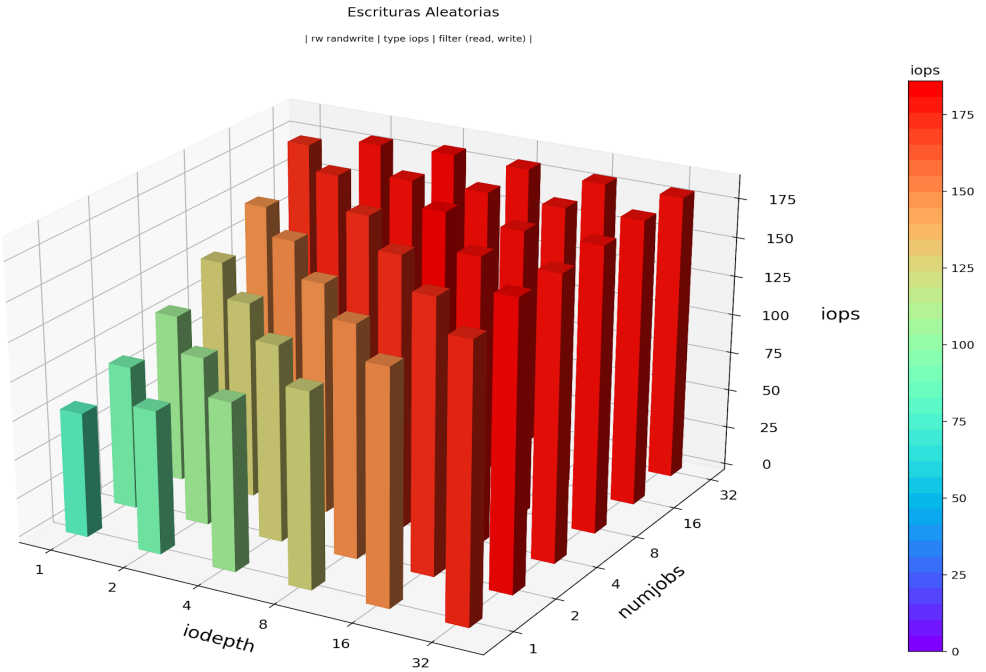


Figura 60. Escrituras aleatorias.

La siguiente imagen (Figura 61) muestra cómo aumenta la latencia (en milisegundos) respecto del número de procesos concurrentes y del número de operaciones simultáneas enviadas.

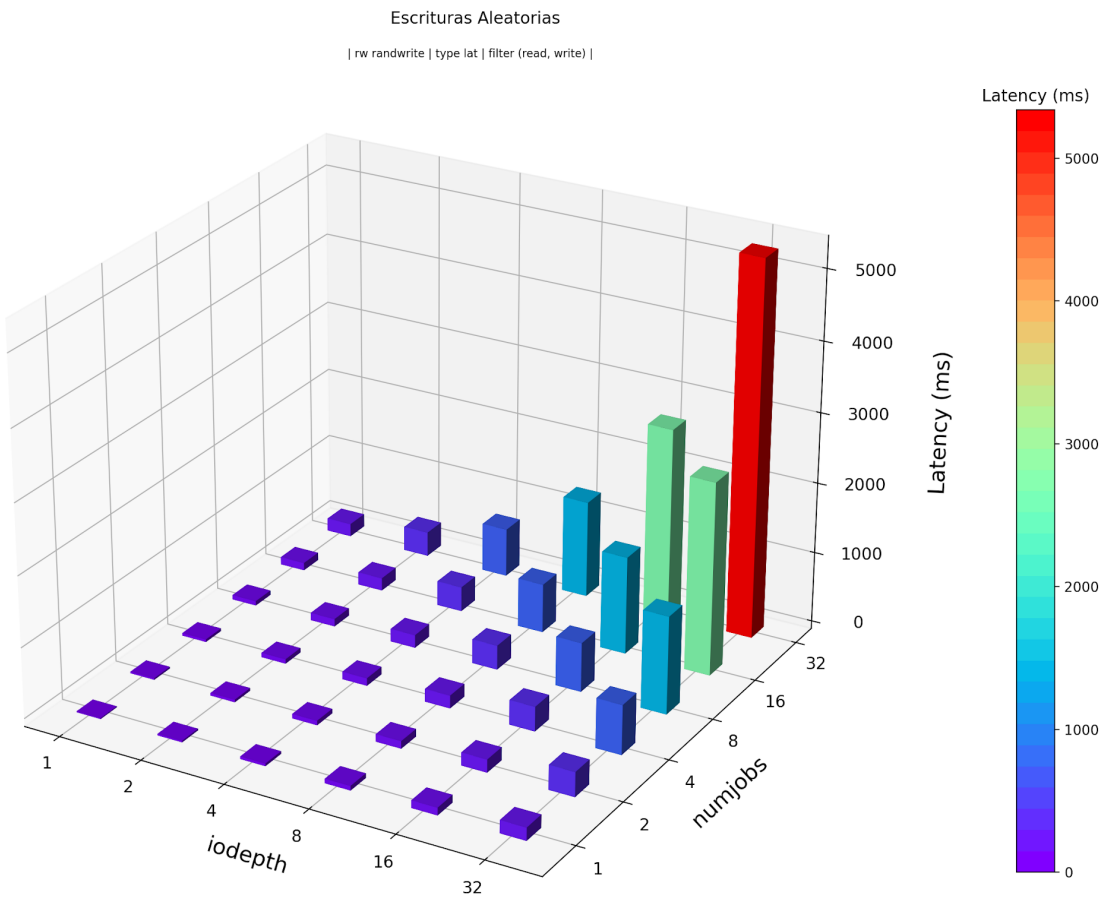


Figura 61. Registro de aumento de latencia en relación a la cantidad de procesos y número de operaciones.

6.10. Solid State Disks

Los discos de estado sólido están basados en chips de memoria flash y no tienen partes móviles, por eso pueden realizar muchas más operaciones de E/S por segundo, con menor latencia y generar una tasa de transferencia mucho mayor que los discos HDD.

Como se observó en los párrafos anteriores, la latencia de los discos mecánicos se mide usualmente en milisegundos (1/1000), mientras que la de los discos de estado sólido se mide en microsegundos (1/1000000).

Además, los discos sólidos de alta gama (enterprise level) pueden manejar mayor cantidad de operaciones de manera paralela, con lo cual -si bien la latencia de una

operación es la misma- puede procesar una cola de requerimientos de E/S en menor tiempo.

Los discos SSD (Figura 62) utilizados en el cluster son HP de 1.9TB, SATA3 a 6 Gbps.

```
=== START OF INFORMATION SECTION ===
Device Model:     MK001920GWSSE
Serial Number:    S523NA0N306149
LU WWN Device Id: 5 002538 e00395d03
Firmware Version: HPG1
User Capacity:    1.920.383.410.176 bytes [1,92 TB]
Sector Sizes:     512 bytes logical, 4096 bytes physical
Rotation Rate:    Solid State Device
Form Factor:      2.5 inches
ATA Version is:   ACS-4, ACS-3 T13/2161-D revision 5
SATA Version is:  SATA 3.2, 6.0 Gb/s (current: 6.0 Gb/s)
```

Figura 62. Características de los discos SSD utilizados.

Y se repite la prueba (Figura 63) sobre el disco para ver una primera idea de su throughput máximo, solicitando que escriba un archivo de 5G, sin intervención de los buffers del kernel.

```
# dd if=/dev/zero of=/dev/sdg bs=1G count=5 oflag=direct
5368709120 bytes (5,4 GB, 5,0 GiB) copied, 11,2378 s, 478 MB/s
```

Figura 63. Prueba inicial de desempeño del disco SSD.

Las pruebas anteriormente realizadas sobre los HDD se repiten ahora sobre los discos de estado sólido (Figura 64), variando inicialmente el tamaño de bloque entre 4 KiB y 4 MiB para observar los valores de IOPS, throughput y latencia, con 4 procesos concurrentes, cada uno con una queue_depht = 4, realizando operaciones por un total de 1 GiB de datos.

Tamaño de Bloque	SECUENCIAL					
	ESCRITURA			LECTURA		
	IOPS	BW (MiB/s)	Latencia (usec)	IOPS	BW (MiB/s)	Latencia (usec)
4k	51600	202.00	308.00	67000	266.00	233.96
16k	23300	364.00	684.78	25500	398.00	626.29
64k	7322	458.00	2182.00	7995	500.00	1998.01
128k	3824	478.00	4177.00	4111	514.00	3885.00
256k	1955	489.00	8172.00	2122	528.00	7566.00
512k	987	494.00	16178.00	1066	533.00	14983.00
1m	497	497.00	32139.00	534	535.00	29877.00
4m	122	491.00	117310.00	133	535.00	108000.00

Tamaño de Bloque	RANDOM					
	ESCRITURA			LECTURA		
	IOPS	BW (MiB/s)	Latencia (microseg)	IOPS	BW (MiB/s)	Latencia (microseg)
4k	17700	69.00	145.53	52800	206.00	252.42
16k	6552	102.00	385.93	19500	305.00	686.07
64k	1906	119.00	1774.00	5624	352.00	2235.00
128k	1007	126.00	3848.01	2927	366.00	4131.00
256k	520	130.00	7865.00	1490	373.00	7975.00
512k	252	126.00	16460.00	724	362.00	16322.00
1m	123	124.00	34419.00	348	348.00	33645.00
4m	32	131.00	120030.00	98	393.00	115550.00

Figura 64.Variación de IOPS, throughput y latencia en discos SSD.

Se realizan ahora escrituras secuenciales (Figura 65), con variación de queue_depth entre 1 y 128, para bloques 4KB y un único proceso activo.

Es interesante observar cómo entre 2 y 32 operaciones simultáneas, el número de IOPS es constante, y que luego vuelve a saltar. La latencia, en cambio, describe una curva con crecimiento típico.

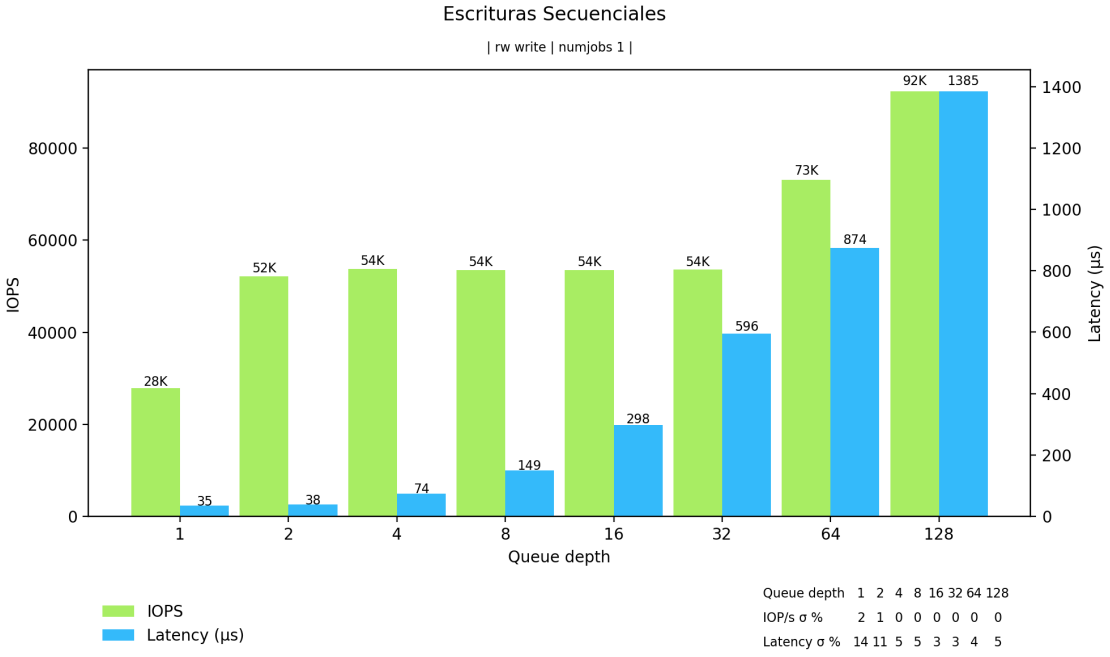


Figura 65. Escrituras Secuenciales sobre SSD

El comportamiento en las lecturas secuenciales (Figura 66), es muy similar al de las escrituras y no se puede apreciar la diferencia que se observaba entre una y otra operación en los discos HDD

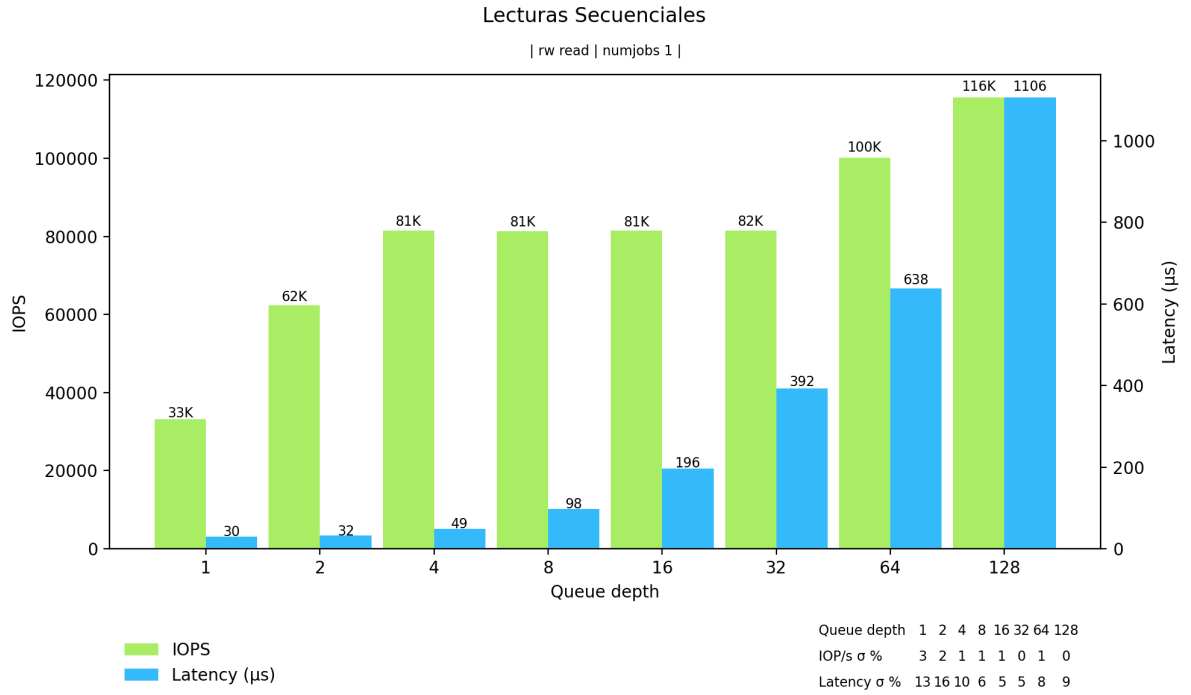


Figura 66. Lecturas secuenciales en SSD

Se testean ahora las operaciones aleatorias (Figura 67) y se observa que en las escrituras random, las IOPS a partir de 4 operaciones simultáneas se mantienen constantes en 53000 por segundo.

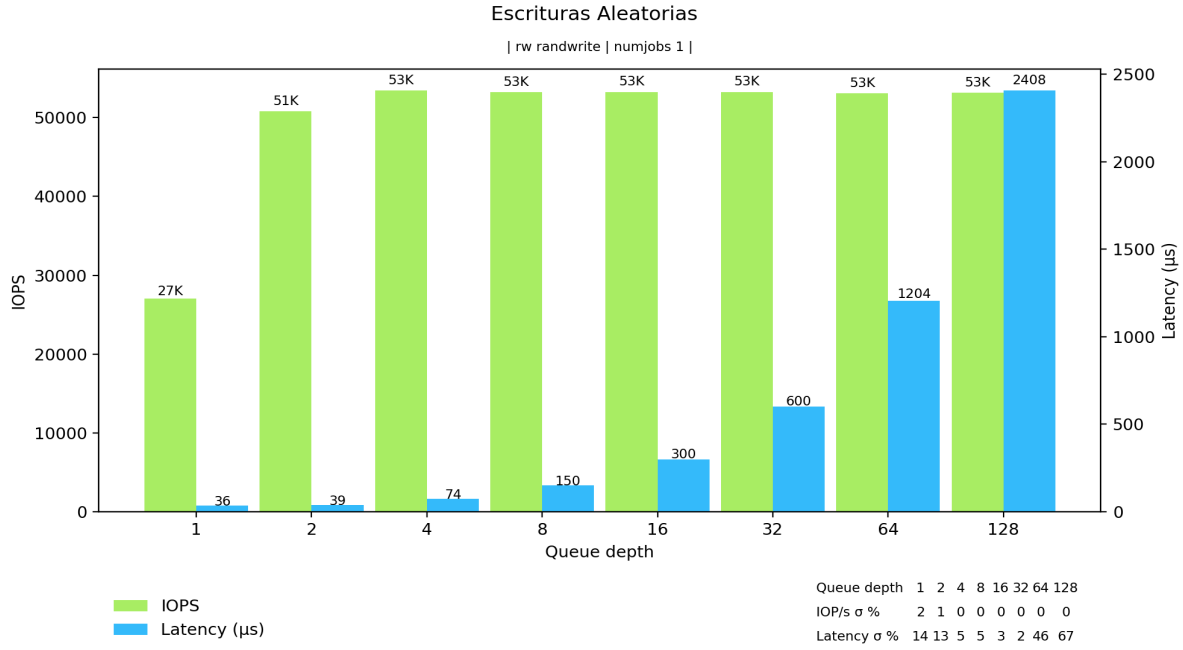


Figura 67. Escrituras aleatorias en SSD.

La latencia, medida en microsegundos, se duplica conforme se duplica el número de operaciones en cada medición, con lo que podríamos decir que la latencia no varía, pues 12 conjuntos de 8 operaciones tarda más o menos el mismo tiempo en procesarse que 6 conjuntos de 16, o que 3 de 32…

Las lecturas aleatorias (Figura 68), en cambio, estabilizan su número de IOPS en 82000 al alcanzar una queue_depth de 32 y la latencia varía sensiblemente entre los valores de queue_depth 1 y 16

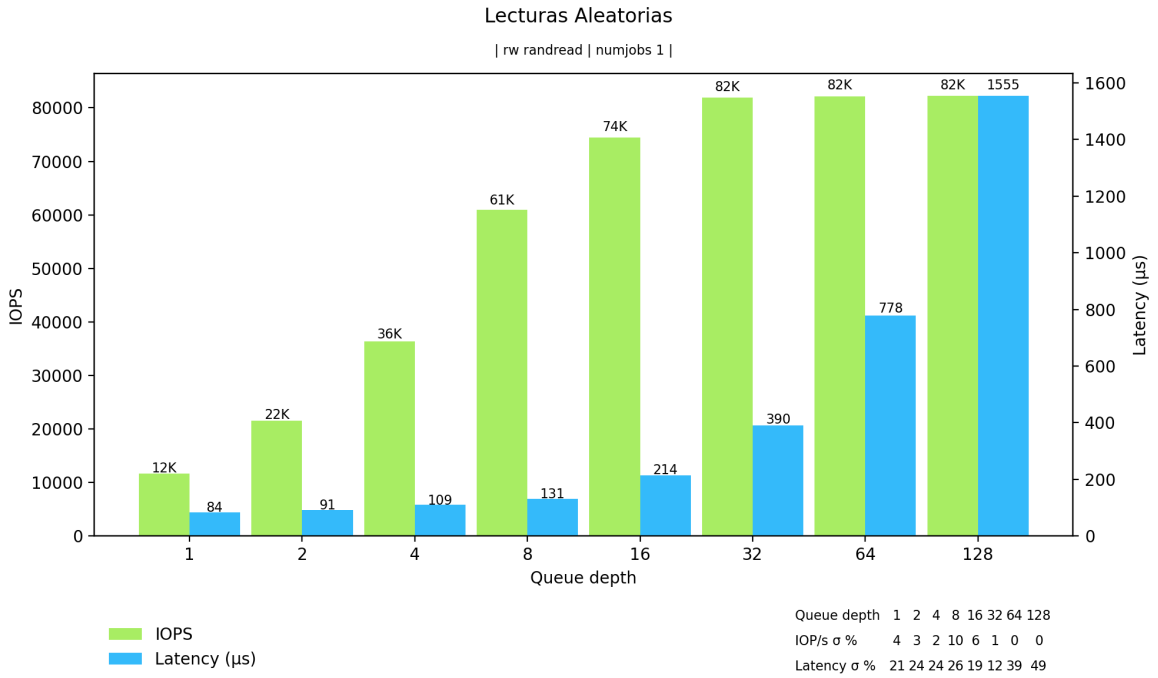


Figura 68. Lecturas aleatorias.

Se prueba a continuación cómo responde el disco en operaciones de escritura aleatoria, con una variación de queue_depth de 1 a 64, y sumando otra variable, el número de procesos que realizan operaciones concurrentes sobre el dispositivo.

En la Figura 69 y en la Figura 70 vemos que el disco SSD mantiene su máximo de IOPS entre 4 y 64 operaciones, y que su performance es muy buena entre 1 y 16 procesos; y que recién superando los 16 procesos con 16 operaciones concurrentes cada uno, la latencia se dispara y supera el orden de los microsegundos hacia los 10 milisegundos.

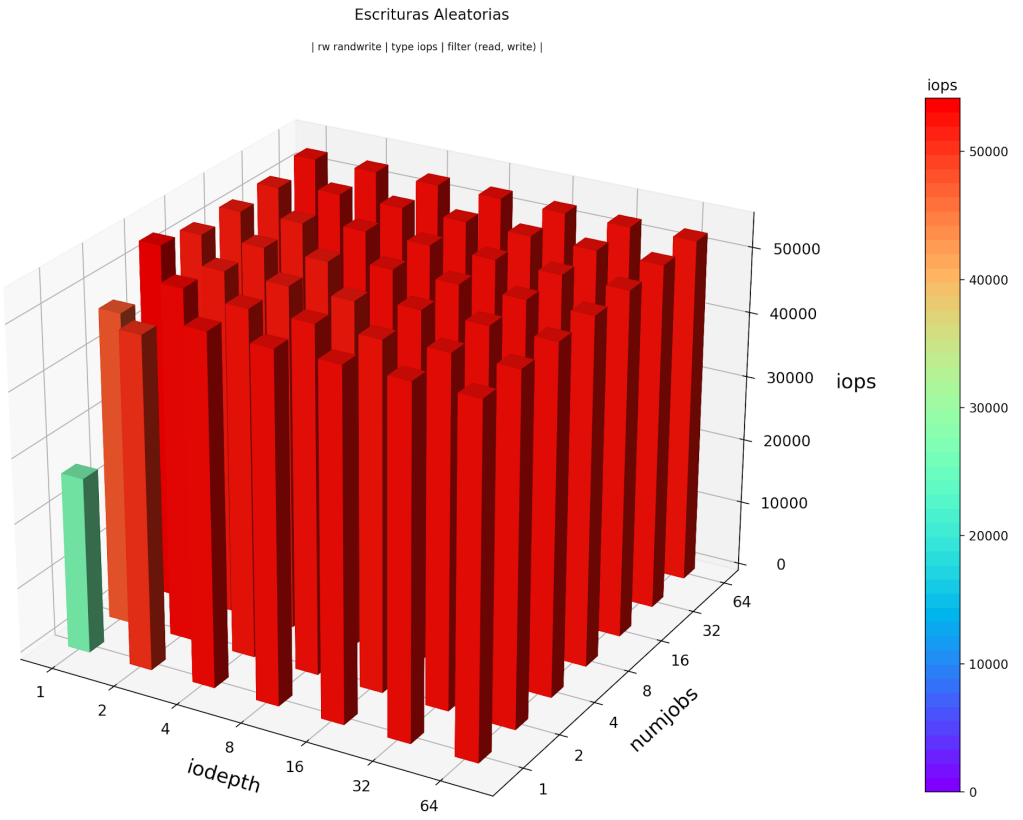


Figura 69. La variación de IOPS respecto de iodepth y número de procesos

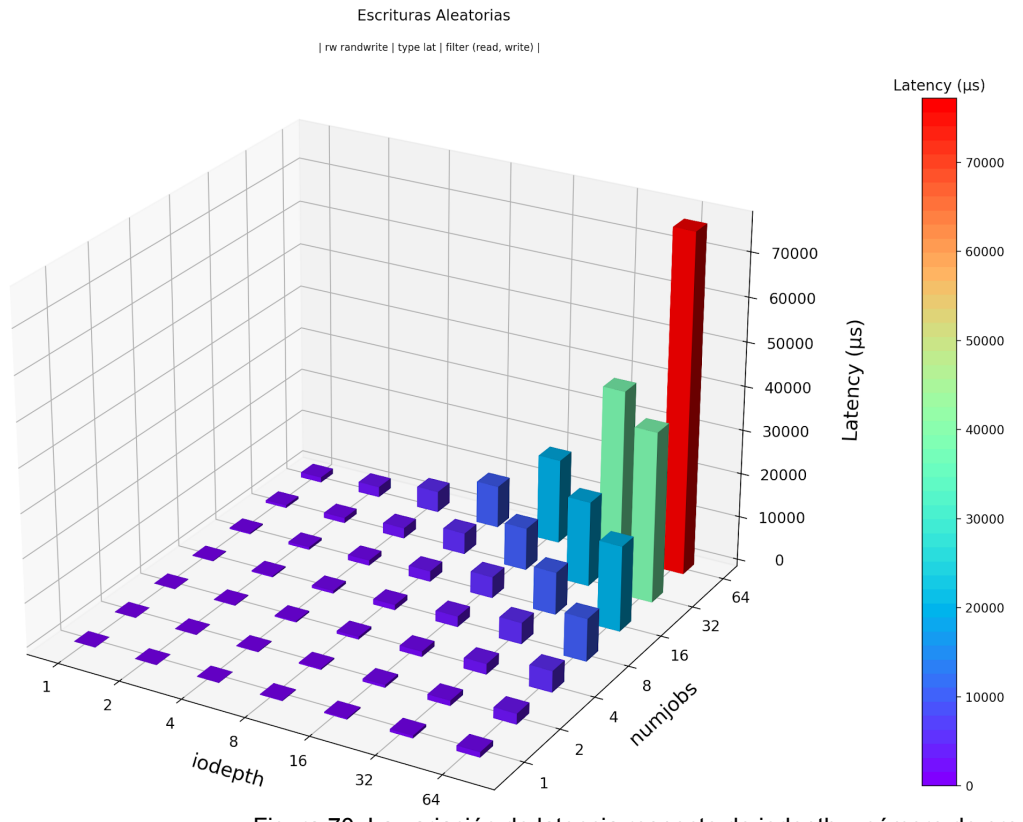
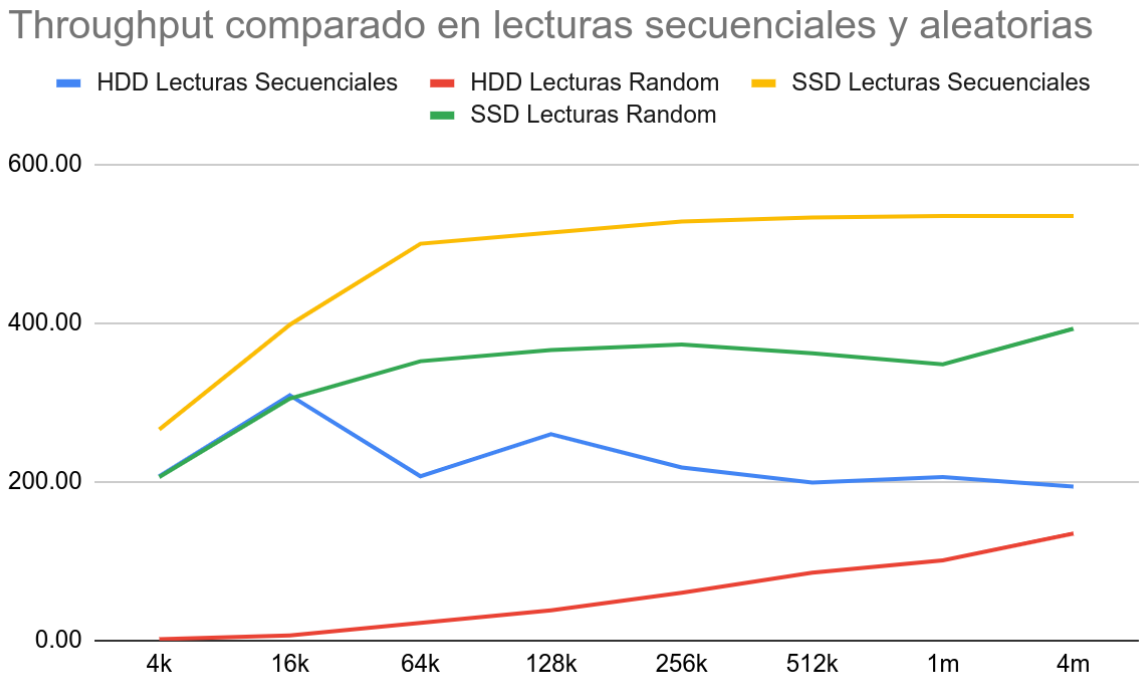


Figura 70. La variación de latencia respecto de iodepth y número de procesos

6.11. Algunas comparaciones más

De los testeos en ambos discos con 4 procesos realizando 4 operaciones de E/S cada uno para un total de 1GiB de datos, variando el tamaño de bloque, se obtuvieron los valores que se muestran en la Figura 71, de manera de visualizar el comportamiento de los discos y anticipar, de alguna manera, las decisiones de tamaño de objeto de 4 MiB para Ceph.



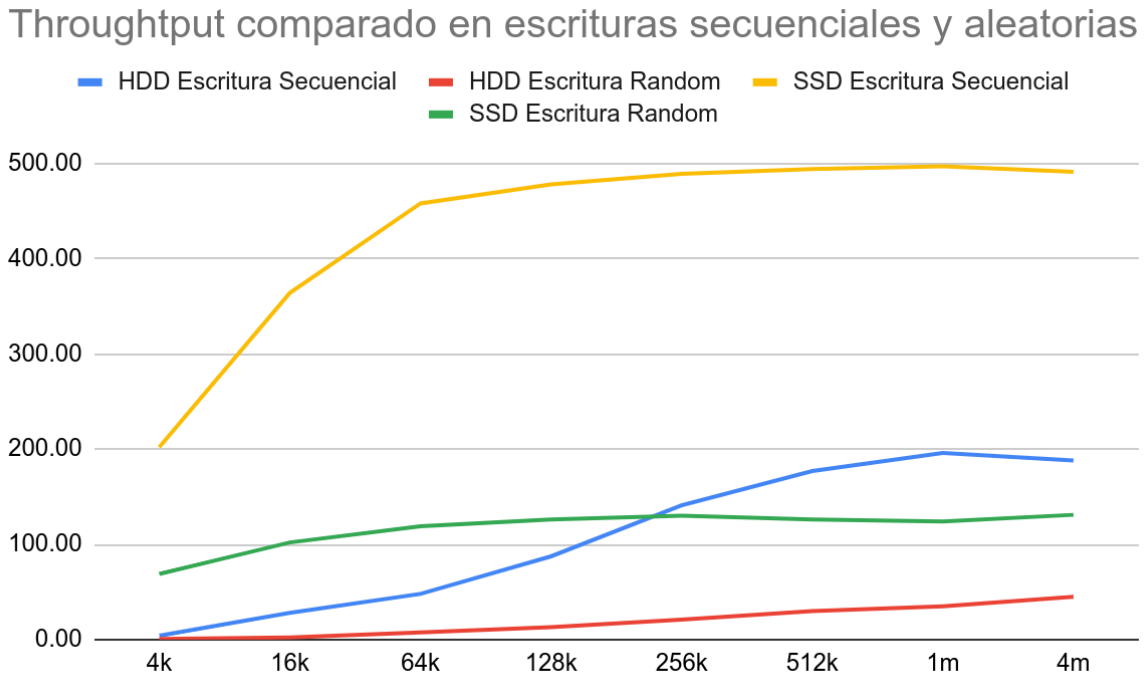


Figura 71 .Variación en número de lecturas o escrituras, conforme cambia el tamaño de bloque.

Con todas estas mediciones podemos inferir que un cluster armado sobre discos HDD va a tener buen tiempo de respuesta y una performance aceptable cuando el número de operaciones sobre cada OSD se mantenga bajo. La escalabilidad del mismo sobre discos HDD es menor que sobre los discos SSD.

En cambio los discos SSD van a mantener una buena performance aún cuando la carga sobre los OSD se incremente, y los tiempos de respuesta no deberían verse afectados, siempre y cuando los valores de operaciones concurrentes y procesos paralelos se mantengan bajo los niveles evaluados.

7. “One backend to rule them all”

En sus inicios y durante una década, Ceph continuó con la metodología tradicional de construir su infraestructura de almacenamiento distribuido sobre un filesystem local, como ext4 o XFS. La mayoría de los filesystems distribuidos actuales que mencionamos anteriormente continúan en ese camino porque les permite beneficiarse con la madurez y “consistencia” de un código que ya fue testeado y usado por muchos usuarios durante mucho tiempo.

No obstante, según mencionan Aghayev y Weil en su trabajo *File Systems Unfit as Distributed Storage Backends* (Aghayev, 2019), existe un precio que debe pagarse por tal decisión:

- Construir el mecanismo de transacciones que requiere el filesystem distribuido, con mínimo overhead, es realmente complejo. Cuando hablamos de transacciones distribuidas, resaltamos el hecho de que los datos deben escribirse en varios nodos de manera simultánea y consistente. Muchos desarrollos tratan de introducirlas, pero finalmente estos no son adoptados debido al alto overhead que afecta la performance, o a que proveen una funcionalidad limitada, o porque su interfaz e implementación son realmente complejas.

- La performance de la gestión de los metadatos a nivel local afecta significativamente la performance a nivel distribuido. La enumeración de directorios con millones de entradas y la devolución de estos valores de manera ordenada constituye un desafío para los filesystems locales.

- Adaptarse al nuevo hardware de almacenamiento les toma a estos filesystems tradicionales demasiado tiempo, pues sus algoritmos están diseñados para optimizar su performance pensando en discos, platos, pistas, cilindros, cabezales móviles… Los nuevos dispositivos, con tecnologías SMR (Castillo, 2020), funcionan mejor con interfaces *zoned* (Western Digital, 2020) (Bjørling, 2019). Sin embargo, cuando un filesystem se vuelve maduro, la rigidez inherente a ese estado los previene de adoptar ese nuevo hardware que abandonó la venerable interfaz de bloques...

Todos los filesystems distribuidos operan sobre un cluster de máquinas que poseen uno o más roles asignados y -aunque varían los nombres de acuerdo al sistema, las funcionalidades habituales engloban monitoreo, gestión de los metadatos y servidores de almacenamiento. Estos últimos reciben los pedidos de E/S de los clientes a través de la red y los atienden a través de los discos que tienen instalados localmente y del software de backend que los gestiona, que tradicionalmente fue XFS.

Esto permitía delegar los problemas de la persistencia de los datos y ubicación de bloques a un código bien testeado y performante, que ofrece una interfaz con la que todo el mundo está familiarizado (POSIX) y que permite utilizar herramientas estándar para su exploración, monitoreo y gestión (ls, find, etc…).

En retrospectiva, algunos trabajos que analizan la relación entre los sistemas operativos y el almacenamiento de bases de datos ya habían mencionado que "*los sistemas operativos ofrecen muchas cosas a costo de un gran overhead*" (Stonebraker, 1981) y otros trabajos sobre *exokernels*, indican que la personalización de las abstracciones tradicionales producen una mejora significativa en la performance (Engler, 1995).

Los filesystems distribuidos *agregan* el espacio de almacenamiento de múltiples nodos físicos en un espacio unificado que ofrece un mayor ancho de banda y E/S paralelas, lo que incide directamente en la gestión de los metadatos. A la vez, esta agregación permite la escalabilidad horizontal, la tolerancia a fallas y una consistencia fuerte que descansa en la implementación de las transacciones que encapsulan una secuencia de operaciones como una única unidad de trabajo, siguiendo el tradicional paradigma de bases de datos: ACID (*Atomicity, Consistency, Isolation, Durability*)

7.1. Bluestore

Por todo esto, Ceph comenzó desarrollando su propio backend de almacenamiento (Figura 72) y el último de ellos es *Bluestore*, diseñado para correr directamente sobre el dispositivo "*raw",* que ejecuta en espacio de usuario y controla el stack de E/S completo. Hace una gestión eficiente de los metadatos almacenándolos en una base de datos clave-valor, permite la reescritura rápida de los datos *erasure-coded* y posee algoritmos de compresión *inline*, entre muchas otras funcionalidades.

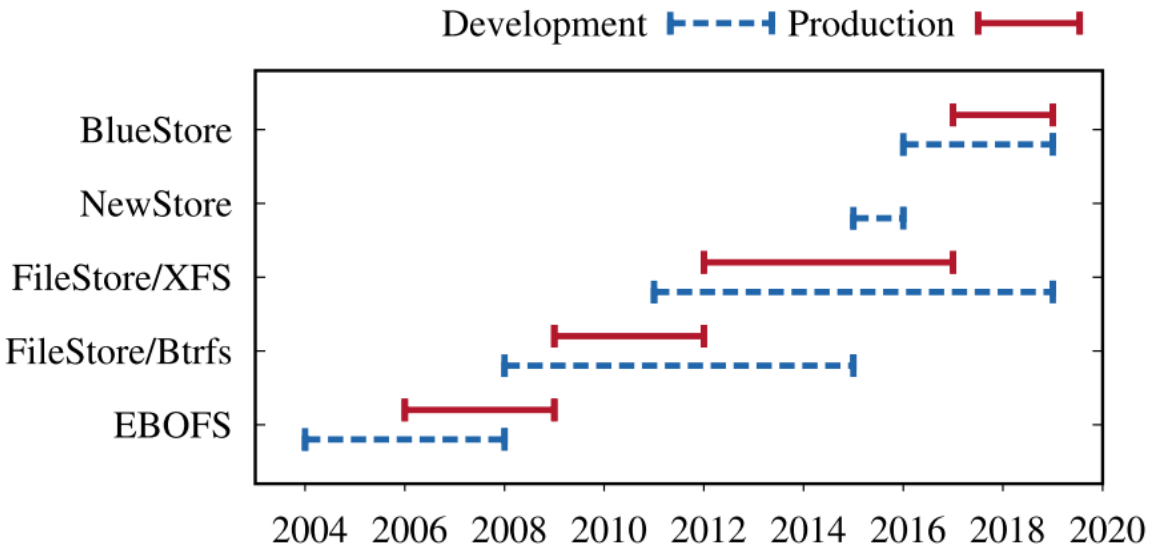


Figura 72. Desarrollo de los backends de Ceph.

En su investigación "*Understanding Write Behaviors of Storage Backends in Ceph Object Storage"* (Lee, 2017) los autores hacen un análisis exhaustivo del comportamiento de los principales backends de Ceph a través de microbenchmarks y comparación de extensas cargas de trabajo, enfocándose en el WAF (*Write Amplification Factor*) aplicado a

cuántos datos termina almacenando el sistema distribuido, por cada dato que quiere escribir el usuario: réplicas, erasure coding, metadatos, journals, logs, etc...

Filestore, por ejemplo, que por defecto utiliza XFS (Figura 73), sufre del problema llamado "*journaling of journal*" (Shen, 2014). Esta situación describe el hecho de que muchos sistemas distribuidos -entre ellos Ceph- manejan la consistencia y confiabilidad de sus datos a través de *write-ahead logs* o *rollback-recovery journaling* propios. Ahora, cuando XFS almacena esos datos y su respectivo log/journal, vuelve a generar una entrada en el journal del filesystem. Esto genera una sobrecarga al momento de escribir datos, y una merma importante en la performance.

En el trabajo anteriormente citado (Aghayev, 2019) puede verse un detalle de hitos y problemas encontrados en cada evolución, hasta llegar a BlueStore. Este último backend aborda los problemas mencionados y promete:

- Operaciones veloces sobre los metadatos.
- Elimina la sobrecarga en la escritura de objetos debido a la consistencia.
- Operaciones COW (*copy-on-write) (*GeeksforGeeks, 2020) que permiten la clonación instantánea de objetos.
- Elimina las doble escrituras de journals.
- Patrones optimizados de E/S para HDD y para SSD

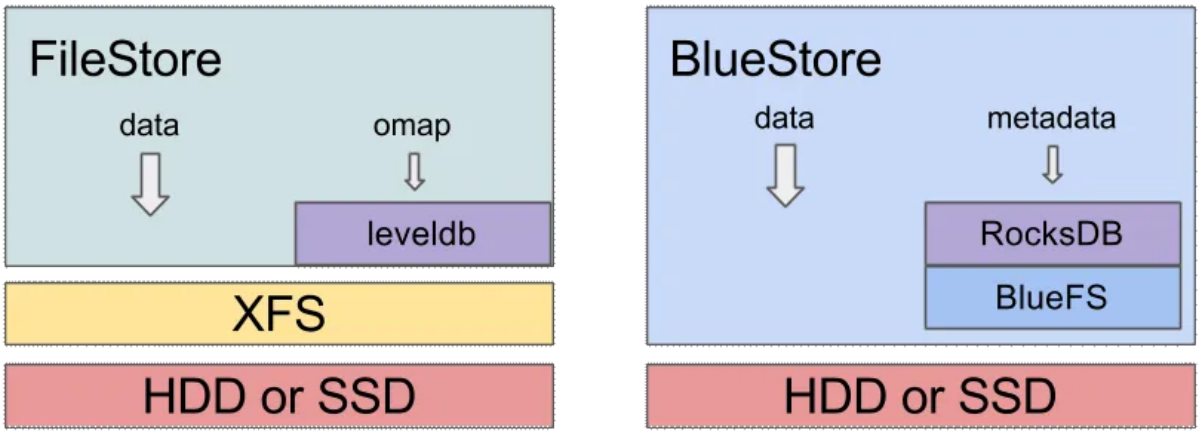


Figura 73. Comparación entre Filestore y Bluestore.

Bluestore funciona sobre discos sin formato. Un módulo interno determina la ubicación de los nuevos datos, que se escriben asincrónicamente en el disco utilizando *direct I/O*. Los metadatos internos propios de Ceph y los de los objetos de los usuarios se almacenan en RocksDB (RocksDB, 2020), una base de datos clave-valor de alta performance que corre sobre BlueFS, un filesystem mínimo que también se instala en el disco. Así, el *Bluestore space allocator* y *BlueFS* comparten el disco y periódicamente se comunican para balancear el espacio libre. RocksDB está basada en LSM-tree (*Log Structured Merge Tree*) (Groom, 2020): Los pares clave-valor primero se registran en el dispositivo WAL. No se graban inmediatamente en el storage, sino que se ponen temporariamente en un buffer llamado *memtable* y sólo cuando éste se llena, su contenido se vuelca al storage en bloque.

Bluestore utiliza más RAM que FileStore, porque todos los metadatos se almacenan en RocksDB, que cachea mucha información en memoria. Como regla sencilla, se recomienda 1GiB de RAM por TiB de almacenamiento, y nunca menos de 2 GiB por OSD.

Bluestore permite también instalarse de manera opcional hasta en 3 discos distintos para un mismo OSD, y separar datos, metadatos y journal. Cuando se dispone de discos HDD, una instalación alternativa que busca mejorar la performance, es instalar el WAL en discos SSD. (Bluestore Config Reference, 2020)

En palabras de sus desarrolladores, "*BlueStore provides a huge advantage in terms of performance, robustness, and functionality over our previous approach of layering over existing file systems.*" (Ceph New in Luminous, 2017*)*

Este backend es testeado y evaluado de manera comparativa en varios trabajos (Singh, K., & Parkes, D, 2019) (Meredith, 2018), y su performance es notoriamente superior respecto de sus predecesores, lo que justifica que sea el estándar de Ceph a partir del release *Luminous* (Oddeye, 2019).

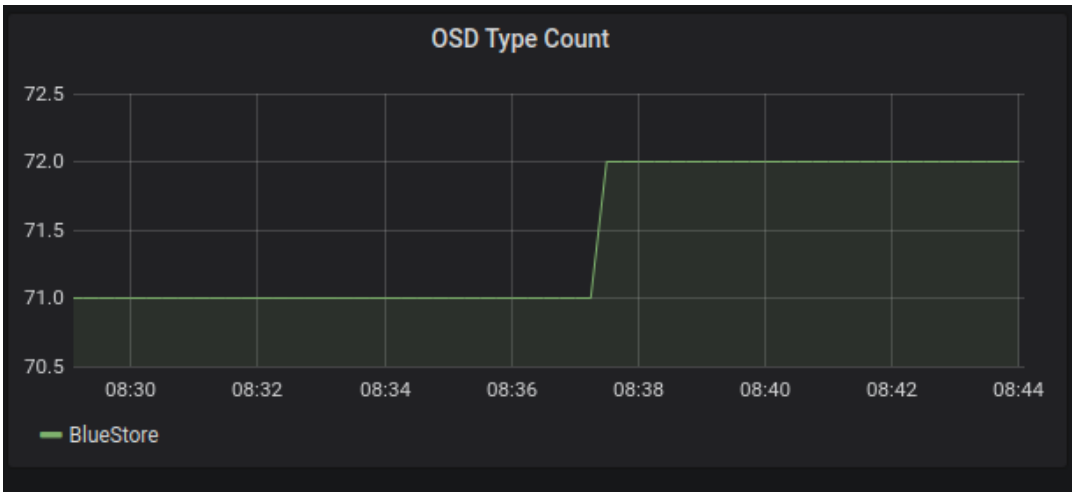
No obstante, el desarrollo nunca es en una única dirección y pueden verse trabajos como "*Accelerate Ceph via SPDK. XSKY's BlueStore as a case study"* (Wang, 2016) presentado en el Ceph Day de Beijing en 2016, o la presentación "Ceph Crimson. A new osd for the age of persistent memory and fast nvme storage" (Ceph Crimson, 2020) donde se plantean modificaciones a los algoritmos internos de Ceph y Bluestore para acompañar el crecimiento de las nuevas interfaces de almacenamiento.

Los OSDs del cluster de pruebas donde se desarrolla el presente trabajo **están todos instalados con Bluestore, configurados con un único disco por OSD**, sin separar WAL ni DB en otro dispositivo. (Figura 74)

```
#ceph-volume lvm create --bluestore --data /dev/sdg
[...]
/usr/bin/ceph-osd --cluster ceph --osd-objectstore bluestore --mkfs
-i 71 --monmap /var/lib/ceph/osd/ceph-71/activate.monmap --keyfile
- --osd-data /var/lib/ceph/osd/ceph-71/ --osd-uuid
6f802c2a-5855-421f-8b32-f629cb2331ea --setuser ceph --setgroup
ceph
```

Figura 74. Creación de un disco OSD.

Al ingresar el nuevo OSD, el cluster actualiza el mapa CRUSH y dispara operaciones "recovery" (Figura 75) para re-balancear los *Placement Groups*, transfiriendo algunos al nuevo disco.



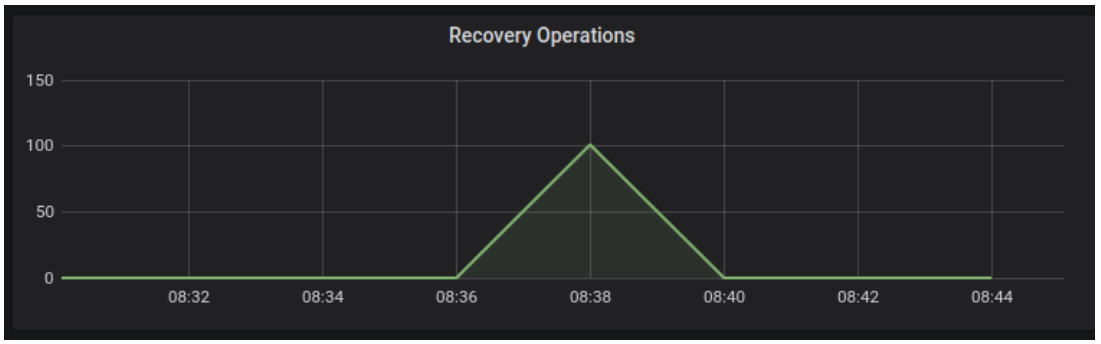


Figura 75. Ingreso de nuevo OSD provoca rebalanceo de placement groups.

Se verifica (Figura 76) el estado del nuevo OSD una vez terminada la operación de *backfilling,* donde también muestra que el nuevo OSD almacena 10 PGs, y que es la copia primaria de 3.

```
# ceph osd df
ID  CLASS  WEIGHT   REWEIGHT  SIZE     RAW USE  DATA     OMAP     META      AVAIL    %USE   VAR   PGS  STATUS
[...]
71    ssd  1.74660   1.00000  1.7 TiB   12 GiB   11 GiB      0 B     1 GiB  1.7 TiB   0.68  0.09   10      up

# ceph pg dump
[...]
71 ...     [0,2,4,8,10,17,22,31,34,37,42,45,48,51,52,59,62,70]         10    3
```

Figura 76. Verificación del nuevo OSD.

A modo de comentario, vale mencionar que al utilizar Bluestore como backend, los comandos tradicionales (Figura 77) ya no son de utilidad para explorar el dispositivo de bloques OSD.

```
root@stor12:/var/lib/ceph/osd/ceph-71# ls -all
total 56
drwxrwxrwt 2 ceph ceph  320 nov 15 08:37 .
```

```
drwxr-xr-x 9 ceph ceph 4096 nov 15 08:36 ..
-rw-r--r-- 1 ceph ceph  406 nov 15 08:36 activate.monmap
lrwxrwxrwx 1 ceph ceph   93 nov 15 08:37 block ->
/dev/ceph-64499086-8bad-4a73-9812-94e203dd47c8/osd-block-6f80
2c2a-5855-421f-8b32-f629cb2331ea
-rw------- 1 ceph ceph    2 nov 15 08:36 bluefs
-rw------- 1 ceph ceph   37 nov 15 08:37 ceph_fsid
-rw-r--r-- 1 ceph ceph   37 nov 15 08:37 fsid
-rw------- 1 ceph ceph   56 nov 15 08:37 keyring
-rw------- 1 ceph ceph    8 nov 15 08:36 kv_backend
-rw------- 1 ceph ceph   21 nov 15 08:37 magic
-rw------- 1 ceph ceph    4 nov 15 08:37 mkfs_done
-rw------- 1 ceph ceph   41 nov 15 08:37 osd_key
-rw------- 1 ceph ceph    6 nov 15 08:37 ready
-rw------- 1 ceph ceph    3 nov 15 08:37 require_osd_release
-rw------- 1 ceph ceph   10 nov 15 08:37 type
-rw------- 1 ceph ceph    3 nov 15 08:37 whoami

# cd block
bash: cd: block: No es un directorio
```

Figura 77. Los comandos tradicionales no sirven para ver los datos almacenados en un OSD.

En resumen, los objetos no se almacenan en el filesystem tradicional, sino en una base de datos embebida que utiliza cada OSD directamente. Esto mejora la performance al saltearse el overhead que impone el filesystem. Al implementar un almacenamiento subyacente desarrollado acorde a sus necesidades específicas y sin tener que "contorsionarse" para adaptarse y encajar en los límites y semánticas de un sistema de archivos diseñado para otros propósitos, Bluestore gestiona los datos con mucha más eficiencia.

Pero… las herramientas tradicionales con las que se interactúa con un filesystem en linux (ls, cd, cat, du, etc…) no sirven para analizar un OSD al momento de resolver problemas, porque ahora utiliza una RocksDB.

7.2. Benchmarking de los OSDs

Ceph incluye una herramienta para testear los OSDs individualmente:

```
# ceph tell osd.N bench [TOTAL_DATA_BYTES] [BYTES_PER_WRITE]
```

que ejecuta un testeo simple contra el osd.N, escribiendo TOTAL_DATA_BYTES en solicitudes de escritura de BYTES_PER_WRITE cada una. Por defecto, el testeo escribe 1 GiB de datos totales, en objetos de 4 MiB. Este testeo es no-destructivo y no sobreescribe los datos de los usuarios que están en el disco, aunque puede afectar la performance de los clientes que estén accediendo al dispositivo en ese preciso momento.

Para obtener resultados consistentes y reales, se debe limpiar las cachés de los OSD entre testeos de benchmark. Ver la operatoria en la Figura 78

```
# ceph tell osd.71 cache status
{
    "object_ctx": 68,
    "bluestore_onode": 5641,
    "bluestore_buffers": 0
}
# ceph tell osd.71 cache drop
# ceph tell osd.71 cache status
{
    "object_ctx": 0,
    "bluestore_onode": 0,
    "bluestore_buffers": 0
}
```

Figura 78. Limpieza de cachés antes de testear los OSD.

En nuestro cluster, vamos a realizar los testeos con escrituras de 2 GiB, sobre los discos HDD y sobre los SSD.

En la Figura 79 se muestra la prueba sobre disco SSD del OSD 71:

```
#ceph tell osd.71 cache drop
#ceph tell osd.71 bench 2147483648
{
    "bytes_written": 2147483648,
    "blocksize": 4194304,
    "elapsed_sec": 4.4240765980000001,
    "bytes_per_sec": 485408333.33916885,
    "iops": 115.73036511878225
}
```

Figura 79. Prueba de performance del OSD 71 (SSD)

Prueba sobre el disco HDD del OSD 29 del nodo Storage6 (Figura 80)

```
# ceph tell osd.29 cache drop
# ceph tell osd.29 bench 2147483648
{
    "bytes_written": 2147483648,
    "blocksize": 4194304,
    "elapsed_sec": 13.837657369,
    "bytes_per_sec": 155191271.95698091,
    "iops": 37.000482548947552
}
```

Figura 80. Prueba de performance del OSD 29 (HDD)

Como se muestra en la Figura 81, el SSD tardó 4.42 segundos en escribir 2 GiB, en bloques de 4 MiB, alcanzando un throughput de 462 MiB/s y 115 IOPS. Al HDD le tomó 13.84 segundos en realizar la misma tarea, con un throughput de 148MiB/s y 37 IOPS

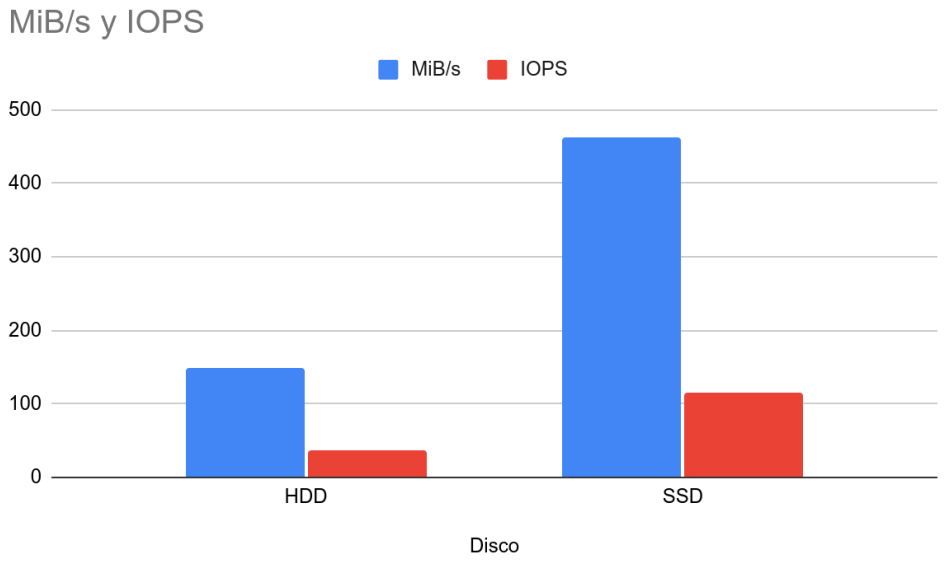


Figura 81. Comparativa de performance de OSDs sobre discos SSD y HDD.

Con esto, observamos que el backend, aunque optimizado y diseñado para obtener la mejor performance, en los HDD no llega a los valores obtenidos en el testeo de los discos “en crudo” realizado. En los discos SSD, no obstante, los resultados son muy similares, lo que demuestra que el overhead del backend es mínimo en este caso.

Un estudio más detallado puede analizarse en el trabajo *Characterization of OSD performance in a Ceph cluster* de Daniel van der Ster y Julien Collet hecho sobre el CERN (Collet, 2019).

8. BENCHMARKING en RADOS y LIBRADOS

Como se detalló en el capítulo 4, Ceph utiliza varias capas de software (Figura 82) para proveer las características de distribución, confiabilidad, disponibilidad y consistencia que hacen al sistema. Y, como es obvio suponer, cada capa afecta la performance general del cluster, agregando latencia y disminuyendo el throughput.

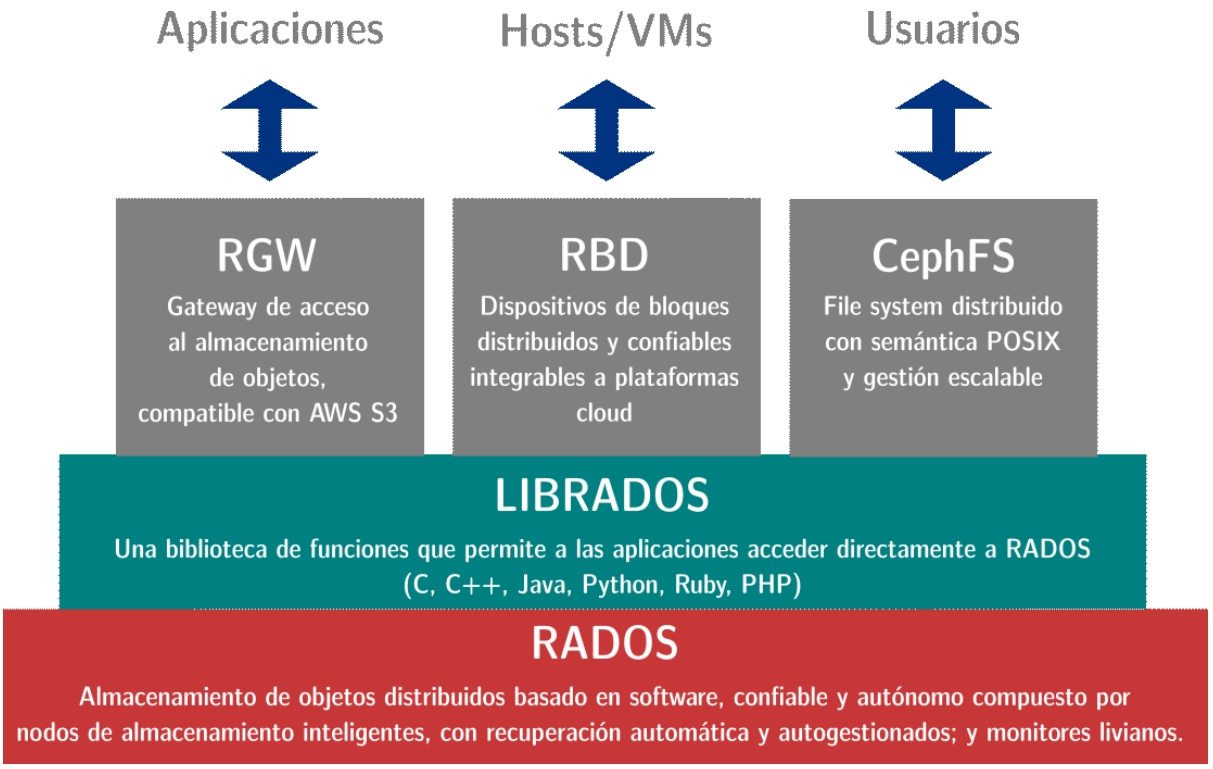


Figura 82. Arquitectura de capas de Ceph.

8.1. RADOS

Vamos ahora a testear la performance de la capa RADOS y LIBRADOS. En su trabajo *"Supporting Heterogeneous Pools in a Single Ceph Storage Cluster"*, (Meyer, 2016) los autores plantean que las aplicaciones tienen diferentes cargas de trabajo y patrones de acceso sobre el almacenamiento, y que normalmente el filesystem distribuido provee a esa variedad de aplicaciones una solución bastante homogénea y diseñada para un escenario "genérico", lo que afecta la performance en varios casos. Así, dependiendo de la aplicación, la métrica que tendría mayor importancia puede ser el throughput mientras que en otras podría ser la latencia promedio, o la latencia máxima, o una mezcla de todas.

La posibilidad de ajustar las configuraciones a la hora de definir los pools, sumado a la capacidad de definir cachés (*tiers)*, permite cierto grado de flexibilidad y de adaptación del almacenamiento a la aplicación que lo va a utilizar.

Inicialmente, vamos a realizar los benchmarks sobre pools sin caché. Para llevar adelante el testeo de performance, vamos a utilizar los 3 pools de almacenamiento descriptos en el capítulo 5:

- testhdd, replicated, r=3
- testSSD, replicated, r=3
- testEC, erasure-coding, k=5, m=3

Cada uno de estos pools tiene 32 *placement groups,* distribuidos en distintos OSDs. Los PG del pool testhdd están únicamente en discos HDD, y de igual manera los PGs del pool testssd están sobre discos SSD. Los PGs del pool testEC (erasure-coded) utilizan 8 OSD HDD, 5 para almacenar un fragmento del PG, y 3 como paridad/redundancia.

Así, utilizando el comando "`ceph osd lspools`" podemos ver el ID asignado a cada pool, y con "`ceph pg dump pgs`" tenemos un detalle de cada placement group. En nuestro caso, el pool *testhdd* tiene el ID=15, y si miramos el Placement Group 15.0 (primero del pool) vemos que está almacenado en los OSD `[9,18,23]` y su copia primaria está en el 9. El pool *testssd* tiene ID=21 y el acting set del PG 21.0 es [**57,**`46,38]` con su copia primaria en el OSD 57. El pool *testec* tiene ID=22 y el PG 22.0 tiene acting set `[23,17,21,5,19,26,8,14]` para k+m=8.

Los testeos se realizan con la herramienta *rados bench* (Figura 83):

```
# rados bench -p POOL <seconds> write|seq|rand [-t
concurrent_operations] [--no-cleanup] [--run-name run_name]
[--no-hints] [--reuse-bench]
     default run-name is 'benchmark_last_metadata'
     default is 16 concurrent IOs and 4 MB ops
     default is to clean up after write benchmark
```

Figura 83. Opciones de la herramienta *rados bench*.

Rados bench puede utilizarse para medir la performance del cluster a nivel de pools, y soporta lecturas y escrituras secuenciales, y lecturas aleatorias. Por defecto, cuando termina, limpia los objetos creados para las pruebas. Esta herramienta escribe objetos en el almacenamiento subyacente tan rápido como puede, y luego los lee en el mismo orden en que los escribió.

Se prueba la escritura secuencial sobre pool testhdd durante 60 segundos, sin limpiar los datos para poder hacer después las pruebas de lectura. (Figura 84)

```
# rados bench -p testhdd 60 write --no-cleanup
[...]
Total time run:         60.1257
Total writes made:      5948
Write size:             4194304
Object size:            4194304
Bandwidth (MB/sec):     395.705
Stddev Bandwidth:       24.2637
Max bandwidth (MB/sec): 440
Min bandwidth (MB/sec): 344
Average IOPS:           98
Stddev IOPS:            6.06593
```

```
Max IOPS:               110
Min IOPS:               86
Average Latency(s):     0.161603
Stddev Latency(s):      0.0641304
Max latency(s):         0.749691
Min latency(s):         0.060445
```

Figura 84. Testeo de escrituras secuenciales de objetos RADOS.

Lecturas secuenciales sobre los datos generados en la prueba de escritura. (Figura 85)

```
# rados bench -p testhdd 60 seq
[...]
Total time run:       44.5068
Total reads made:     5948
Read size:            4194304
Object size:          4194304
Bandwidth (MB/sec):   534.57
Average IOPS:         133
Stddev IOPS:          3.11742
Max IOPS:             142
Min IOPS:             125
Average Latency(s):   0.118933
Max latency(s):       0.511
Min latency(s):       0.0352361
```

Figura 85. Testeo de lecturas secuenciales de objetos RADOS.

Puede notarse que el testeo de lectura secuencial terminó en 44 segundos, no en los 60 requeridos. Esto se debe a que la velocidad de lectura es superior a la de escritura y que el proceso *rados bench* terminó de leer todos los datos generados durante la prueba de escritura. Recordar que las escrituras en este pool se hacen en 3 OSD, y que la lectura sólo se realiza sobre la copia primaria del objeto.

Se prueba (Figura 86) la respuesta del pool con lecturas aleatorias sobre los datos generados en la prueba de escritura y se repiten las mismas 3 pruebas sobre el pool testSSD y el pool testEC, arrojando los resultados que se muestran en la Figura 87.

```
# rados bench -p testhdd 60 rand
Total time run:       60.1029
Total reads made:     7592
Read size:            4194304
Object size:          4194304
Bandwidth (MB/sec):   505.267
Average IOPS:         126
Stddev IOPS:          6.32
Max IOPS:             139
Min IOPS:             113
Average Latency(s):   0.125892
Max latency(s):       0.607919
Min latency(s):       0.0090957
```

Figura 86. Testeo de lecturas aleatorias de objetos RADOS.

TESTHDD	Bandwidth (MiB/s)	IOPS	Latencia (ms)
write	385	98	161
read	534	142	118
random	505	139	126

TESTSSD	Bandwidth (MiB/s)	IOPS	Latencia (ms)
write	1083	270	59
read	563	140	112
random	560	140	113

TESTEC	Bandwidth (MiB/s)	IOPS	Latencia (ms)
write	428	107	149
read	557	145	114
random	523	130	121

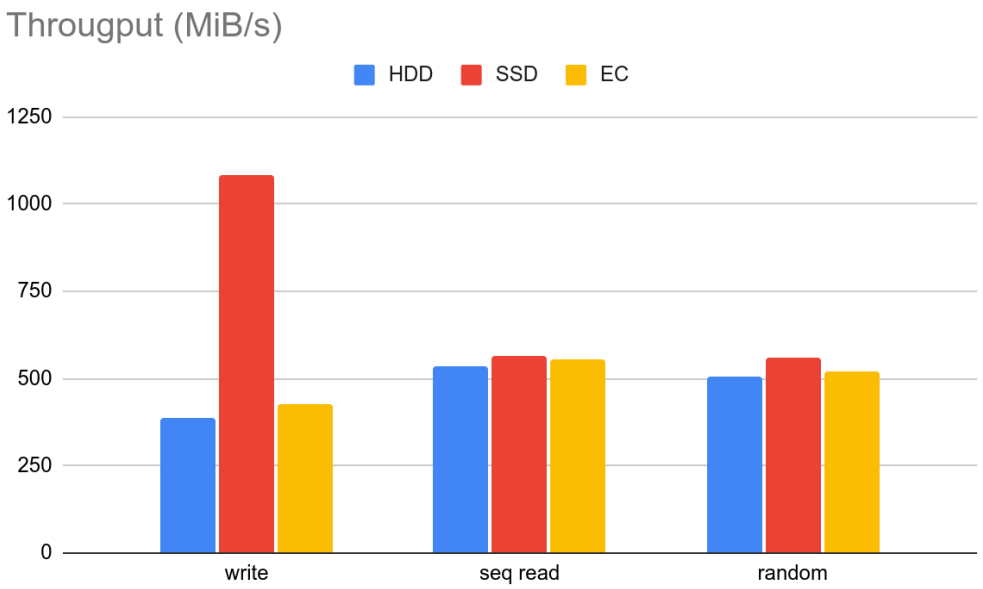


Figura 87. Resultados de pruebas sobre objetos RADOS.

Podemos observar una performance superior en la escritura del pool con discos SSD, mientras que en el resto de las operaciones, los valores obtenidos son similares.

En el el sistema de monitoreo, puede verse el tráfico registrado en los nodos que tienen los OSDs afectados por la prueba, mientras que los otros están casi inactivos (Figura 88).

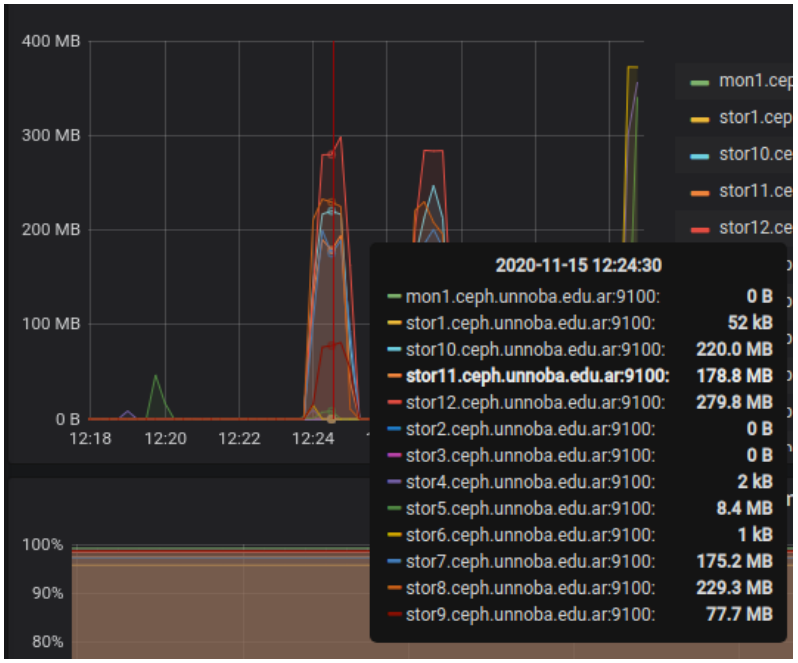


Figura 88. Monitoreo del tráfico en los nodos durante las pruebas.

8.2. Cache Tiering

La documentación oficial de Ceph (Ceph, 2020), presenta una opción para mejorar la performance de E/S de los clientes, introduciendo en el diseño una capa que actúa como caché. Esta capa se implementa creando un pool sobre dispositivos rápidos (y caros) como pueden ser SSD o NVMe, mientras que el almacenamiento se realiza en un pool sobre dispositivos más lentos, como los HDD. El backend determina dónde ubicar los objetos y el agente de *tiering* gestiona la migración de datos entre la caché y el almacenamiento de manera automática.

Esta caché puede configurarse con dos tipos de comportamientos:

modo writeback: Cuando los clientes **escriben** datos en el cluster, obtienen un ACK cuando esos datos están en la caché. En el momento en que el agente de *tiering* lo determina, migra los datos a la capa de almacenamiento. Cuando un cliente hace una solicitud de **lectura** de datos que están en el almacenamiento, el agente migra esos datos a la capa de caché y luego se los envía al cliente. Así, el cliente SIEMPRE interactúa con la caché (Figura 89), hasta que el dato se vuelve inactivo. Este modo es óptimo para datos que sufren cambios con frecuencia.

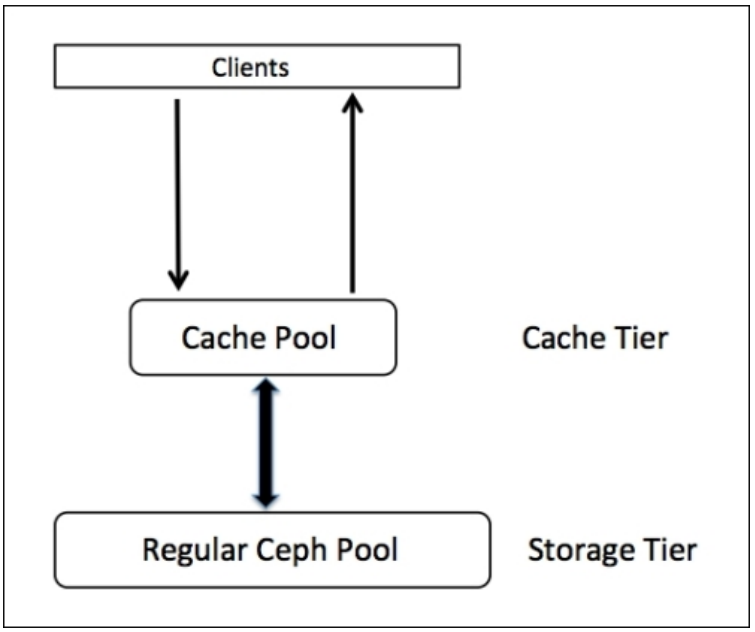


Figura 89. Esquema de interacción con cache writeback

modo readproxy: Cualquier objeto que está en la caché puede utilizarse, pero si un objeto no está presente en el caché, el pedido se redirige a la capa de almacenamiento, sin agregarlo a la caché. Este modo se utiliza cuando se necesita "vaciar" una caché, sin agregar objetos nuevos, en una transición hacia el modo deshabilitado, sin interrumpir el servicio. Se utiliza cuando se quiere eliminar la capa de caché de un pool de modo transparente para los usuarios.

El modo readonly es experimental y los objetos de la caché son de sólo lectura: cualquier operación de escritura se redirige al pool de almacenamiento. Esto -por supuesto- tiene implicancias a nivel de consistencia, pues los objetos se actualizan en el almacenamiento, no en la caché.

Pero no todo lo que brilla es oro. La documentación se asegura de mencionar que la capa de caché no es la "solución mágica" que funciona para todos los patrones de acceso y cargas de trabajo. Habilitar la caché significa agregar mucha complejidad en los procesos e incrementar la probabilidad de encontrar bugs en el código que afecte todo el sistema.

La caché va a representar una mejora en la performance, siempre y cuando la carga de trabajo no sea excesiva, porque hay un overhead inherente a mover objetos entre pools, entrando y saliendo de la caché. El tamaño del pool destinado a caché debería ser el suficiente como para poder almacenar todos los objetos del *working set* de la carga de trabajo habitual de la aplicación. De lo contrario, se produciría una suerte de "*thrashing*": el cluster invertiría más tiempo moviendo objetos entre capas que procesando requerimientos de los usuarios. Si la carga de trabajo no está diseñada para ser cacheada, la performance va a ser menor que un pool común, sin caché.

Además, un diseño con caché es difícil de medir: las herramientas para benchmark van a obtener un *miss* en todos los requests iniciales de lectura, porque la caché está vacía.

Esto va a mostrar una mala performance, porque llenar la caché implica ir a buscar los objetos al almacenamiento subyacente.

Algunas cargas de trabajo ya han sido probadas y obtienen buenos resultados: acceso a objetos RGW, en los que se leen objetos que han sido escritos recientemente.

Otras, obtienen **pobres resultados** con la caché habilitada (Ceph, 2020) (Umrao, 2017) (Meyer, 2016): volúmenes RBD con caché replicada y almacenamiento en pool erasure-coded, y volúmenes RBD con caché y base replicada. Justamente las que nosotros estamos probando en este trabajo, porque son los usos definidos para el cluster implementado.

La modificación de muchos objetos de un volumen RBD implica tenerlos en la caché, y luego bajarlos al almacenamiento. Si los objetos afectados son irregulares, la performance se degrada mucho porque todo el tiempo tiene que ir a buscarlos al pool de almacenamiento para tenerlos disponibles en la caché. Conviene directamente hacer un pool para estos volúmenes en los discos rápidos en los que se intenta implementar el pool de caché.

No obstante, vamos a hacer algunas pruebas de performance, agregando el pool testssd como caché writeback del pool testhdd. (Figura 90)

```
# ceph osd tier add testhdd testssd
pool 'testssd' is now a tier of 'testhdd'

# ceph osd tier cache-mode testssd writeback
set cache-mode for pool 'testssd' to writeback

# ceph osd tier set-overlay testhdd testssd
overlay for 'testhdd' is now 'testssd'
```

Figura 90. Configuración de un pool como caché de otro.

La caché debe ser configurada con algunos parámetros (Umrao, 2017) que afectan el comportamiento del pool y también tiene implicaciones en la RAM y CPU que utilizan los OSD que son afectados por ese pool. (Figura 91)

```
# ceph osd pool set testssd hit_set_type bloom
# ceph osd pool set testssd hit_set_count 1
# ceph osd pool set testssd hit_set_period 300
# ceph osd pool set testssd target_max_bytes 5368709120       //5GiB
# ceph osd pool set testssd min_read_recency_for_promote 1
# ceph osd pool set testssd min_write_recency_for_promote 1
# ceph osd pool set testssd cache_min_flush_age 300
# ceph osd pool set testssd cache_min_evict_age 300
# ceph osd pool set testssd cache_target_dirty_ratio .2
# ceph osd pool set testssd cache_target_full_ratio .7
```

Figura 91. Valores de configuración del pool que funciona como caché.

En la Figura 92 se genera un archivo de 100 MiB, se carga en el pool testhdd y se verifica que está presente en ambos pools (almacenamiento y caché).

```
# dd if=/dev/zero of=archivo100 bs=1M count=50
100+0 registros leídos
100+0 registros escritos
104857600 bytes (105 MB, 100 MiB) copied, 0,0795434 s, 1,3 GB/s
# rados -p testhdd put objeto1 archivo100
# rados -p testhdd ls
[...]
objeto1
# rados -p testssd ls
[...]
objeto1
```

Figura 92. Verificación del objeto en almacenamiento y caché.

Repetimos el mismo comando (Figura 93) introducido previamente sobre el poolHDD sin caché. Obtenemos los resultados de la (Figura 94), y graficamos la comparativa de las escrituras secuenciales (Figura 95), lecturas secuenciales (Figura 96) y lecturas random (Figura 97) para verificar si se registra una mejora en la performance.

```
# rados bench -p testhdd 60 write --no-cleanup
[...]
Total time run:         60.1248
Total writes made:      5194
Write size:             4194304
Object size:            4194304
Bandwidth (MB/sec):     345.548
Stddev Bandwidth:       25.423
Max bandwidth (MB/sec): 400
Min bandwidth (MB/sec): 268
Average IOPS:           86
Stddev IOPS:            6.35574
Max IOPS:               100
Min IOPS:               67
Average Latency(s):     0.185056
Stddev Latency(s):      0.0714683
Max latency(s):         0.559082
Min latency(s):         0.0691094
```

Figura 93. Comando para testeo del pool con caché habilitada.

Repetimos la misma configuración del testeo para lecturas secuenciales y random.

	Operación	Bandwidth (MiB/s)	IOPS	Latencia (ms)
Sin caché	write	385	98	161
	read	534	142	118
	random	505	139	126
Con caché	write	345	86	185
	read	532	133	119
	random	548	137	115

Figura 94. Valores obtenidos en los testeos del pool con y sin caché.

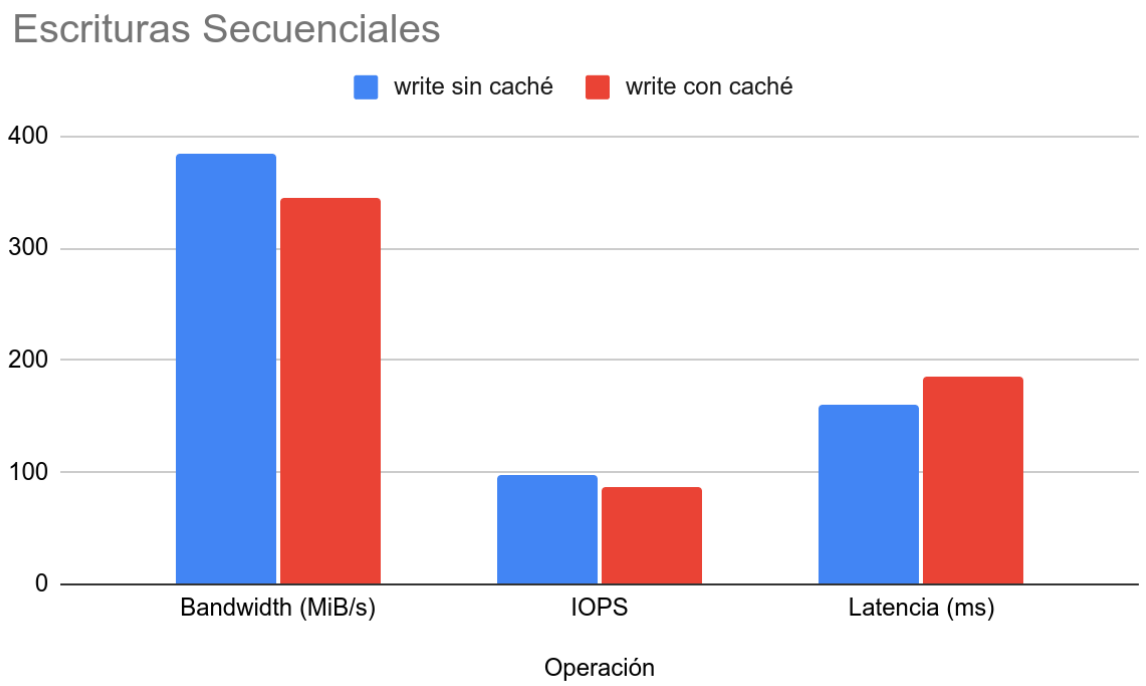


Figura 95. Testeos de Escrituras secuenciales sobre poolHDD con y sin caché

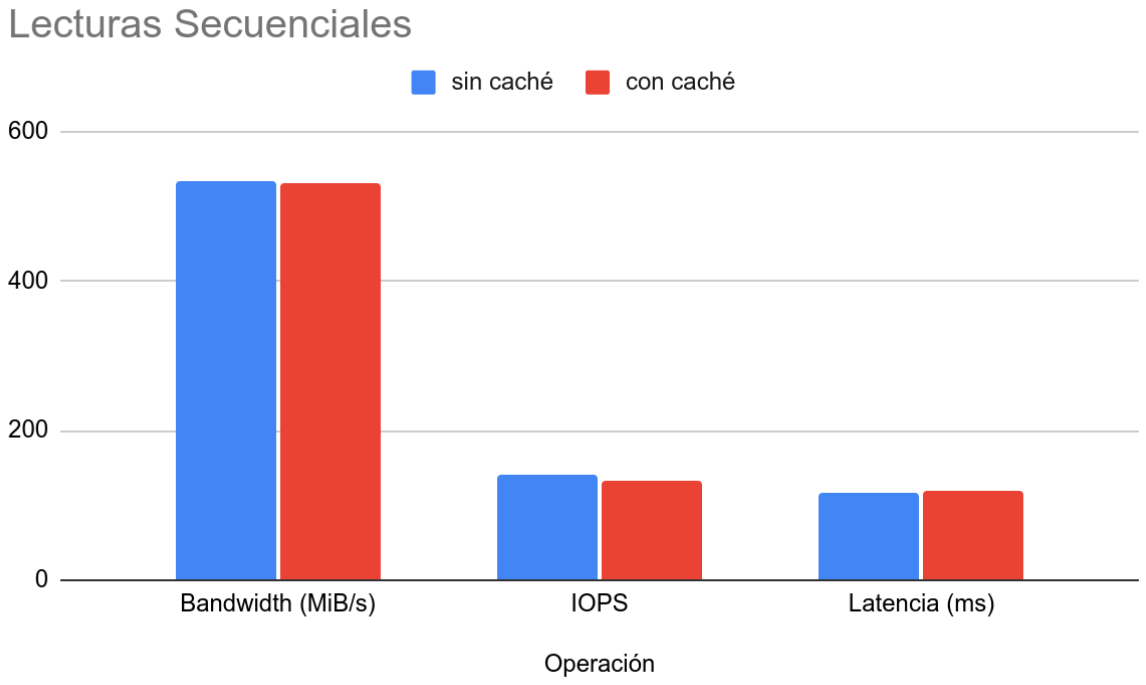


Figura 96. Testeos de Lecturas secuenciales sobre poolHDD con y sin caché

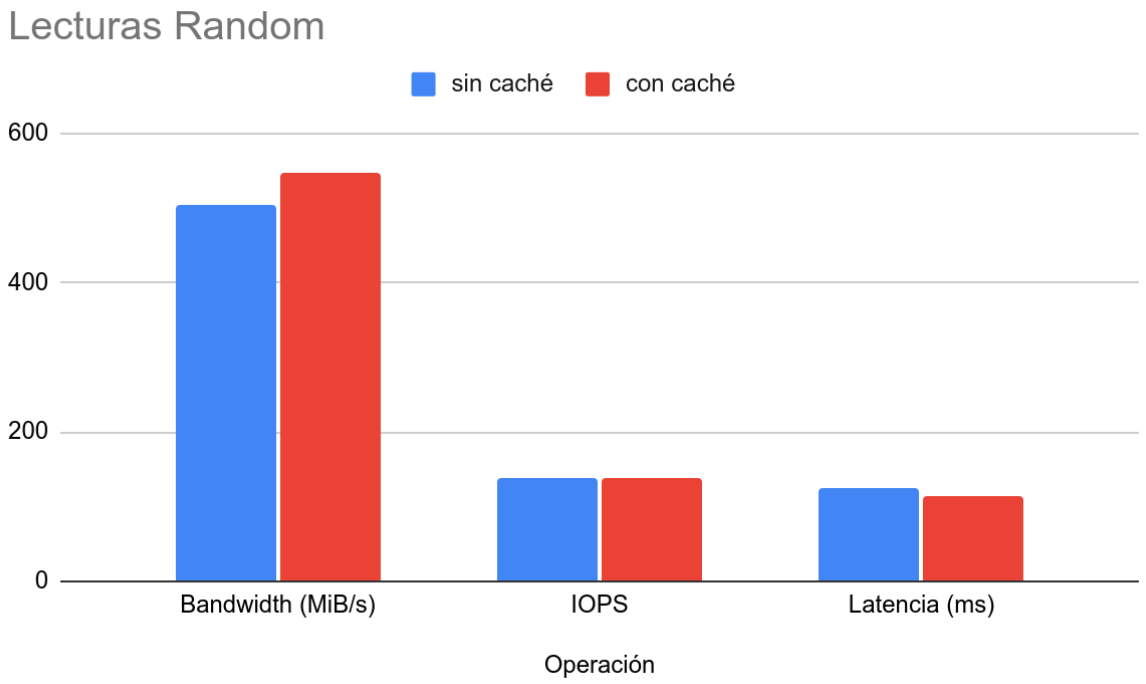


Figura 97. Testeos de Lecturas random sobre poolHDD con y sin caché

La única mejora en la performance que observamos es en las lecturas random, lo cual es consistente con la arquitectura de discos SSD que subyace al pool de caché, que carece de las altas latencias de los discos HDD provocadas por el movimiento continuo de brazo para encontrar los objetos aleatorios.

8.3. Compresión

El backend Bluestore soporta compresión *inline* utilizando distintos algoritmos como snappy, zlip, lz4. La elección de qué datos se comprimen y en qué momento depende de distintos parámetros y modos de configuración, que pueden resumirse en:

- **none**: Sin compresión.
- **passive**: No comprime los datos a no ser que la operación produzca una reducción importante.
- **aggressive**: Comprime los datos, a no ser que la operación sea inútil.
- **force**: Comprime los datos sin hacer ninguna estimación.

No obstante el modo elegido, si el tamaño del fragmento de datos no se reduce lo suficiente, no va a ser utilizado y en su lugar se almacena el dato original sin comprimir. Si el ratio de compresión se establece en .7, el dato comprimido debe ser igual o menor al 70% del tamaño del objeto original.

Esta operación, que busca el beneficio de lograr mayor capacidad de almacenamiento, implica un overhead de procesamiento y una degradación en la performance. El trabajo “*Bluestore compression performance*” (Singh, 2019B) realiza un

detallado análisis del impacto de la compresión en la latencia, el throughput, el uso de CPU y RAM de los OSD y concluye que comparados con los pools sin compresión, los pools con la compresión habilitada mostraron únicamente un 10% de reducción en la performance en una carga de trabajo generada utilizando la herramienta *fio* y un 7% en una carga de trabajo generada por MySQL. (Figura 98)

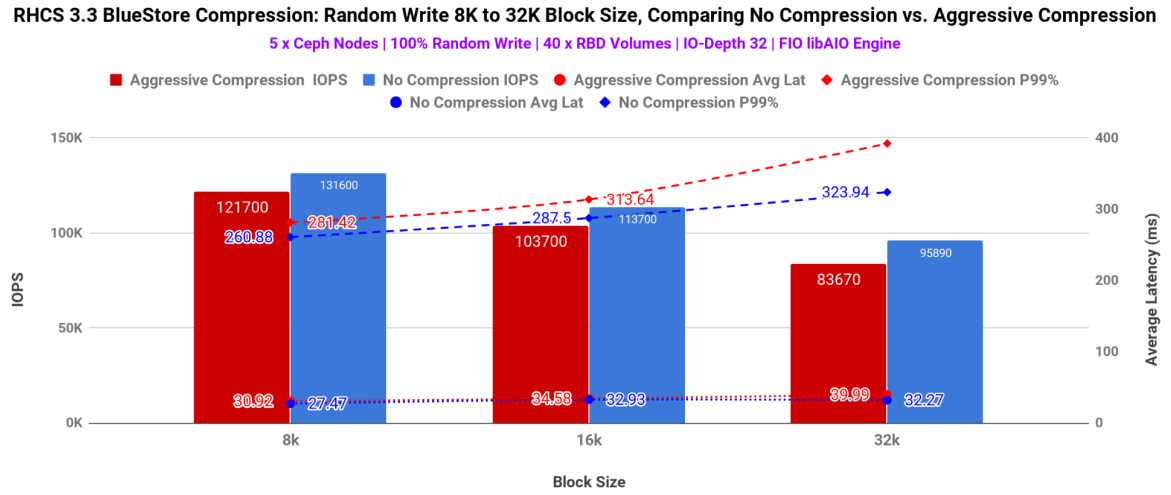


Figura 98. Comparativa de performance en IOPS y latencia, con y sin compresión.

Si el tamaño de los bloques es pequeño (8 KiB), la diferencia en el uso de la CPU entre el modo aggressive y el deshabilitado es menor que si se incrementa el tamaño de bloques a 16 KiB, 32 KiB y 1 MiB.

8.4. Performance de los dispositivos de Bloque

RBD *(Rados Block Device)* es el nombre que recibe el dispositivo de bloques en Ceph, que constituye una secuencia consecutiva de bytes dividido en objetos de 4MiB. Cuando un cliente modifica una región del RBD, los objetos RADOS correspondientes se fragmentan en forma de tiras (stripe) y se replican en los otros nodos del cluster y en varios PGs, para obtener mejor performance.

La Figura 99, tomada del libro “Ceph Cookbook. Second Edition” (Umrao, 2017), muestra que RBD es una interfaz *iscsi-like* que está implementada en dos bibliotecas: **librbd** y **krbd**: la primera para utilizarse a nivel de usuario y la segunda como módulo del kernel, que exporta *devices* al usuario para ser formateado y utilizado como un disco físico local.

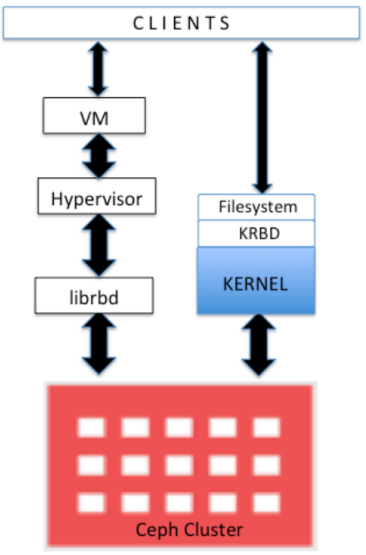


Figura 99. librbd y krbd

Para evaluar la performance de los RBD, puede utilizarse la herramienta *fio*, que trae soporte para rbd, o *rbd bench* que viene incluída con Ceph. (Figura 100)

```
rbd bench --io-type <read | write | readwrite | rw> [--io-size
size-in-B/K/M/G/T] [--io-threads num-ios-in-flight] [--io-total
size-in-B/K/M/G/T] [--io-pattern seq | rand] [--rw-mix-read read
proportion in readwrite] image
```

Figura 100. Opciones del comando rbd bench

Este comando genera una serie E/S a la imagen y mide el throughput y la latencia. Los valores por defecto son objetos de 4MiB, 16 threads, 1GiB de IO Total, patrón secuencial.

Vamos a generar una imagen en los dos primeros pools para realizar las pruebas, con los comandos descriptos en la Figura 101

```
# rbd create imagen1 --size 2048 --pool testhdd
# rbd create imagen1 --size 2048 --pool testssd
# rbd feature disable testhdd/imagen1 object-map fast-diff
deep-flatten
# rbd feature disable testssd/imagen1 object-map fast-diff
deep-flatten
```

Figura 101. Creación de imágenes rbd para testeos.

Testeamos el RBD sobre el pool de discos HDD, replica = 3. (Figura 102)

```
# rbd bench --io-type=write --io-size=4M --io-total=2G
--io-pattern=seq testhdd/imagen1
[...]
elapsed: 6   ops: 512   ops/sec: 80.1475   bytes/sec: 321 MiB/s
# rbd bench --io-type=read --io-size=4M --io-total=2G
--io-pattern=seq testhdd/imagen1
[...]
elapsed: 3   ops: 512   ops/sec: 155.334   bytes/sec: 621 MiB/s
# rbd bench --io-type=readwrite --io-size=4M --io-total=2G
--io-pattern=seq testhdd/imagen1
[...]
elapsed: 4   ops: 512   ops/sec: 124.874   bytes/sec: 499 MiB/s
#rbd bench --io-type=readwrite --io-size=4M --io-total=2G
--io-pattern=rand testhdd/imagen1
[...]
elapsed: 3   ops: 512   ops/sec: 136.602   bytes/sec: 546 MiB/s
```

Figura 102. Testeos sobre la imágen del pool HDD.

Repetimos la operación sobre las otras imágenes y mostramos los resultados obtenidos en la Figura 103.

TESTHDD	IOPS	Throughput
write seq	80	321
read seq	155	621
readwrite seq (50/50)	132	499
readwrite rand	136	546
TESTSSD	IOPS	Throughput
write seq	210	842
read seq	158	632
readwrite seq (50/50)	247	988
readwrite rand	252	1010

Figura 103. Resultados obtenidos en los pools testHDD y testSSD.

Usar imágenes rbd en pools con Erasure Coding tiene algunas complicaciones mencionadas en la documentación oficial de Ceph (Erasure Code, 2016): por defecto, los pools con erasure code sólo son utilizados en RGW (Rados Gateway, la interfaz de objetos tipo S3 de AWS), porque estas aplicaciones escriben o modifican objetos completos. Desde el release *Luminous* de Ceph, la reescritura parcial de los objetos puede habilitarse en una configuración especial del pool para permitir a RBD y CephFS almacenar datos en este tipo de pools:

```
# ceph osd pool set testec allow_ec_overwrites true
```

Esto sólo funciona si los OSD donde reside el pool están sobre el backend Bluestore. Además, los pools erasure-coded no soportan omap, de modo que para utilizarlos con RBD hay que configurarlos para que almacenen los datos en el pool con erasure-code, pero los metadatos en un pool replicado.

A los efectos de medir la performance en el contexto de este trabajo, se crea una imagen donde los datos están sobre el pool **testec** y los metadatos sobre el pool **testhdd**, pero ya se tiene una idea de que no es una práctica muy utilizada en clusters de producción para almacenar volúmenes lógicos:

```
# rbd create --size 2G --data-pool testec testhdd/imagen2
```

Repetimos las pruebas, y obtenemos los valores mostrados en la Figura 104.

TESTEC	IOPS	Throughput
write	91	365
read	159	639
read write seq	157	631
readwrite rand	162	651

Figura 104. Resultados obtenidos de los testeos sobre pool configurado con Erasure Code.

Si tomamos los valores de **lecturas y escrituras aleatorias**, que suelen ser el patrón de acceso con más ocurrencia en los casos de uso de este cluster, vemos que la performance de un volumen sobre SSD casi duplica la de un volumen sobre HDD, y si miramos en detalle las tablas anteriores, podemos concluir que las operaciones más costosas son las **escrituras random**, y es en ellas donde los SSD hacen la diferencia. (Ver Figura 105)

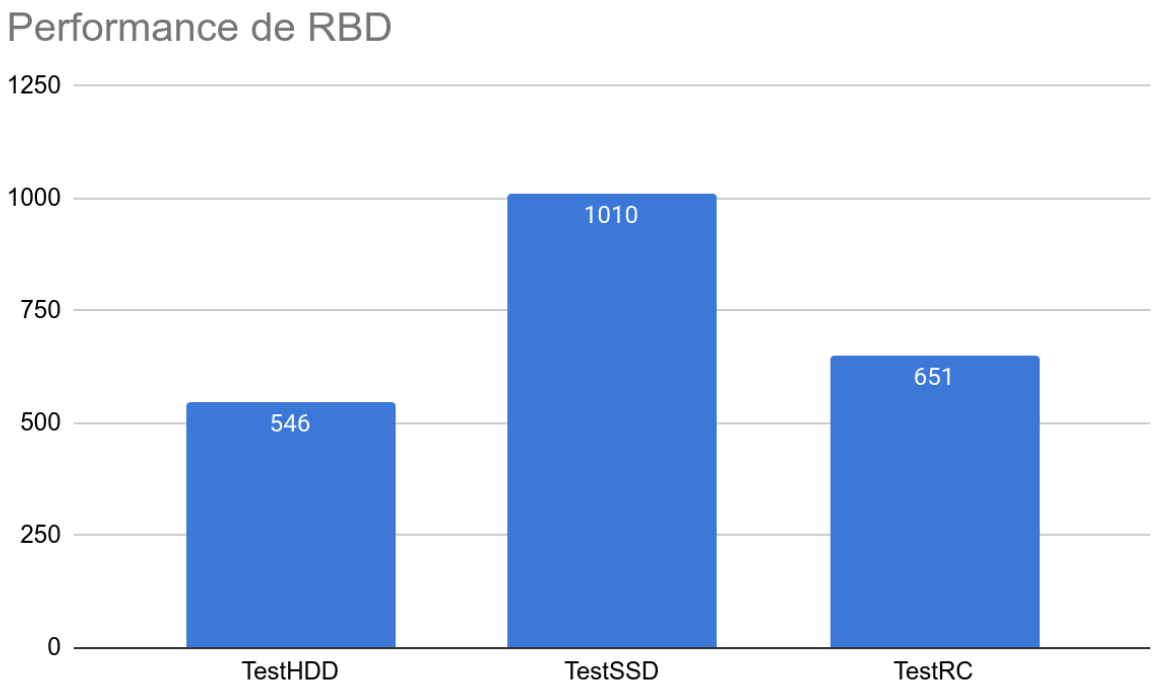


Figura 105. Comparación del throughput de escrituras random medido en MiB/s en los diferentes pools

9. BENCHMARKING y AJUSTES en los CLIENTES

Como se mencionó en el capítulo 3, Ceph provee a los clientes 3 tipos de interfaces: objetos, archivos y dispositivos de bloque.

Debido al uso extendido de los dispositivos de bloque y volúmenes lógicos presentes en la infraestructura de la universidad, este trabajo se centra en la performance y estrategias de optimización del servicio RBD, por lo que se instala en el cliente los módulos de kernel para rbd (Figura 106) de manera de poder ejecutar las pruebas y obtener las métricas habituales: throughput, latencia e IOPS.

```
# lsmod | grep rbd
rbd                    94208  0
ibceph                364544  1 rbd
```

Figura 106. Verificación de módulo rbd cargado en cliente.

Los módulos están instalados y la configuración del cluster está ubicada en /etc/ceph/ceph.conf.

9.1. Optimizar la caché del cliente

En el archivo de configuración conviene ajustar algunos valores para los datos que se cachean en el cliente con el objetivo de mejorar la performance. Esta configuración únicamente es tenida en cuenta por librbd, y no por krbd, pues el driver del kernel para los rbd puede utilizar la caché de páginas de Linux para mejorar la performance.

La implementación en espacio del usuario (*librbd)* no puede aprovechar la caché del kernel y por eso incluye su propio mecanismo en el que, cuando el Sistema Operativo envía un pedido de *flush,* todos los datos que estaban en memoria se bajan a disco. La caché propia de librbd (Figura 107) utiliza el algoritmo LRU (*Last Recently Used)* y el modo *writearound* por defecto (rbd cache mode, 2020).

```
[client]
        rbd_cache = true
        rbd_cache_size = 67108864     // 64 MiB
        rbd_cache_max_dirty = 50331648    // 48 MiB
        rbd_cache_target_dirty = 33554432  // 32 MiB
```

Figura 107. Configuración de los valores de caché en el cliente Ceph.

Si la caché de RBD está deshabilitada, las escrituras y lecturas van directamente al cluster de almacenamiento y las llamadas a escrituras sólo retornan cuando el dato está en disco en todas las réplicas.

Esta caché está en la memoria de cada usuario, lo que significa que no hay coherencia entre la caché de cada cliente, por más que estén accediendo al mismo

volumen. Por eso no debe habilitarse la caché si se va a utilizar la unidad para algún sistema de archivo de acceso concurrente, como OCFS2 o GFS.

Además, esta caché es POR UNIDAD MAPEADA en el kernel. Tener en cuenta esto a la hora de reservar recursos en la RAM para todas las unidades que un servidor puede mapear simultáneamente.

9.2. Otras funcionalidades y características de las imágenes RBD

Las imágenes rbd poseen algunas características avanzadas que pueden especificarse en el momento de su creación, que afectan su desempeño y tiempos de respuesta:

- Layering: habilita el uso de clonación
- Striping: distribuye los datos en múltiples objetos para mejorar el paralelismo
- Exclusive locking: requiere que el cliente bloquee la imagen antes de hacer una escritura en ella.
- Object map: Los dispositivos de bloque se implementan con “provisionamiento liviano”, es decir, que van ocupando el almacenamiento a medida que se van escribiendo datos. Object map lleva la trazabilidad de qué objetos existen para mejorar la velocidad de operaciones de E/S en clonación, importación, exportación y borrado de imágenes.
- Fast-diff permite calcular con mayor rapidez la diferencia entre una imagen y su snapshot.
- Deep-flatten: permite desconectar los snapshots de la imagen original.
- Journaling: registra todas las modificaciones que se hacen a una imagen. Esto lo utiliza mirroring para replicar una imagen en un cluster remoto.
- Data pool: en pools erasure-coded, los objetos de datos de la imagen deben almacenarse en un pool separado de la imagen que guarda los metadatos.
- Operations: restringe algunas operaciones de los clientes (clonar, snaps,...)
- Migrating: restringe el acceso a una imagen mientras está siendo migrada.

Generamos imágenes en los pool **testhdd** y **testssd** (Figura 108)

```
# rbd create testhdd/imagen1 --size 10G
# rbd create testssd/imagen1 --size 10G
# rbd list -p testhdd
     imagen1
```

Figura 108. Generación de imágenes de 10 GiB para testeos

El kernel del cliente que estamos utilizando (Debian 10, kernel 4.19.0-8-amd64) no es compatible con algunas de las características de las recién mencionadas y tenemos que deshabilitarlas, si no se obtiene un error a la hora de mapearlas a un identificador local, por

eso ejecutamos el comando `rbd feature disable testhdd/imagen1 object-map fast-diff deep-flatten`

Luego solicitamos al módulo que mapee la unidad “imagen1” del pool “testhdd” en una unidad local, en el directorio /dev con el comando `rbd map testhdd/imagen1`. A partir de este comando, la unidad del cluster se referencia con `/dev/rbd0` o `/dev/rbd/testhdd/imagen1`, que es un link simbólico al device anterior

Formateamos con ext4 y montamos en el árbol de directorios local. (Figura 109)

```
#mkfs.ext4 -m0 /dev/rbd0
mke2fs 1.45.6 (20-Mar-2020)
[...]
Se está creando un sistema de ficheros con 2621440 bloques de 4k y
655360 nodos-i
UUID del sistema de ficheros: e2fd73c6-f177-40e6-a3be-5b2fa4bef436
Respaldos del superbloque guardados en los bloques:
      32768, 98304, 163840, 229376, 294912, 819200, 884736, 1605632
[...]
# mount /dev/rbd0 /mnt
```

Figura 109. Formateo de la imagen rbd con ext4

Y realizamos esas mismas pruebas iniciales, con las que habíamos probado los discos antes de armar el cluster. (Figura 110)

```
#dd if=/dev/zero of=/mnt/archivo bs=1G count=5 oflag=direct
5368709120 bytes (5,4 GB, 5,0 GiB) copied, 11,7749 s, 456 MB/s
```

Figura 110. Pruebas iniciales sobre el dispositivo rbd.

Si volvemos algunas páginas atrás, veremos que la misma prueba sobre un único disco nos dio 202MB/s, cuando ahora obtenemos 476 MB/s. Por supuesto, acá intervienen todas las capas de software que describimos, y la escritura de estos 5 GiB se dividen en objetos de 4 MB, que se agrupan en distintos Placement Groups, que se distribuyen en varios OSD, cuyo ancho de banda se agrega.

Sería la situación ideal: todo un cluster para un único usuario, que usa un único volumen. En su trabajo E*valuating the performance and scalability of the Ceph distributed storage system* (Gudu, 2015) los autores manifiestan la dificultad de *saturar* un cluster desde el lado de los clientes, porque -justamente- el cluster está diseñado para escalar y balancear grandes volúmenes de datos sin perder performance. No obstante, es interesante hacer pruebas de escrituras simultáneas de varios procesos sobre el mismo volumen o distintos volúmenes del mismo pool, para ver si se registra una merma en el throughput o un incremento en la latencia.

Se muestran resultados de los testeos de escrituras aleatorias IOPS en x1 y Latencia en Figura 112.

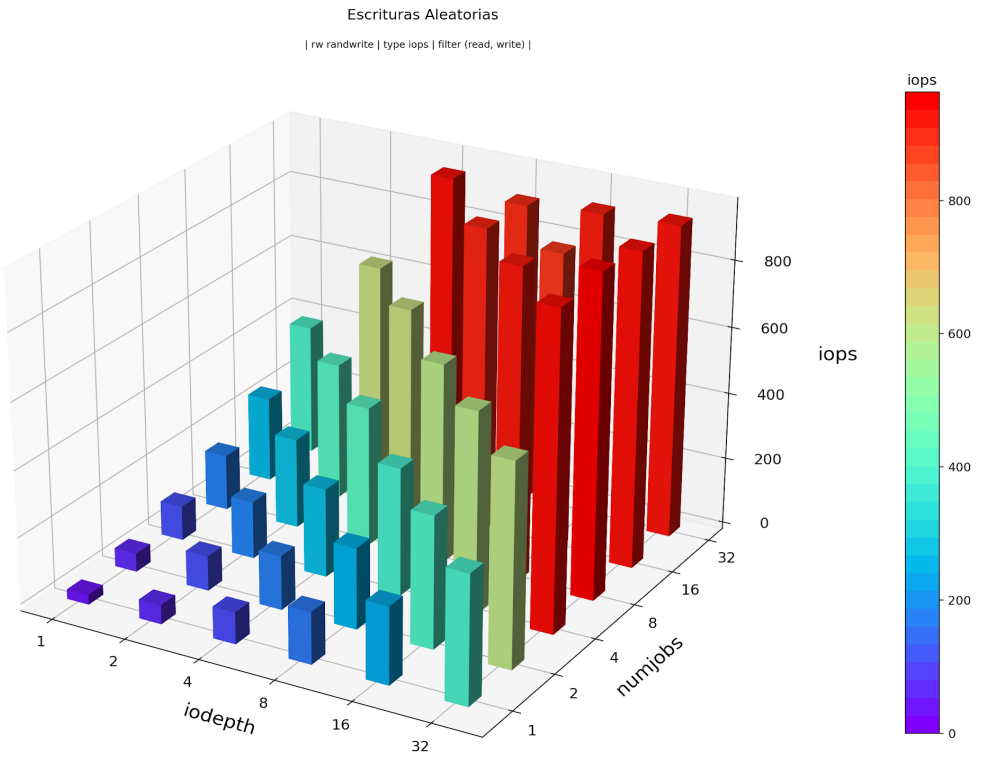


Figura 111. Pruebas de escrituras random (IOPS)

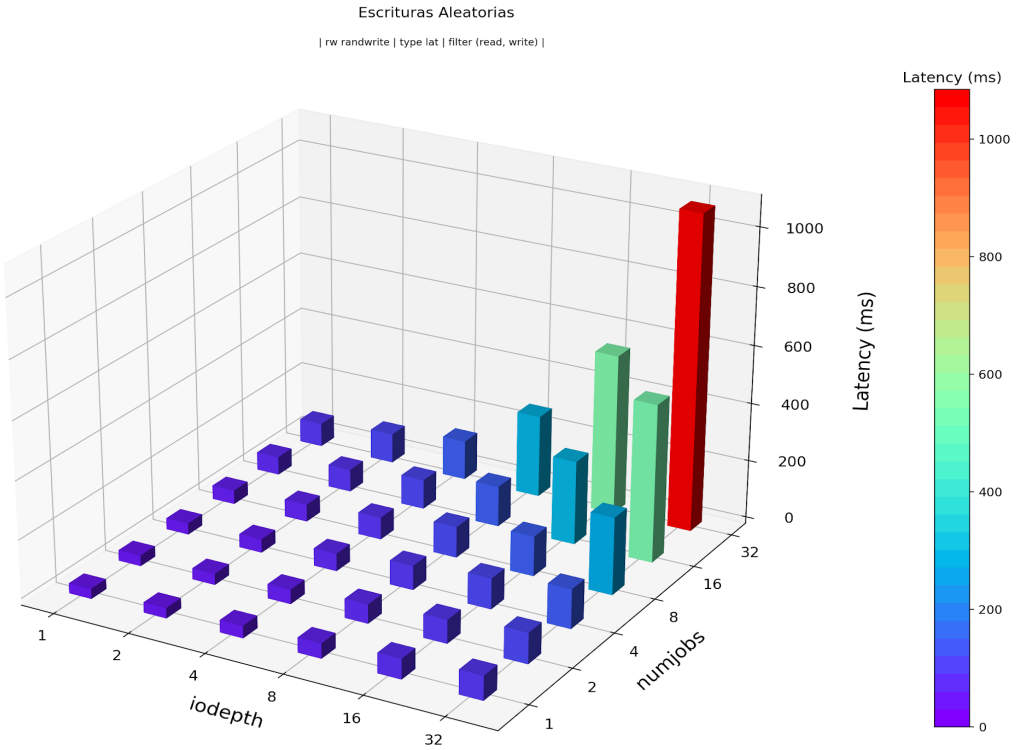


Figura 112. Prueba de escrituras random (Latencia).

Si analizamos más en detalle para n=8 procesos vemos en la Figura 113 que los IOPS tienen un techo en 916, y una latencia de 140 milisegundos cuando cada uno de esos procesos ejecuta 8 solicitudes de E/S concurrentes. Si aumentamos la cantidad de threads, se dispara la latencia y las IOPS no registran un aumento.

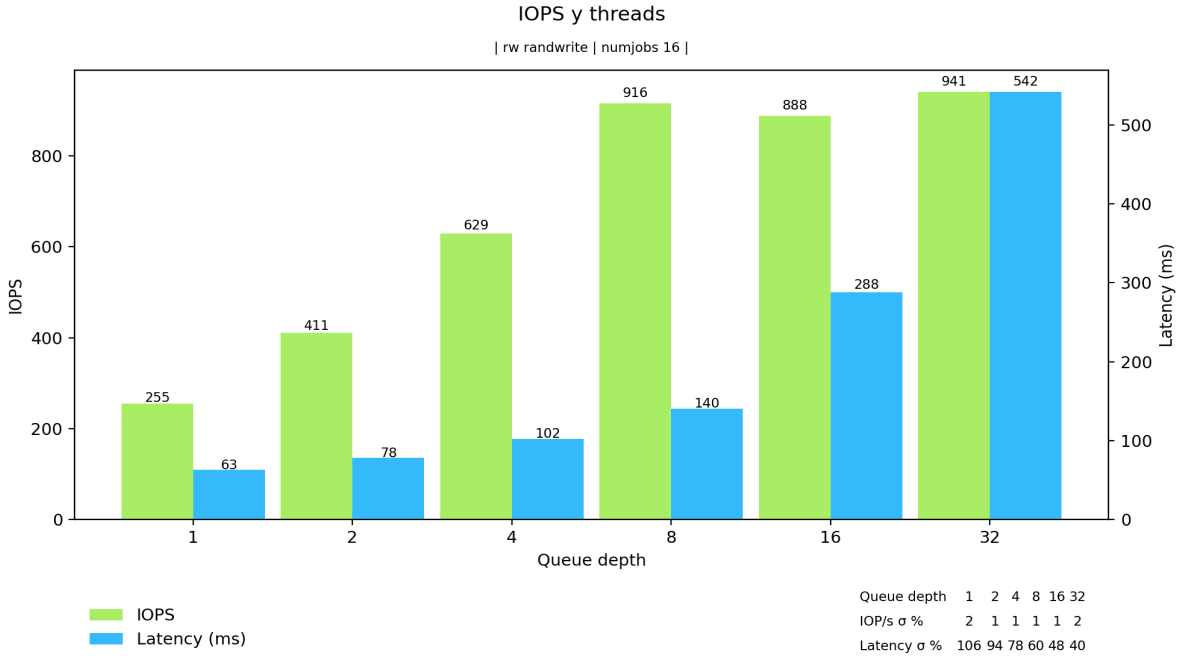


Figura 113. Correlación entre IOPS y latencia ejecutando 8 procesos.

Si reiteramos el análisis en detalle para n=4 procesos concurrentes, vemos en la Figura 114 que los IOPS llegan a 945 con 32 threads por proceso, y una latencia de 135 milisegundos.

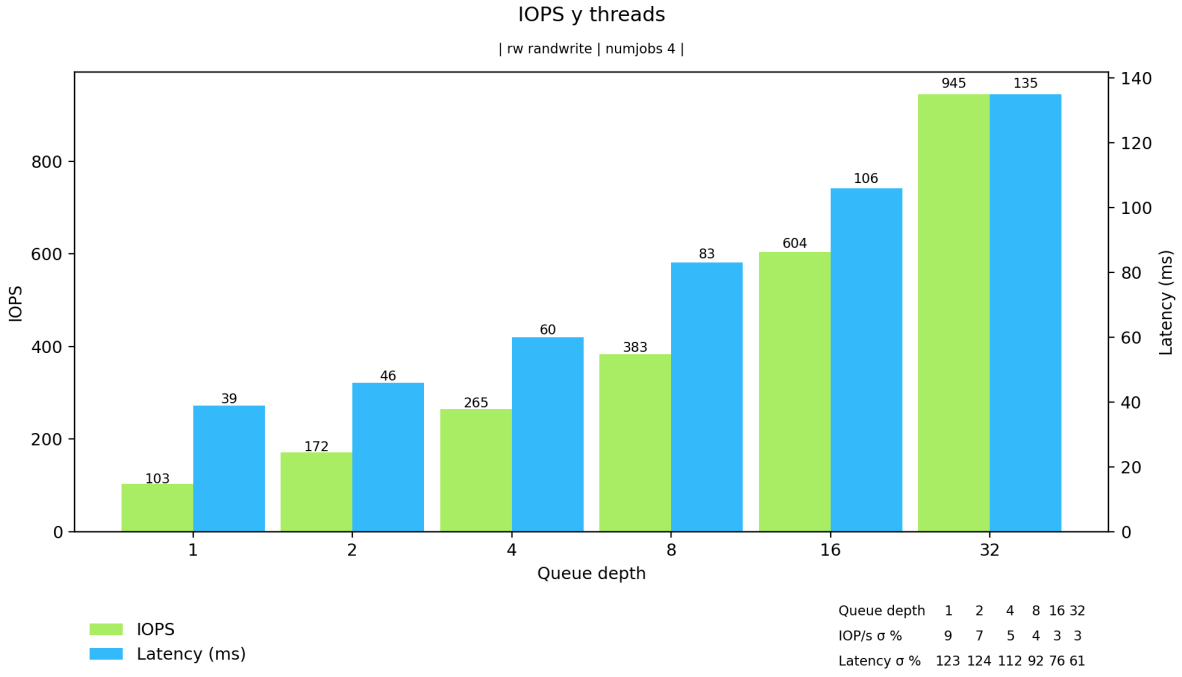


Figura 114. Correlación entre IOPS y latencia lanzando 4 procesos.

Así, vemos que la performance mejora con 4 procesos que ejecutan 32 operaciones de entrada salida, antes que con 8 procesos que ejecutan 16 operaciones.

A modo de comentario, el sistema de monitoreo (Figura 115) registra las IOPS del pool durante los 36 minutos que dura la prueba, en los 6 tests iterados (1,2,4,8,16 y 32 procesos) cada uno con un número de threads de 1,2,4,8,16 y 32.

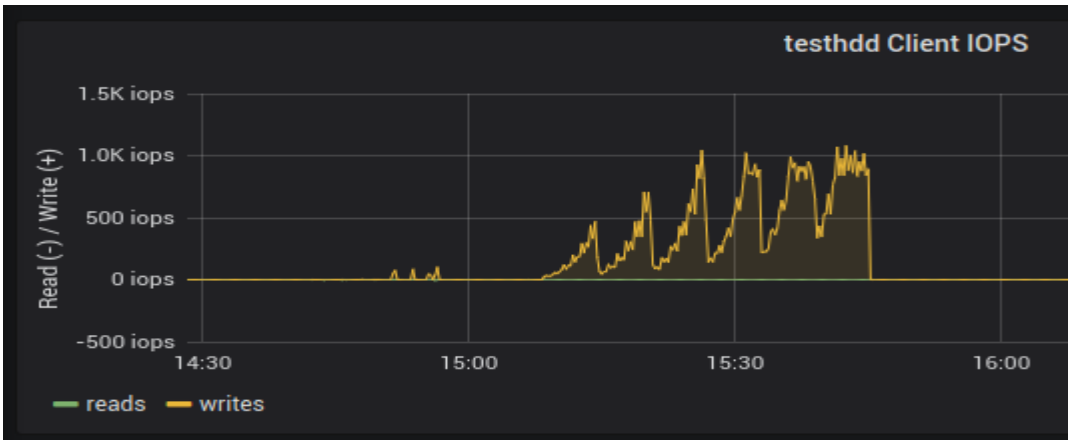


Figura 115. Escrituras registradas en el sistema de monitoreo del cluster.

Repetimos ahora las pruebas de IOPS (Figura 116) y latencia (x8) con una imagen rbd en el pool sobre discos SSD.

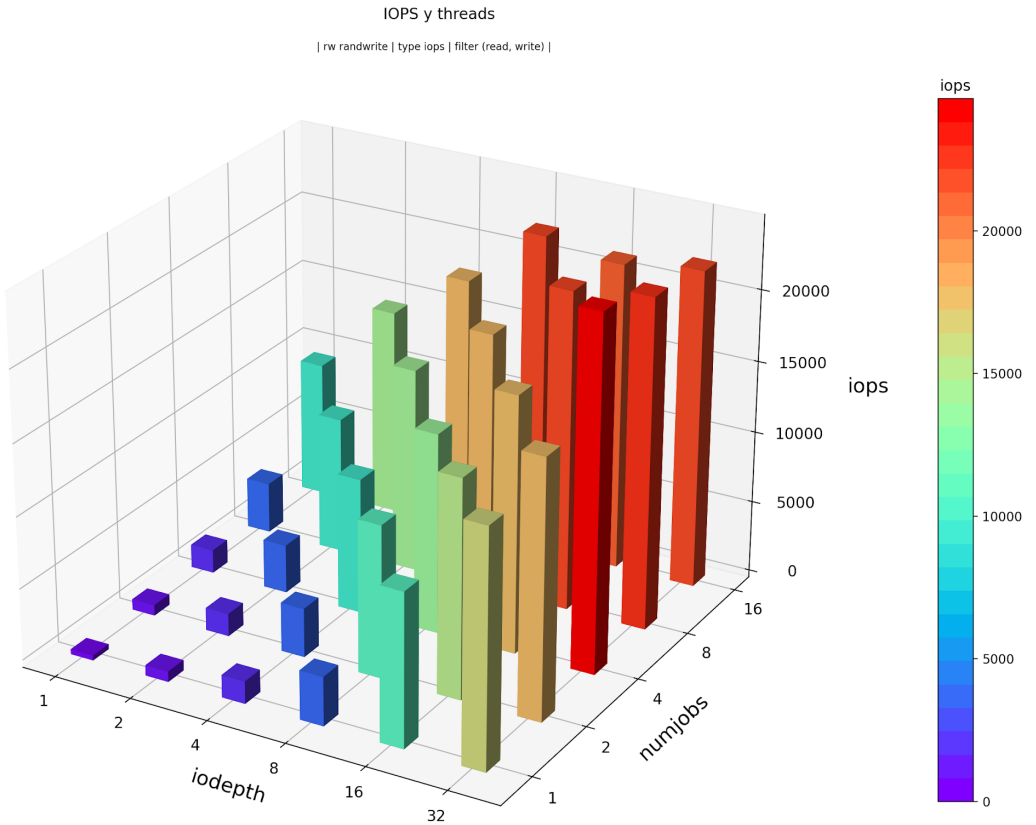


Figura 116. Prueba de escrituras random (IOPS)

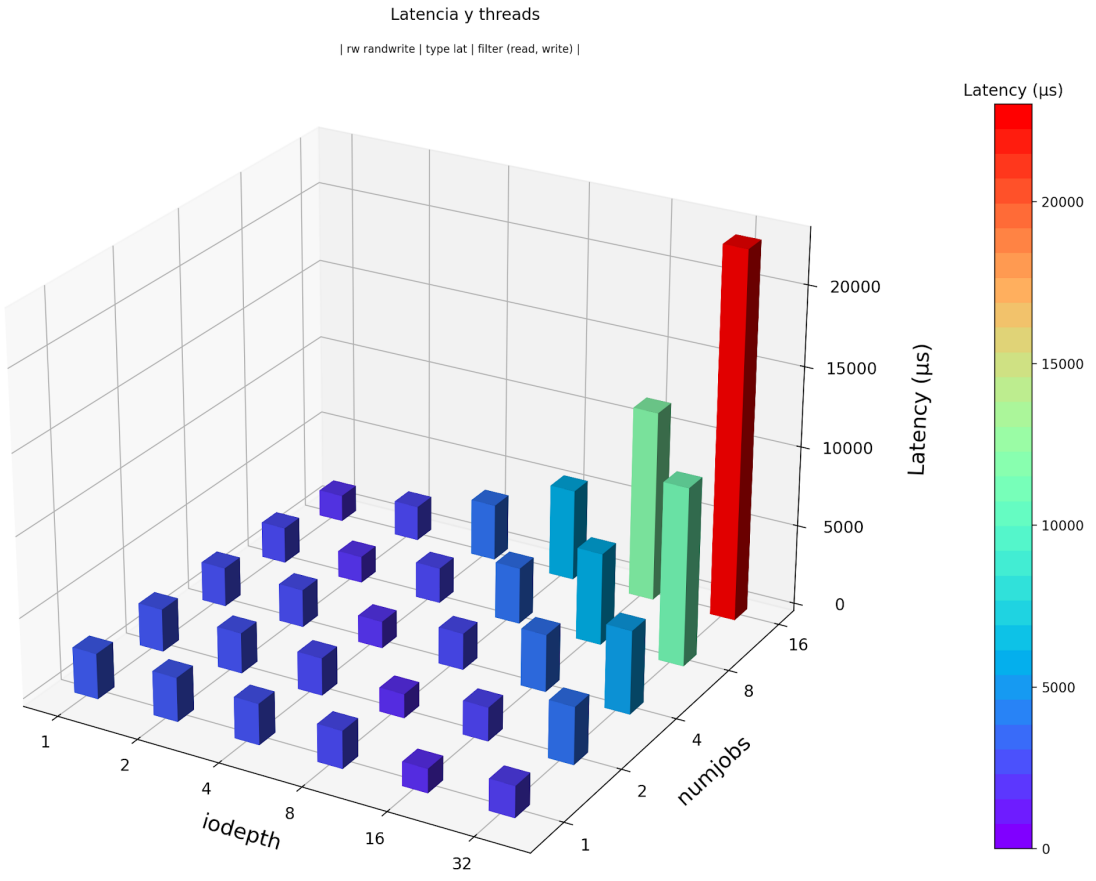


Figura 117. Prueba de escrituras random (latencia)

Parecería entonces que estas imágenes RBD tienen su mejor performance utilizando 8 procesos (Figura 118), cada uno con 32 threads, donde se obtienen 23000 IOPS, y una latencia de 11 milisegundos.

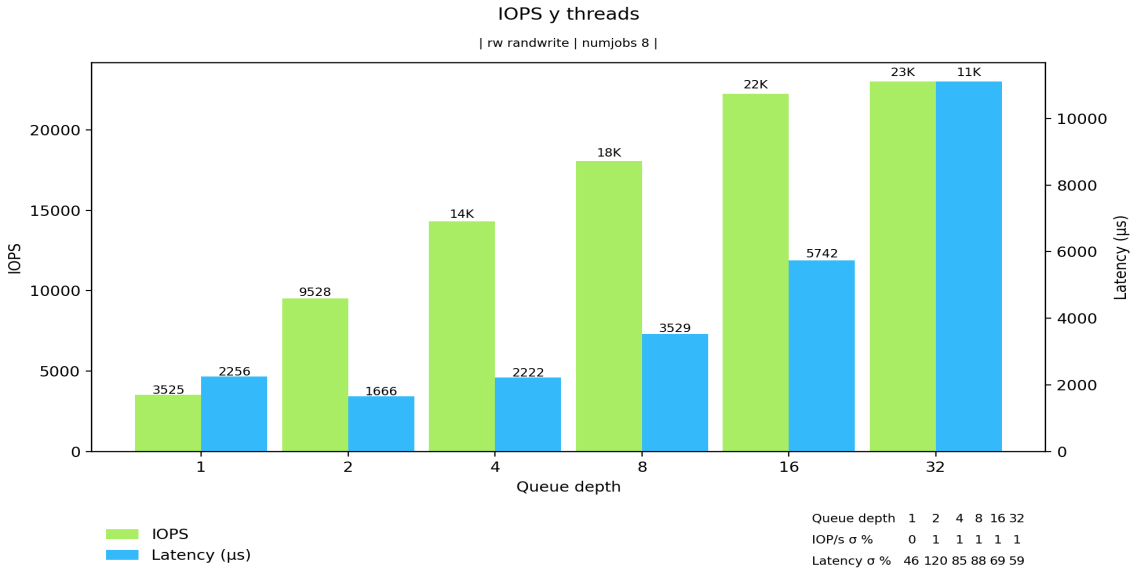


Figura 118. IOPS y Latencia para 8 procesos concurrentes.

Pues al aumentar la cantidad de procesos a 16 (Figura 119), para 16 threads se obtienen 21000 IOPS y 11 milisegundos (2000 IOPS menos que para 8 con 32 del caso anterior). Y si aumentamos a 16 procesos con 32 threads, los IOPS suben a 22000, pero la latencia se duplica a 23 milisegundos.

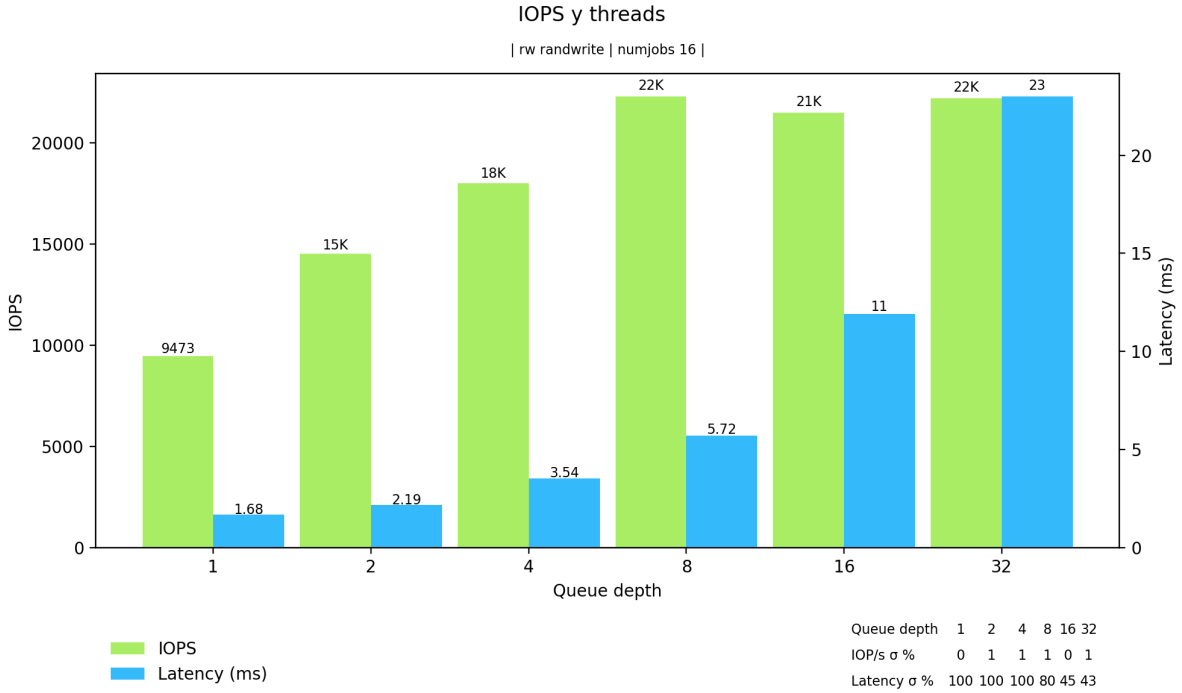


Figura 119. IOPS y Latencia para 16 procesos concurrentes.

Pero si miramos en detalle qué sucede cuando hay 32 procesos (Figura 120) realizando operaciones de E/S, con 4 threads por proceso nos da la mejor performance: 22000 IOPS, 5.75 milisegundos.

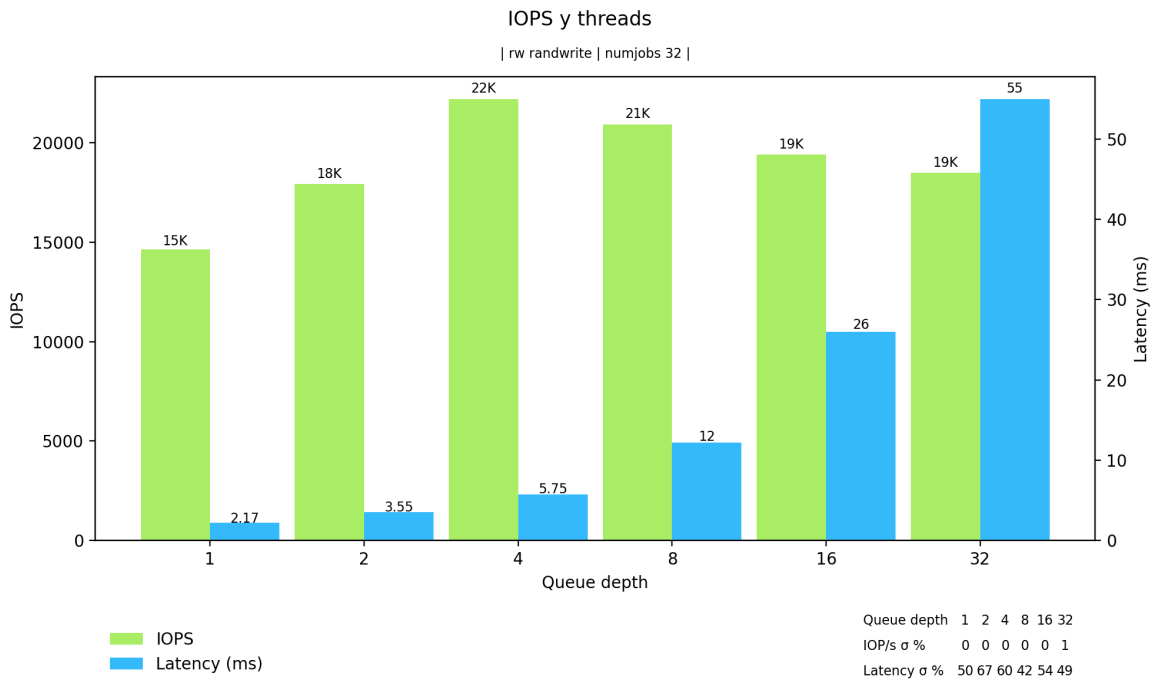


Figura 120. IOPS y Latencia para 32 procesos concurrentes.

El sistema de monitoreo (Figura 121) registra las IOPS sobre el pool durante el testeo:

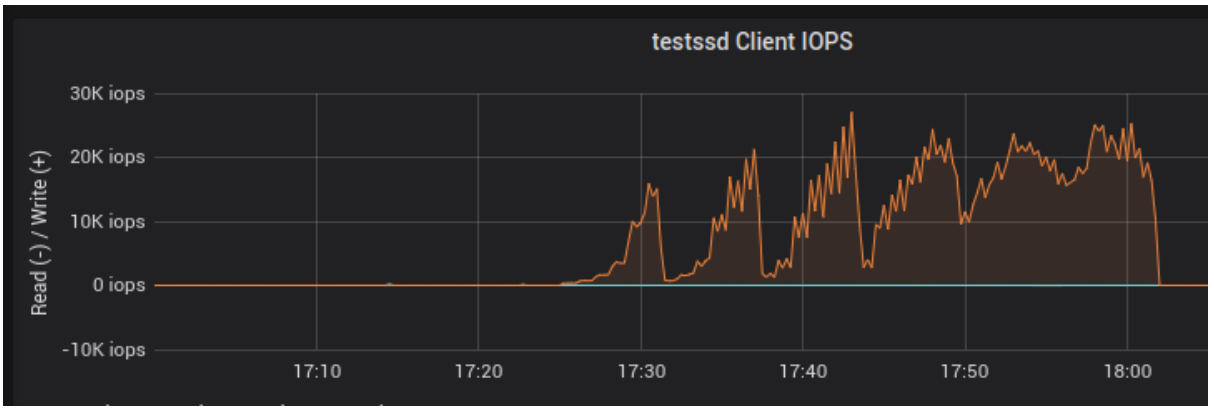


Figura 121. Sistema de monitoreo registra las IOPS durante los testeos.

9.3. Impacto del hardware que mapea el RBD

Se prueba ahora el acceso a **una misma imagen**, desde 3 clientes distintos:

- Server 1: Server Físico, con 8 procesadores y 8 GB RAM, 1Gbps LAN
- Server 2: Server Físico, con 48 procesadores y 128 GB RAM, 10Gbps LAN
- Server 3: Server Virtual, con 1 procesador y 2 GB RAM, 10 Gbps LAN

La prueba se realiza con *fio* (Figura 122), con una carga media de escrituras random de 4K, con 4 procesos concurrentes y cada uno con 16 threads haciendo E/S. Esta carga no busca la máxima performance del RBD, sino corroborar el impacto sobre ella de hardware y el sistema operativo que mapea la unidad. Se testea también como referencia la red LAN de los clientes contra uno de los nodos que tienen los OSD con los PG del pool utilizado.

```
# fio --ioengine=libaio --direct=1 --filename=/dev/rbd0 --bs=4k
--rw=randwrite --group_reporting --iodepth=16 --numjobs=4
--name=test1 --runtime=60
```

Figura 122. Comando utilizado para testear el server que mapea el RBD.

Se puede observar que la performance de una misma imagen RBD varía de acuerdo al equipo que la mapea, obteniendo un mayor throughput y menor latencia el equipo con más hardware (Figura 123). Y también puede verse que la LAN no tiene incidencia en este nivel de carga.

	CPU	RAM	Latencia promedio (ms)	IOPS	Throughtput Disco (MiB/s)	Throughput LAN (Mbps)
Server1	8	8	2.75	23200	90.6	919
Server2	48	128	2.55	25000	97.8	9360
Server3	1	2	3.29	19300	75.6	8560

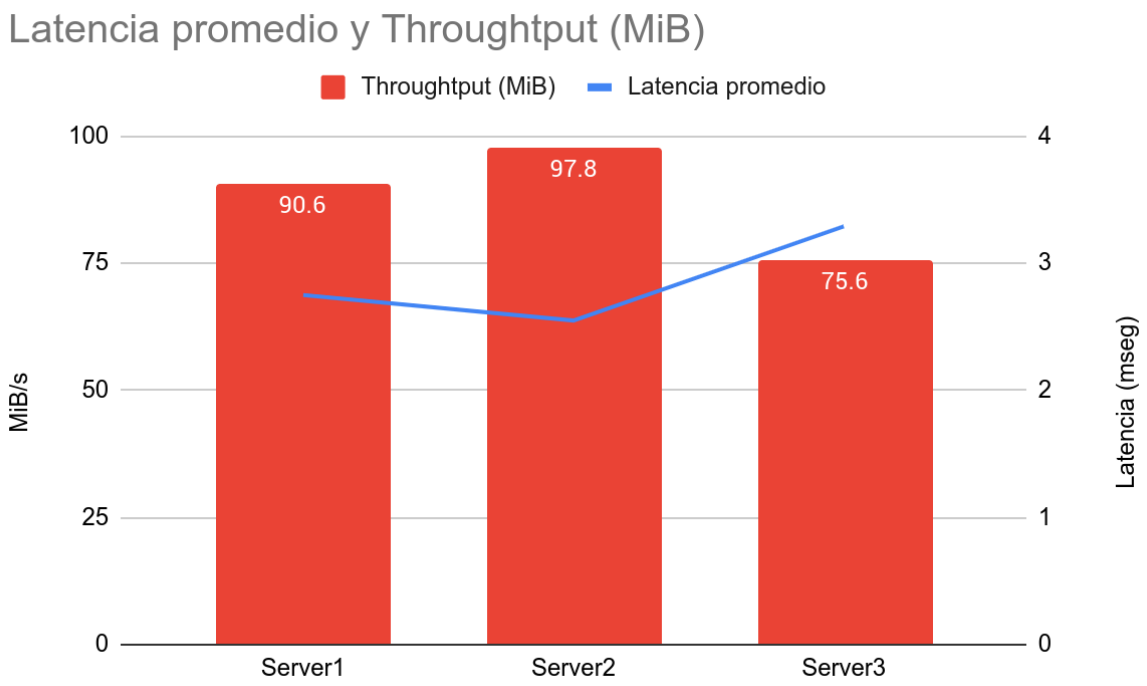


Figura 123. Incidencia del hardware del equipo que mapea la unidad RBD.

9.4. Integración con plataformas de virtualización

Este cluster inicialmente se pensó como una opción para almacenar los volúmenes de las VM (domU) que actualmente están sobre LVM, o conectados a unidades remotas vía NFS o iSCSI, tal y como se describió en la introducción de este trabajo.

A todas las capas de software que ya describimos, estamos agregando un par más: el hipervisor y el mismo sistema operativo instalado en cada máquina virtual. Nuevamente, vamos a observar una merma en la performance.

Es interesante el trabajo de Johanes Johari *Comparison of various virtual machine disks images performance on GlusterFS and Ceph Rados Block Devices* (Johari, 2014),

donde compara la performance de las distintas imágenes (Qcow2, raw, QED, RBD) habitualmente utilizadas en distintos hipervisores.

9.4.1. Xen

Las máquinas virtuales corren bajo el hipervisor Xen versión 4.11.2-pre (Debian 4.11.1+92-g6c33308a8d-2), y hubo que buscar un parche que posibilitara utilizar las unidades rbd de Ceph, ya que la versión utilizada de Xen no proveía la interfaz de manera nativa. En la Figura 124 se muestra un esquema de la arquitectura del hipervisor.

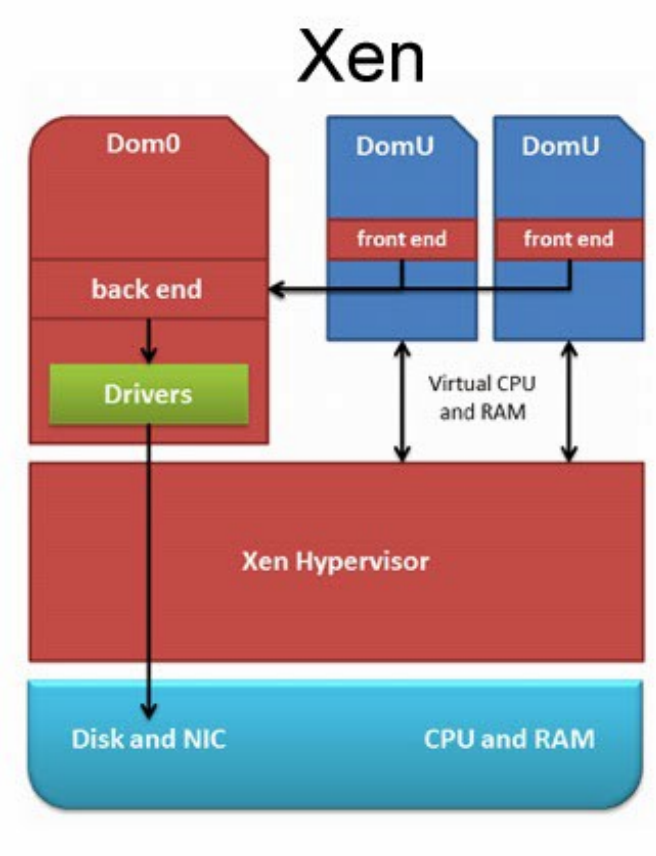


Figura 124. Esquema de la arquitectura de XEN.

Un script desarrollado por Florian Heigl (Heigl, 2020) permite la integración de pools e imágenes rbd en las configuraciones de máquinas virtuales de Xen. La línea que especifica el almacenamiento indica la ejecución del script **block-rbd** que realiza el mapeo a través del módulo de kernel de la imagen rbd de Ceph a un dispositivo local, y la conecta al domU:

```
disk = [ 'script=block-rbd,vdev=xvdN,target=poolname:image_name' ]
```

La configuración del domU puede verse en la Figura 125.

```
name        = ‘test_vmachine'
memory      = '2048’
vcpus       = 2
kernel      = "/boot/vmlinuz-4.19.0-8-amd64"
ramdisk     = "/boot/initrd.img-4.19.0-8-amd64"
root        = '/dev/xvda2 ro'
disk        = [ 'script=block-rbd,vdev=xvda,target=testssd:discovmachine']
vif         = [ 'bridge=vlan14, ip=172.19.205.52, mac=00:16:3E:E0:D1:0A']
on_poweroff = 'destroy'
on_reboot   = 'restart'
on_crash    = 'restart'
```

Figura 125. Configuración completa del domU.

Algunas pruebas (Figura 126) sobre distintos domU verifican el hecho de que la performance se degrada y que hay una merma importante en el throughput de la unidad RBD. Se prueban 4 procesos, cada uno escribe un archivo de 1 GiB con accesos random de 4k en 16 threads.

```
# fio --ioengine=libaio --direct=1 --bs=4k --rw=randwrite
--group_reporting --iodepth=16 --numjobs=4 --name=test1
--runtime=60 --size=1G
```

Figura 126. Comando utilizado para testear la VM virtualizada con Xen

- Prueba concurrente de 3 máquinas virtuales (domU) con volúmenes en pool de discos **ssd** realizando la prueba descripta. (Figura 127)

```
write: IOPS=20.0k, BW=78.3MiB/s, lat=3.1 mseg
write: IOPS=19.5k, BW=76.3MiB/s, lat=3.2 mseg
write: IOPS=15.3k, BW=59.0MiB/s, lat=4.1 mseg
```

Figura 127. Prueba sobre VM en pool SSD.

- Prueba concurrente de 3 máquinas virtuales sobre el pool de discos **hdd** (Figura 128).

```
write: IOPS=605, BW=2422KiB/s, lat=105 mseg
write: IOPS=492, BW=1971KiB/s, lat=129 mseg
write: IOPS=589, BW=2358KiB/s, lat=108 mseg
```

Figura 128. Prueba sobre VM en pool SSD.

La performance de las máquinas virtuales no se ve afectada por el acceso concurrente al cluster, aunque es notoria la degradación que produce el hipervisor utilizado (xen) en los valores obtenidos.

9.4.2. Kubernetes

Para probar el desempeño de las imágenes rbd asociadas a contenedores, se usa un cluster **Kubernetes** ya instalado y operativo.

Kubernetes es una plataforma portable y extensible de código abierto para administrar cargas de trabajo y servicios que facilita la automatización y la configuración

declarativa porque orquesta la infraestructura de cómputo, redes y almacenamiento. En la Figura 129 puede verse un esquema comparativo de las arquitecturas tradicionales, de virtualización con hipervisores y de virtualización con contenedores.

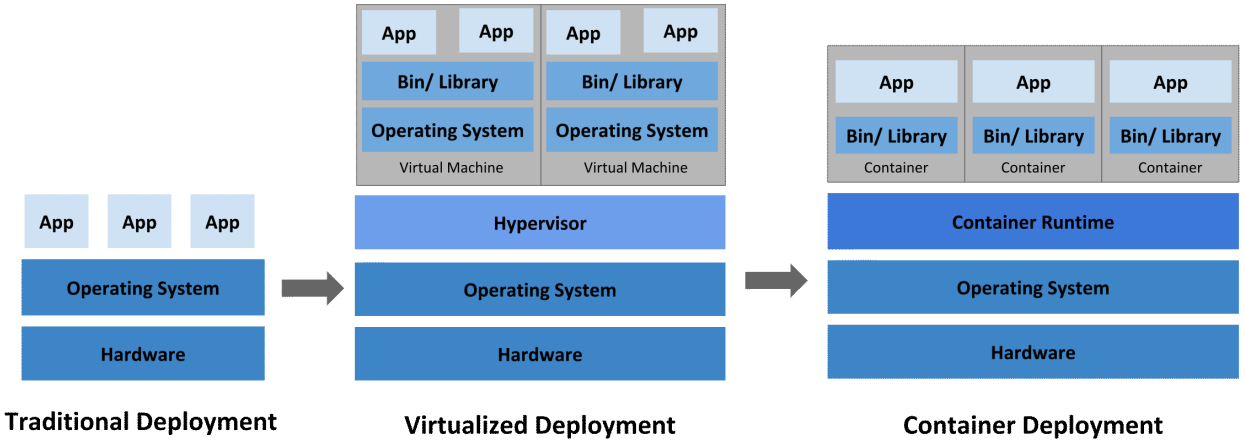


Figura 129. Comparativa de arquitecturas.

En Ceph creamos un pool sobre discos ssd para las imágenes de uso exclusivo de Kubernetes (Block Devices and Kubernetes, 2020) de manera de probar la performance de los contenedores en el mismo conjunto de operaciones. Además, utilizamos un plugin para provisionar de manera dinámica las imágenes, attachearlas y montarlas en los contenedores. (rbd CSI plugin, 2020)

Utilizamos Rancher (Rancher, 2020) para gestionar el cluster de Kubernetes y definimos una clase de storage para conectar con Ceph (Figura 130).

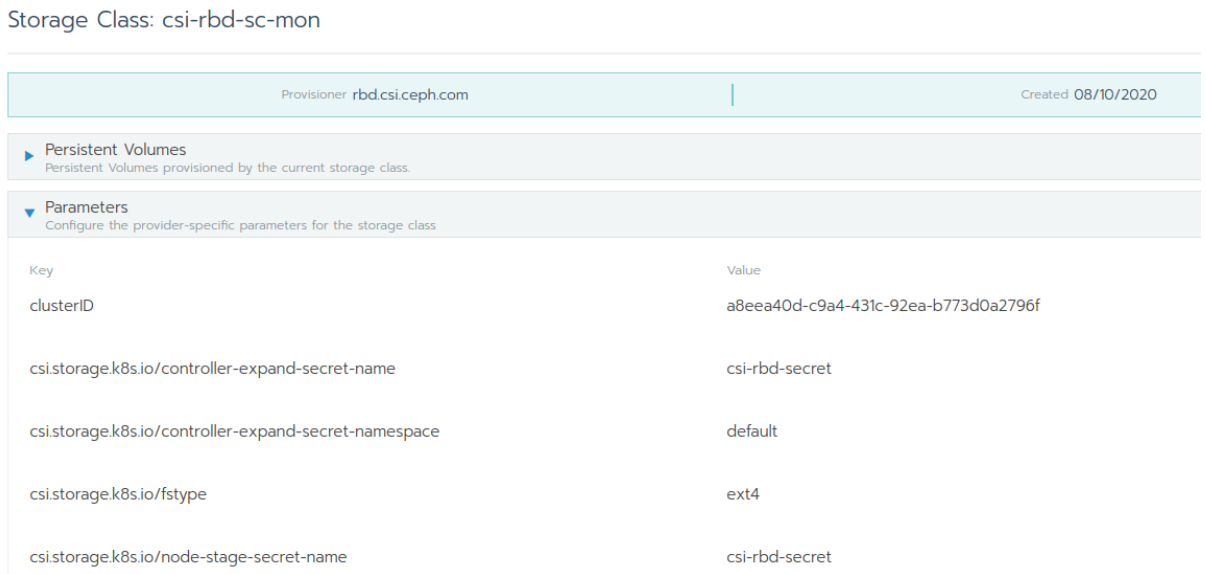


Figura 130. Clase definida en Rancher para utilizar Ceph como almacenamiento.

En la Figura 131 vemos la creación de un volumen persistente, asociado al contenedor:

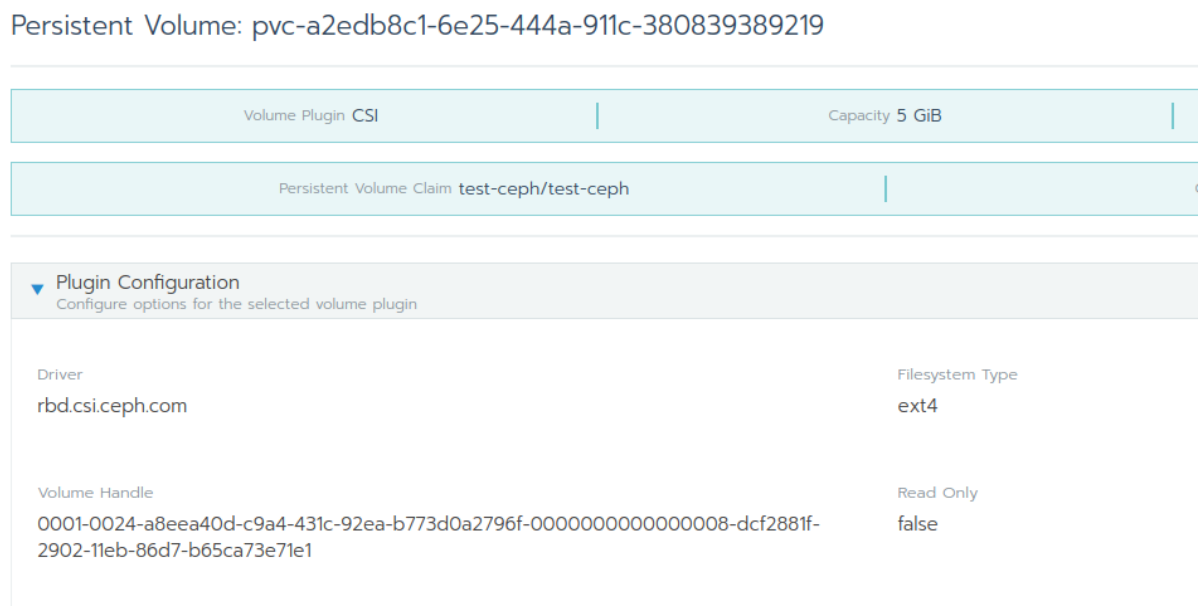


Figura 131. Asociación de un volumen al contenedor.

Lanzamos el contenedor test-ceph-77d9d4875b-kwsl9, que ejecuta el mismo comando fio y se obtiene un throughput de 57.6 MiB/s, con 14700 IOPS y una latencia de 4.33 milisegundos (Figura 132). Analizando estos valores, vemos que la performance no es óptima porque el nodo de cómputo del cluster kubernetes que ejecutó el contenedor está sobre una máquina virtual, y esto afecta su desempeño. El monitoreo de la prueba puede verse en la Figura 133.

```
...
fio-3.16
Starting 4 processes
test1: Laying out IO file (1 file / 1024MiB)
test1: Laying out IO file (1 file / 1024MiB)
test1: Laying out IO file (1 file / 1024MiB)
test1: Laying out IO file (1 file / 1024MiB)
Jobs: 4 (f=4): [w(4)][100.0%][w=51.8MiB/s][w=13.3k IOPS][eta 00m:00s]
test1: (groupid=0, jobs=4): err= 0: pid=691: Tue Nov 17 18:35:20 2020
  write: IOPS=14.7k, BW=57.6MiB/s (60.4MB/s)(3456MiB/60005msec); 0 zone resets
    slat (usec): min=4, max=49514, avg=35.50, stdev=177.46
    clat (usec): min=300, max=193530, avg=4300.92, stdev=3698.78
     lat (usec): min=989, max=193568, avg=4337.34, stdev=3703.23
    clat percentiles (usec):
     |  1.00th=[  1713],  5.00th=[  2540], 10.00th=[  2868], 20.00th=[  3261],
     | 30.00th=[  3556], 40.00th=[  3785], 50.00th=[  4015], 60.00th=[  4293],
     | 70.00th=[  4555], 80.00th=[  4948], 90.00th=[  5538], 95.00th=[  6194],
     | 99.00th=[  8717], 99.50th=[ 10814], 99.90th=[ 51643], 99.95th=[ 92799],
     | 99.99th=[173016]
   bw (  KiB/s): min=32696, max=71722, per=99.97%, avg=58954.32, stdev=1740.10, samples=480
   iops        : min= 8174, max=17930, avg=14738.42, stdev=435.02, samples=480
  lat (usec)   : 500=0.01%, 750=0.01%, 1000=0.01%
  lat (msec)   : 2=1.80%, 4=47.07%, 10=50.51%, 20=0.37%, 50=0.14%
  lat (msec)   : 100=0.07%, 250=0.05%
  cpu          : usr=2.15%, sys=9.67%, ctx=1016488, majf=0, minf=46
  IO depths    : 1=0.1%, 2=0.1%, 4=0.1%, 8=0.1%, 16=100.0%, 32=0.0%, >=64=0.0%
     submit    : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     complete  : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.1%, 32=0.0%, 64=0.0%, >=64=0.0%
     issued rwts: total=0,884673,0,0 short=0,0,0,0 dropped=0,0,0,0
     latency   : target=0, window=0, percentile=100.00%, depth=16

Run status group 0 (all jobs):
  WRITE: bw=57.6MiB/s (60.4MB/s), 57.6MiB/s-57.6MiB/s (60.4MB/s-60.4MB/s), io=3456MiB (3624MB), run=60005-60005msec

Disk stats (read/write):
  rbd2: ios=0/883876, merge=0/24199, ticks=0/2720760, in_queue=2695624, util=99.84%
root@test-ceph-77d9d4875b-kwsl9:/mnt#
```

Figura 132. Ejecución del comando fio dentro de un contenedor.

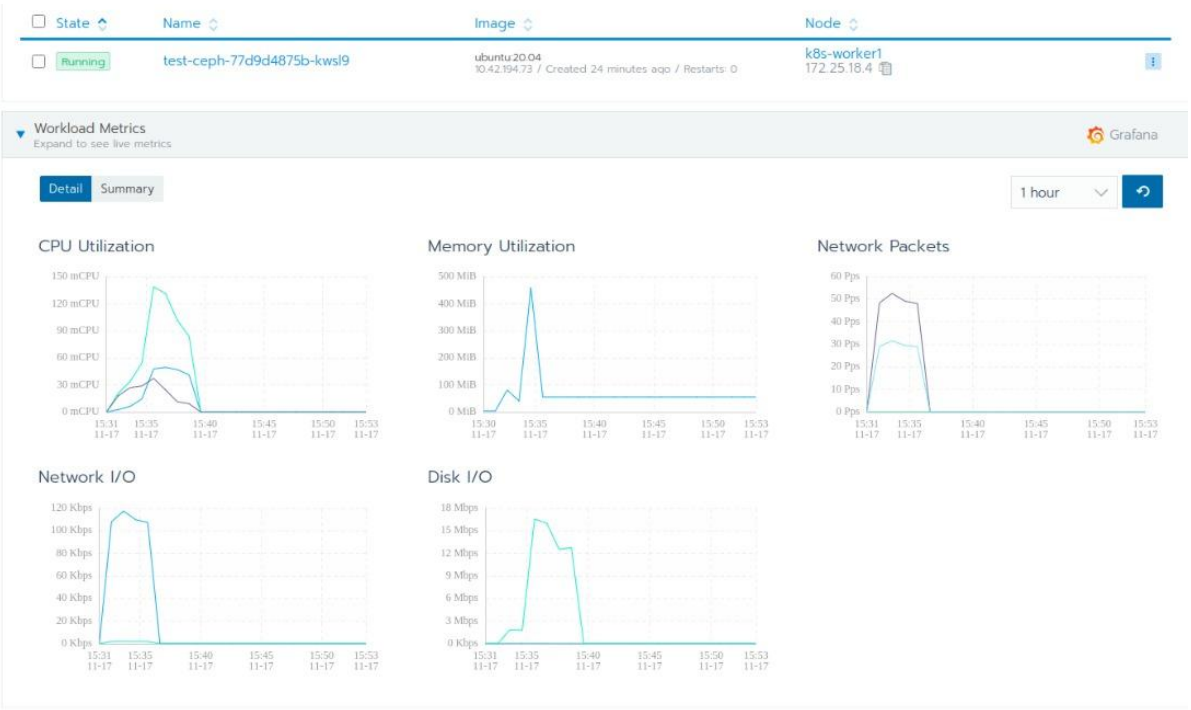


Figura 133. Monitoreo de la prueba en el contenedor.

10. A MODO DE SLA

SLA *(Service Level Agreement)* o "Acuerdo de Niveles de Servicio" es un acuerdo normalmente escrito que realizan un proveedor y su cliente, donde establecen el nivel acordado para la calidad de dicho servicio. Este documento ayuda a ambas partes a llegar a un consenso en términos de latencia, throughput, disponibilidad, tiempo de respuesta ante incidentes, etc… Sirve como referencia para las operaciones de monitoreo, mantenimiento y control del servicio en el proceso de mejora continua, y también como marco ante conflictos, porque establece los puntos donde había un compromiso de las partes.

En nuestro caso, proveedor y cliente del servicio recaen en la misma entidad; no obstante, es útil disponer de un documento informal que describa cuál es el comportamiento habitual del cluster de almacenamiento distribuido instalado, cuáles son los ajustes y configuraciones realizadas, cuáles son las expectativas de performance y qué valores de latencia, throughput e IOPS pueden esperarse.

De acuerdo a la metodología bottom-up empleada en este trabajo, se sintetizan ahora las configuraciones y los distintos valores obtenidos en los testeos realizados en cada nivel: red de datos, discos, OSDs, RADOS, RBD y los clientes.

10.1. La red de datos

- Verificación de balanceo de las interrupciones de cada interfaz de red en el vector de interrupciones de cada procesador disponible en el nodo.
- Aumentar el valor de net.core.netdev_budget si la tercera columna de /proc/net/softnet_stat tiene valores crecientes.
- Verificación de pause frames y interrupt coalescing, que por defecto vienen habilitados.
- Aumentar el valor de net.core.netdev_max_backlog si la segunda columna de /proc/net/softnet_stat tiene valores distintos de 0.
- Ampliación de los buffers de recepción de los adaptadores de red al máximo que permite el driver (ethtool -g)
- Si están habilitados los jumbo-frames en el nodo, deben estarlo en **todas las interfaces de todos** los switches, servidores y clientes del dominio de broadcast.
- Ampliación de los valores de memoria mínimos, default y máximo reservados por el sistema operativo para los sockets TCP en net.ipv4.tcp_rmem y net.core.rmem_max

Habiendo realizado esas configuraciones, los servidores con interfaces de 10 Gbps SFP+ tienen un throughput sostenido de **9.40 Gbps**, con mínimas retransmisiones (119 en 20 segundos) y un RTT promedio de 163 microsegundos.

En los nodos con interfaces de 1 Gbps en cobre, el throughput es de 941 Mbps, sin retransmisiones en los 20 segundos del intervalo de pruebas, y un RTT promedio de 220 microsegundos.

Los discos HDD

Los discos magnéticos de 4TiB, 7200 rpm e interfaz SATA 3 en 6 Gbps tuvieron un throughput máximo de **202 MB/s** en la prueba con *dd* de escritura secuencial de 5 GiB.

Los análisis con *fio* mostraron resultados aproximados a ese valor en la escritura secuencial realizada por 4 procesos con 4 threads y un tamaño de bloque de 1 MiB (**196 MiB/s** y latencia de 59 mseg.) mientras que la lectura secuencial tuvo su máximo de **309 MiB/s** al realizar lecturas en bloques de 16KiB, con una latencia de 0.7 mseg.

En las operaciones random, con un tamaño de bloque de 4 MiB se obtuvieron los mejores rendimientos, de **44.9 MiB** y **135 MiB** para escritura y lectura respectivamente, pero las latencias fueron las más altas, de 400 y 332 mseg. Esto es consistente con el hecho de que, una vez que se ubicó el brazo en posición, leer o escribir un bloque de mayor tamaño aumenta el throughput por operación.

Al dejar fijo el tamaño de bloque en 4 KiB y observar los valores que toman las IOPS y latencia conforme se amplía la queue_depth, vemos que el mejor resultado es cuando esa cola de operaciones tiene un tamaño de 64. Escapa a este trabajo el análisis de la incidencia del mecanismo utilizado por el driver del disco para minimizar movimientos de brazo y obtener máxima performance, pero el rememorar algoritmos como el C-scan o el del ascensor nos provoca una sonrisa.

En las lecturas secuenciales, no obstante, no se registra una mejora en la performance al aumentar las operaciones encoladas: una cola de 2 o de 64 tienen el mismo rendimiento y la latencia empeora.

En las operaciones aleatorias, las escrituras y las lecturas tienen un número de IOPS mejor cuando la cola es mayor, aunque también aumenta la latencia.

Para estresar el dispositivo en escrituras aleatorias, al mismo tiempo en que se itera sobre el tamaño de la queue_depth, se aumenta el número de procesos que realizan simultáneamente operaciones de E/S, y se observa que el máximo de IOPS obtenido es de **175** para 32 procesos con una cola de 2 operaciones, o para un único proceso con una cola de 32 operaciones, ambas con una latencia por debajo de los 40 milisegundos. Al aumentar los procesos o la cola de operaciones en este esquema se dispara la latencia hasta valores de 5 segundos (32 procesos, 32 operaciones).

10.2. Los discos SSD

Los discos de estado sólido de 1.9 TiB, SATA 3 a 6 Gbps, tuvieron un throughput máximo de **478 MB/s** en la prueba con *dd* de escritura secuencial de 5 GiB.

Los análisis con *fio* mostraron que las operaciones de escritura secuencial realizadas por 4 procesos con 4 threads llegaban a un máximo de **497 MiB/s** y una latencia de 32 mseg para un tamaño de bloque de 1 MiB, y que la lectura secuencial obtenía su

máximo también usando bloques de 1 MiB, con un throughput de **535 MiB/s** y una latencia de 29.8 mseg.

Las operaciones aleatorias registraron un máximo en bloques de 256 KiB, con 1**30 MiB/s** y latencia de 7.8 mseg para las escrituras, y de **373 MiB/s** y latencia de 7.9 mseg.

Al mantener fijo el tamaño de bloque en 4 KiB, y evaluar las IOPS y latencia a medida que se aumenta el número de operaciones encoladas, vemos que en las escrituras, entre 1 y 32 operaciones, las iops se mantienen exactamente iguales en 54000, luego saltan a 73000 con queue_depth = 64 y a 92000 con queue_depth=128. La latencia, en cambio, registra una curva con crecimiento geométrico. Un comportamiento similar se registra para las operaciones de lecturas, con meseta de IOPS en 81000 y un máximo de 116000 para queue_depth=128.

En cambio, en las operaciones de escrituras aleatorias, a partir de queue_depth=4 se alcanza el máximo de 53000 para escrituras y agregar operaciones encoladas sólo aumenta la latencia. Las lecturas aleatorias alcanzan su máximo de 82000 IOPS en queue_depth=32.

Al estresar el disco realizando testeos con procesos concurrentes crecientes, y aumentos progresivos de queue_depth, vemos que el disco SSD mantiene su máximo de IOPS entre 4 y 64 operaciones, y que su performance es muy buena entre 1 y 16 procesos. La latencia se dispara cuando el resultado de multiplicar procesos por operaciones es mayor a 128 (2*64 = 4*32 = 8*16…)

10.3. Los OSD

Los OSD, configurados con Bluestore en un único disco (sin separar WAL ni DB en otro dispositivo), fueron testeados con *ceph tell osd bench*.

Esta herramienta realiza operaciones con bloques de 4 MiB, y se observó que el OSD sobre discos HDD obtiene un throughput de **148 MiB/s**, con 115 iops y completó la tarea en 13.84 segundos. Los OSD sobre discos SSD lo hicieron en 4,42 segundos, con un throughput de **462 MiB/s.**

10.4. RADOS

Las pruebas se realizan utilizando *rados bench* en pools configurados sobre dispositivos HDD y SSD, con replica n=3, y sobre un pool sobre discos HDD configurado con *erasure code*, con k=5 y m=3.

El pool replicado sobre discos HDD registra un throughput de **395 MiB/s** y una latencia de 161 milisegundos en operaciones de escrituras, **534 MiB/s** y 118 milisegundos para las lecturas secuenciales y **505 MiB/s** y 126 milisegundos para lecturas random sobre los mismos objetos.

El pool replicado sobre discos SSD registra para las mismas pruebas estos valores: **1083 MiB/s** y latencia de 59 milisegundos para escrituras, **563 MiB/s** y latencia de 112 milisegundos para lecturas secuenciales y **560 MiB/s** con 113 milisegundos para lecturas aleatorias.

En el pool configurado con erasure coding, los números son de **448 MiB/s** y 149 milisegundos para las escrituras, **557 MiB/s** con 114 milisegundos para las lecturas secuenciales y **523 MiB/s** con 121 milisegundos para lecturas random.

Vemos que la diferencia importante está en el throughput y latencia que el pool sobre discos SSD registra en las operaciones de escritura.

Al implementar el esquema de *cache-tiering* los valores de escritura registran una performance menor (de 385 MiB/s se pasa a 345 MiB/s) , y los de lectura ramdom una mejora, pasando de 505 MiB/s a 548 MiB/s.

Los esquemas de compresión no se aplican en esta batería de pruebas porque están destinados a optimizar el espacio de almacenamiento en detrimento de la performance, que es del orden del 10%.

Los dispositivos de bloque RADOS (RBD) fueron testeados con la herramienta *fio* con un tamaño de bloque de 4 MiB (en concordancia con el tamaño de los objetos) en 1 proceso realizando 16 operaciones encoladas para un total de 2 GiB de datos. Las escrituras secuenciales registran un throughput de **321 MiB/s** y **842 MiB/s** para imágenes sobre pools HDD y SSD, respectivamente. Las lecturas secuenciales son similares en ambos con **621 y 632 MiB/s**, y las operaciones de lectura y escrituras random (en un patrón aleatorio de 50:50), registran **546 MiB/s** para RBD sobre pool HDD y **1010 MiB/s** para RBD sobre SSD.

10.5. Los clientes

Si se utiliza *librbd*, se configuran los clientes para que reserven 64 MiB de memoria como caché. Si se utiliza *krbd*, el módulo de kernel no mira la configuración de caché del cliente de ceph.

- Las pruebas de escritura secuencial con *dd* entregan un throughput de **456 MiB/s** y **797 MiB/s** en unidades sobre pools HDD y SSD.
- Los RBD sobre discos HDD tienen su techo en 945 IOPS de 4 KiB y responden con una latencia mejor frente a 4 procesos que encolan 32 operaciones cada uno (140 milisegundos)
- Los RBD sobre discos SSD tienen su mejor performance con 8 procesos concurrentes, cada uno con 32 operaciones: 23000 IOPS para una latencia de 11 milisegundos. Y valores similares registran 32 procesos con 4 operaciones cada uno para 22000 IOPS y una latencia de 5.75 milisegundos.

El hardware y sistema operativo que mapean los RBD en el cliente local tiene un impacto significativo en la performance: el número de procesadores, la cantidad de memoria disponible y la velocidad de la interfaz de red son factores limitantes al throughput e introducen un aumento en la latencia de cada operación.

Un servidor físico con pocos recursos puede tener una disminución de la performance de hasta el 10%. Y en un servidor virtualizado, la merma es del 25% al 30%. La incidencia del hipervisor en el rendimiento del almacenamiento es un punto importante a tener en cuenta a la hora de integrar Ceph con la infraestructura actual.

11. CONCLUSIONES y TRABAJOS FUTUROS

La performance de los sistemas de almacenamiento distribuido es un tópico que preocupa a cualquier equipo que decide implementarlo en su infraestructura de cómputo. Típicamente, la decisión responde al deseo de otorgar al almacenamiento escalabilidad, flexibilidad, alta disponibilidad, tolerancia a fallas, resiliencia, gestión, monitoreo, integración, convergencia y seguridad. Y Ceph ofrece todas esas características y varias más.

La implementación realizada en la UNNOBA proveyó todas las características que se buscaban, y la integración con los sistemas de virtualización mencionados -Xen y Kubernetes- fue directa, con una transición sin sobresaltos. Las aplicaciones y servicios de almacenamiento “histórico”, como las resoluciones del Consejo Superior, la biblioteca de medios del área de Prensa y Comunicaciones, el FTP, el TFTP, etc… se almacenan en pools sobre discos magnéticos; y los servicios que requieren una E/S con mejor latencia, FileServer, emails, nubes privadas, etc... se almacenan en pools sobre discos SSD, con tiempos de respuesta acotados en diferentes cargas de trabajo. Como trabajo futuro, se proyecta la integración de Ceph como almacenamiento de volúmenes, imágenes y backups del servicio Cinder y Glance de Openstack (versión Wallaby), proyecto que está en etapa de desarrollo y pruebas en la Universidad.

Ahora, cuando se habla de performance ya sabemos que hay alguna “basurita en el carburador” porque -de entrada- tenemos que meternos en el terreno de la semántica y definir qué se entiende por performance… ¿La performance de un cliente accediendo a un único recurso? ¿La de muchos clientes accediendo a un único recurso? ¿La de muchos clientes accediendo a muchos recursos? Y ese acceso ¿es mayormente aleatorio? ¿Son grandes transferencias de datos en un único stream o pequeñas? ¿Se ejecutan en batch? La aplicación que realiza la E/S contra el cluster ¿es asincrónica o se queda esperando la respuesta para continuar con la ejecución?

Todas estas preguntas justifican el esfuerzo por tipificar los distintos patrones de acceso, y tratar de medir la performance del cluster en términos de latencia de cada operación, en el número de operaciones que pueden realizarse por segundo y en el volumen de datos que pueden transferirse desde y hacia el almacenamiento.

Los testeos realizados sobre los discos magnéticos y de estado sólido ponen en número la experiencia de que los últimos son “realmente más rápidos” que los primeros. Y que esa velocidad nos hace mirar con recelo a los HDD, buscando en nuestras cabezas argumentos para mantenerlos como opción en nuestras instalaciones.

Pero es claro que no puede pedirse a un software de almacenamiento distribuido que tenga la misma performance que un único disco conectado localmente. Al menos no para un único cliente contra un único recurso. No es ese el escenario para el cual estas soluciones de almacenamiento definido por software fueron diseñadas. La red y las varias capas de código que proveen las características que mencionamos al principio tienen su costo reflejado en el aumento de la latencia y en la disminución del throughput.

No obstante, Ceph hace un buen trabajo. La paralelización y distribución de los accesos en los distintos OSD logra valores incluso mejores que en los discos locales (sobre todo cuando éstos son HDD) y esa performance se mantiene cuando aumenta la carga de trabajo. Los patrones de acceso, que en los clientes pueden ser flujos secuenciales, al ser divididos en diferentes objetos, y éstos terminar en distintos placement groups, que a su vez se graban/leen de distintos OSD, terminan siento aleatorios: un OSD recibe distintos requests de distintos clientes a distintos objetos de manera random todo el tiempo. Por eso, cuantos más OSD tiene el cluster, mejor tiempo de respuesta, porque cada OSD atiende menos requerimientos por segundo. El cluster escala linealmente cuando se agregan OSDs y su latencia conjunta disminuye porque las operaciones se paralelizan y distribuyen entre más nodos. Además, hay que tener en cuenta que todas las escrituras en Ceph son transaccionales, con lo cual las escrituras random de pocos bytes hechas por procesos sincrónicos, son las operaciones más costosas en términos de performance.

Vimos que los SSD tienen mejor performance que los HDD, pues carecen de las latencias que el movimiento de brazo y la rotación de los platos introduce en las operaciones; y que los dispositivos NVMe tienen mejor performance que los SSD, porque su bus de acceso no es SATA, sino PCI-X. Este es otro problema que enfrentan quienes pueden acceder a ese nuevo hardware: *Ceph es lento con dispositivos NVMe*. Como se analizó en los puntos 6.4 y 6.5, ese problema todavía está monetariamente muy lejos de aparecer en nuestro horizonte.

A la hora de adquirir los SSD para nuestros OSD, debemos asegurarnos de que sean "*enterprise grade*". Ceph no utiliza ningún buffer de escritura. Cada operación de escritura es transaccional y no se completan hasta que están registradas en los journals de los OSDs y sincronizadas al disco. Pero los discos SSD tienen cachés internas para organizar las lecturas/escrituras de manera de mejorar la performance, y además tienen baterías y protecciones que aseguran poder completar esas operaciones frente a un fallo de energía. Algunos SSD más baratos, no poseen estas protecciones y, sin embargo, devuelven el OK cuando esa operación está en la caché del dispositivo, no cuando está en el almacenamiento.

Una vez analizados los discos y la red de datos, el proceso de benchmarking se vuelve bastante artesanal, conforme se va subiendo de niveles. La elección de tamaño de bloques, concurrencia de procesos, longitud de operaciones enviadas al driver, sincronía o asincronía, dependen del patrón de tráfico y la carga de trabajo que quiera simularse, si quiere evaluarse el uso en situaciones normales, o quiere estresarse el sistema. Por esto las configuraciones que se hacen luego en el cluster, en pos de mejorar la performance, buscan una optimización general, que aplique a *varios* patrones de tráfico, con una carga *normal* estimada en el cluster. Como ya lo dice la bibliografía en múltiples lugares: no hay una receta única que funcione para todo.

4 Mib es el tamaño por defecto para fragmentar los datos, pero puede ser configurado al momento de crear los pools, pues con objetos pequeños puede proveer mejor performance en algunas operaciones, a costo de tener mayor número de objetos, y la *fragmentación interna* es menor. La latencia de escribir objetos más pequeños es menor que escribiendo objetos grandes en pooles sobre discos SSD, pero en pooles sobre discos

HDD, 4MiB tiene una mejor performance, porque una vez posicionado el cabezal, realiza una operación secuencial sobre un bloque de mayor tamaño. Pero como Bluestore utiliza *extents* internamente, cuando se escriben 4KB en un objeto de 4MB en una imagen RBD, no necesariamente la operación resulta en una escritura de 4MB, por eso el tamaño de bloque de 1MB o 4MB tienen una performance similar.

Así, comprobamos que una estrategia que funciona es definir reglas CRUSH de almacenamiento que separen los dispositivos: una que almacene exclusivamente en discos SSD y otra en discos HDD; y de esta manera poder definir pooles sobre uno y otro tipo, y destinar esos pools a las aplicaciones con características específicas que aseguren su mejor performance, de acuerdo al patrón de tráfico esperado. Un servidor FTP para almacenamiento de registros audiovisuales históricos de la organización perfectamente puede almacenarse en un pool sobre discos HDD, con tamaño de bloques grandes, lo mismo que un servidor web con poca actividad, o una aplicación que almacena documentos que son raramente modificados.

En cambio, una aplicación de monitoreo que lanza procesos concurrentes que levantan vía SNMP el estado de múltiples dispositivos y los guarda en una base de datos, sufriría mucho si no se almacena sobre discos SSD. O plataformas web con accesos de múltiples clientes que envían y reciben archivos de manera concurrente. O proxys web y servidores de correo, que escriben y leen miles de archivos de pequeño tamaño.

Comprobamos además que implementar el modelo de *cache tiering* con un pool pequeño de discos SSD que funciona de caché de un pool con más capacidad de discos HDD no tiene una buena performance en el servicio de volúmenes RBD utilizado. El esquema tal vez funcione bien cuando se pueden cachear objetos completos de Rados Gateway, o archivos de CephFS. Pero para RBD, los objetos modificados en cada operación son muchos, tal vez no completamente, pero modificar unos pocos bytes en un objeto, implica traer los 4 MiB y la caché se renueva con demasiada rapidez, de modo que en la lectura los accesos a caché dan *MISS* la mayoría de las veces, y el mecanismo debe ir a buscar los datos a la capa subyascente. La bibliografía oficial de Ceph lo aclara de entrada: "*Cache tiering will degrade performance for most workloads. Users should use extreme caution before using this feature.*" Como si no tuviéramos suficiente adrenalina en este trabajo…

No obstante, como tarea futura, en varios trabajos se menciona el uso del *kernel block layer cache* de Linux (bcache, 2020) como esquema de optimización de performance de Ceph. B-cache permite el uso de un dispositivo SSD como caché de lecto-escritura (writeback) o de lectura (writethrough/writearoung) de un disco HDD. Y, en la misma línea de generar dispositivos "híbridos", también queda pendiente probar la separación del journal de Bluestore (Write-Ahead Log) configurándolo sobre un disco SSD.

Otro punto a probar es habilitar *jumbo frames* en toda la red de datos. Hemos hallado opiniones encontradas al respecto: la mejora se registra en los procesos de recovery y backfilling que realizan los nodos entre sí, cuando los OSD rebalancean o recuperan PGs, pero no se observan mejoras sustanciales en el tráfico desde y hacia los clientes del cluster. Habrá que configurar y probar, pero son decisiones difíciles de implementar cuando se tiene una infraestructura ya operativa.

Si la carga de trabajo está muy atada a la E/S, Ceph no es una buena opción, sobre todo cuando es realizada por un único proceso/thread, con una cola = 1 (sincrónico), y bloques de 4K. Con bloques grandes y multi-thread la performance mejora mucho.

Sintetizando, podemos decir que en la mayoría de los patrones de acceso, Ceph mantiene una buena performance frente a la escalabilidad de usuarios y recursos, permite una flexibilidad moderada mediante la configuración de *pools* adaptados a las aplicaciones que los utilizan, y cumple con todo lo que promete: un almacenamiento unificado, flexible, confiable y consistente, basado en software libre, con una comunidad detrás que lo desarrolla, soporta e impulsa a extender sus límites.-

12. REFERENCIAS BIBLIOGRÁFICAS

Abd-El-Malek, M., Courtright, W. V, Cranor, C., Ganger, G. R., Hendricks, J., Klosterman, A. J., Mesnier, M., Prasad, M., Salmon, B., Sambasivan, R. R., Sinnamohideen, S., Strunk, J. D., Thereska, E., Wachs, M., & Wylie, J. J. (2005). *Ursa Minor: versatile cluster-based storage*.

Aghayev, A., Weil, S., Kuchnik, M., Nelson, M., Ganger, G. R., & Amvrosiadis, G. (2019). File systems unfit as distributed storage backends: Lessons from 10 years of Ceph evolution. SOSP 2019 - Proceedings of the 27th ACM Symposium on Operating Systems Principles, 353–369. https://doi.org/10.1145/3341301.3359656

Amazon EBS Volume Types - *Amazon Elastic Compute Cloud*. Retrieved November 22, 2020, from https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ebs-volume-types.html

Azagury, A., Dreizin, V., Factor, M., Henis, E., Naor, D., Rinetzky, N., Rodeh, O., Satran, J., Tavory, A., & Yerushalmi, L. (2003). *Towards an object store. Proceedings - 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies, MSST 2003*, 165–176. https://doi.org/10.1109/MASS.2003.1194853

Bainbridge, J., & Maxwell, J. (2014). *Red Hat Enterprise Linux 7 Performance Tuning Guide*. 1–76. www.redhat.com

Barkes, J., & et al. (1998). GPFS: a parallel file system. *IBM International Technical Support Organization*. http://www.redbooks.ibm.com

bcache: Kernel Block Layer Cache. (2020). Retrieved November 30, 2020, from https://www.kernel.org/doc/html/latest/admin-guide/bcache.html

Bjørling, M. (2019). From Open-Channel SSDs to Zoned Namespaces Forward-Looking Statements. January. https://www.usenix.org/conference/vault19/presentation/bjorling

Block Devices and Kubernetes — *Ceph Documentation*. (2020). Retrieved November 23, 2020, from https://docs.ceph.com/en/latest/rbd/rbd-kubernetes/

BlueStore Config Reference. (2020). Retrieved November 29, 2020, from https://docs.ceph.com/en/latest/rados/configuration/bluestore-config-ref/

btrfs Wiki. (2020). Retrieved October 12, 2020, from https://btrfs.wiki.kernel.org/index.php/Main_Page

Castillo, J. A. (2020). *PMR, CMR y SMR: Qué es y cómo escribe los datos tu disco duro ??* https://www.profesionalreview.com/2020/05/11/pmr-cmr-y-smr-que-es/

Ceph. (2020). Ceph Homepage - Ceph. https://ceph.io/

Ceph Cache Tiering Introduction. Retrieved November 16, 2020, from https://docs.ceph.com/en/latest/rados/operations/cache-tiering/

Ceph Crimson. (2020). Retrieved November 20, 2020, from https://www.usenix.org/sites/default/files/conference/protected-files/vault20_slides_just.pdf

Ceph New in Luminous: BlueStore - Ceph. (2017). Retrieved November 23, 2020, from https://ceph.io/community/new-luminous-bluestore/

Chamarthy, M. (2020). Introduction to Client-side Caching in Ceph. | USENIX. (n.d.). Retrieved October 22, 2020, from https://www.usenix.org/conference/vault20/presentation/chamarthy

Collet, J., Ster, D. Van Der, Cameselle, R. V., & Lamanna, M. (2019). *Characterization of OSD performance in a Ceph cluster*.

Coulouris, G., Dollimore, J. & Kindberg, T. (1994). *Distributed systems : concepts and design. Wokingham, England Reading, Mass: Addison-Wesley.*

Dahlin, M. D., Mather, C. J., Wang, R. Y., Anderson, T. E., & Patterson, D. A. (1994). *A quantitative analysis of cache policies for scalable network file systems.* ACM SIGMETRICS Performance Evaluation Review, 22(1), 150-160.

Darabseh, A., Al-Ayyoub, M., Jararweh, Y., Benkhelifa, E., Vouk, M., & Rindos, A. (2015). SDStorage: A software defined storage experimental framework. *Proceedings - 2015 IEEE International Conference on Cloud Engineering, IC2E 2015*, 341–346. https://doi.org/10.1109/IC2E.2015.60

Dugan, J., Estabrook, J., Ferbuson, J., Gallatin, A., Gates, M., Gibbs, K., Hemminger, S., Jones, N., Qin, F., Renker, G., Tirumala, A., & Warshavsky, A. (2010). *iPerf - The TCP, UDP and SCTP network bandwidth measurement tool*. https://iperf.fr/

Engler, D. R., Kaashoek, M. F., & O'Toole, J. (1995). Exokernel: An operating system architecture for application-level resource management. Proceedings of the 15th ACM Symposium on Operating Systems Principles, SOSP 1995, 251–266. https://doi.org/10.1145/224056.224076

Erasure Code - *Ceph Documentation*. (2016). Retrieved November 23, 2020, from https://docs.ceph.com/en/latest/rados/operations/erasure-code/

ext4 Data Structures and Algorithms — The Linux Kernel documentation. (2020). The Linux Kernel Documentation. Retrieved October 12, 2020, from https://www.kernel.org/doc/html/latest/filesystems/ext4/index.html

Filippov, V. (2019). *Ceph performance — YourcmcWiki*. Retrieved November 24, 2020, from https://yourcmc.ru/wiki/Ceph_performance

Filippov, V. (2020) *vitalif/vitastor: Simplified distributed block storage with strong consistency, like in Ceph - README.md at master - vitastor - Gitea*. Retrieved November 7, 2020, from https://yourcmc.ru/git/vitalif/vitastor/src/branch/master/README.md

FIO (2020) *FIO's documentation — fio 3.23-48-ga0b72-dirty documentation*. (n.d.). Retrieved November 22, 2020, from https://fio.readthedocs.io/en/latest/

Fischer, M. J., Lynch, N. A., & Paterson, M. S. (1985). Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM (JACM)*, *32*(2), 374–382. https://doi.org/10.1145/3149.214121

Fisk, N. (2019). Mastering Ceph - Second Edition. Birmingham, England: Packt Publishing.

GeeksforGeeks. (2020). *Copy on Write* - https://www.geeksforgeeks.org/copy-on-write/

Ghemawat, S., Gobioff, H., & Leung, S.-T. (2003). *The Google file system*. 29. https://doi.org/10.1145/945445.945450

Gifford, D. K., Needham, R. M., & Schroeder, M. D. (1988). The Cedar file system. *Communications of the ACM*, *31*(3), 288–298. https://doi.org/10.1145/42392.42398

Gilbert, S., & Lynch, N. (2002). Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. ACM SIGACT News, 33(2), 51–59. https://doi.org/10.1145/564585.564601

Gluster (2016). Gluster: Storage for your Cloud. https://www.gluster.org/

Gooch, R., Enberg, P. (1998). *Overview of the Linux Virtual File System — The Linux Kernel documentation*. https://www.kernel.org/doc/html/latest/filesystems/vfs.html

Groom, B. (2020). *How do LSM Trees work?* Retrieved November 24, 2020, from https://yetanotherdevblog.com/lsm/

Gudu, D., Hardt, M., & Streit, A. (2015). Evaluating the performance and scalability of the Ceph distributed storage system. *Proceedings - 2014 IEEE International Conference on Big Data, IEEE Big Data 2014*, 177–182. https://doi.org/10.1109/BigData.2014.7004229

Hackett, M. (2019). Ceph : Designing and implementing scalable storage systems. Birmingham, England: Packt Publishing.

Hard Drive – *Performance, transfer rates, latency and seek times*. pctechguide.com. Retrieved November 23, 2020, from http://www.pctechguide.com/hard-disk-hard-drive-performance-transfer-rates-latency-and-seek-times

Heigl, F. (2020). *GitHub - FlorianHeigl/xen-ceph-rbd: Lets you run Xen VMs on Ceph RBD images without hassle or libvirt*. Retrieved November 16, 2020, from https://github.com/FlorianHeigl/xen-ceph-rbd

Hertel, C. (2004). *Implementing CIFS*. Upper Saddle River, N.J.: Prentice Hall PTR.

Howard, J. H., Kazar, M. L., Menees, S. G., Nichols, D. A., Satyanarayanan, M., Sidebotham, R. N., & West, M. J. (1988). Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems (TOCS)*, *6*(1), 51–81. https://doi.org/10.1145/35037.35059

Howard, S. (2015). *Queue Depth, IOPS and Latency*. Scott's Blog. https://blog.docbert.org/queue-depth-iops-and-latency/

IEEE. (2009). IEEE Standard for Prefixes for Binary Multiples. *IEEE Std 1541-2002 (R2008)*, c1-4. https://standards.ieee.org/standard/1541-2002.html

Intel. (2018). *Leveraging RDMA Technologies to Accelerate Ceph Storage Solutions*. https://software.intel.com/content/www/us/en/develop/articles/leveraging-rdma-technologies-to-accelerate-ceph-storage-solutions.html

Interrupt coalescing - Wikipedia. (2020). Retrieved November 23, 2020, from https://en.wikipedia.org/wiki/Interrupt_coalescing

Jiangang, D. (2013). *Measure Ceph RBD performance in a quantitative way (part I)*. https://software.intel.com/content/www/us/en/develop/blogs/measure-ceph-rbd-performance-in-a-quantitative-way-part-i.html

Johari, J. J., Khalid, M. F., Nizam, M., Mydin, M., & Wijee, N. (2014). Comparison of Various Virtual Machine Disk Images Performance on GlusterFS and Ceph Rados Block Devices. 1–7. https://www.researchgate.net/publication/266477355_Comparison_of_Various_Virtual_Machine_Disk_Images_Performance_on_GlusterFS_and_Ceph_Rados_Block_Devices

Kernel Documentation. (2020). Retrieved November 22, 2020, from https://www.kernel.org/doc/Documentation/sysctl/vm.txt

Kronenberg, N. P., Levy, H. M., & Strecker, W. D. (1986). VAXcluster: A Closely-Coupled Distributed System. *ACM Transactions on Computer Systems (TOCS)*, *4*(2), 130–146. https://doi.org/10.1145/214419.214421

Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., ... & Wells, C. (2000). *Oceanstore: An architecture for global-scale persistent storage.* ACM SIGOPS Operating Systems Review, 34(5), 190-201.

KVM. (2020). Retrieved November 22, 2020, from https://www.linux-kvm.org/page/Main_Page

Lamport, L., Shostak, R., & Pease, M. (1982). The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, *4*(3), 382–401. https://doi.org/10.1145/357172.357176

Lamport, L. (2001). Paxos Made Simple. *ACM SIGACT News*, *32*(4), 51–58.

Lamport, L. (2005). *Generalized Consensus and Paxos*. *April*, 60. http://research.microsoft.com/apps/pubs/default.aspx?id=64631

Lee, D. Y., Jeong, K., Han, S. H., Kim, J. S., Hwang, J. Y., & Cho, S. (2017). Understanding write behaviors of storage backends in ceph object store. In Proceedings of the 2017 IEEE International Conference on Massive Storage Systems and Technology (Vol. 10).

Liskov, B., Ghemawat, S., Gruber, R., Johnson, P., Shrira, L., & Williams, M. (1991). Replication in the harp file system. *ACM SIGOPS Operating Systems Review*, *25*(5), 226–238. https://doi.org/10.1145/121133.121169

Love, R. (2005). *The Dentry Object*. Linux Kernel Development Second Edition. http://books.gigatux.nl/mirror/kerneldevelopment/0672327201/ch12lev1sec7.html

Lowrentius. (2020). *louwrentius/fio-plot: Create charts from FIO storage benchmark tool output*. https://github.com/louwrentius/fio-plot

Lustre. (2020). Retrieved November 22, 2020, from https://www.lustre.org/

McKusick, M. K., Joy, W. N., Leffler, S. J., Fabry, R. S. (1984). A fast file system for UNIX. *ACM Transactions on Computer Systems (TOCS)*, *2*(3), 181–197. https://doi.org/10.1145/989.990

McKusick, M. K., & Ganger, G. R. (1999). *THE ADVANCED COMPUTING SYSTEMS ASSOCIATION Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem*.

Mellanox Technologies. (2019). *Understanding Interrupt Moderation*. Knowledge Article. https://community.mellanox.com/s/article/understanding-interrupt-moderation

Meredith, R. (2018). *Ceph BlueStore vs. FileStore: Block performance comparison when leveraging Micron NVMe SSDs.* Retrieved November 14, 2020, from https://www.micron.com/about/blog/2018/may/ceph-bluestore-vs-filestoreblock-performance-comparison-when-leveraging-micron-nvme-ssds

Meyer, S., & Morrison, J. P. (2016). Supporting Heterogeneous Pools in a Single Ceph Storage Cluster. *Proceedings - 17th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2015*, 352–359. https://doi.org/10.1109/SYNASC.2015.61

MySQL :: MySQL 8.0 Reference Manual :: 8.12.1 Optimizing Disk I/O. (2020). Retrieved November 22, 2020, from https://dev.mysql.com/doc/refman/8.0/en/disk-issues.html

Oddeye. (2019). *Ceph: Why to Use BlueStore*. Retrieved November 23, 2020, from https://www.oddeye.co/blog/ceph-why-to-use-bluestore/

ONF. (2015). OpenFlow Switch Specification Version 1.5.1 (Protocol version 0x06) for information on specification licensing through membership agreements. 1, 283. http://www.opennetworking.org

Ousterhout, J., & Douglis, F. (1989). Beating the I/O bottleneck: A case for log-structured file systems. *Operating Systems Review (ACM)*, *23*(1), 11–28. https://doi.org/10.1145/65762.65765

Piernas, J., Cortes, T., & García, J. M. (2002). DualFS. *Proceedings of the 16th International Conference on Supercomputing - ICS '02*, 137. https://doi.org/10.1145/514191.514213

Placement Groups — Ceph Documentation. (2020). Retrieved November 22, 2020, from https://docs.ceph.com/en/latest/rados/operations/placement-groups/

Placement Group Concepts — Ceph Documentation. (2020) Retrieved November 24, 2020, from https://docs.ceph.com/en/latest/rados/operations/pg-concepts

Poat, M. D., & Lauret, J. (2017). Achieving cost/performance balance ratio using tiered storage caching techniques: A case study with CephFS. Journal of Physics: Conference Series, 898(6). https://doi.org/10.1088/1742-6596/898/6/062022

Quinlan, S., & Dorward, S. (2002). Venti: A new approach to archival storage. *Proceedings of the FAST 2002 Conference on File and Storage Technologies*. http://doc.cat-v.org/plan_9/4th_edition/papers/venti/

Ra, M.-R. (2018). *Understanding the Performance of Ceph Block Storage for Hyper-Converged Cloud with All Flash Storage*. https://www.researchgate.net/publication/323355785_Understanding_the_Performance_of_Ceph_Block_Storage_for_Hyper-Converged_Cloud_with_All_Flash_Storage

Rancher. (2020). *Enterprise Kubernetes Management | Rancher*. https://rancher.com/

rbd cache mode (2020). Retrieved November 23, 2020, from http://people.redhat.com/bhubbard/nature/default/rbd/rbd-config-ref/

rbd CSI plugin. (2020). Retrieved November 23, 2020, from https://github.com/ceph/ceph-csi/blob/master/docs/deploy-rbd.md

Red Hat. (2019). *¿Qué es el almacenamiento definido por software?* https://www.redhat.com/es/topics/data-storage/software-defined-storage

Red Hat. (2020). *Red Hat Gluster Storage*. Retrieved November 22, 2020, from https://www.redhat.com/es/technologies/storage/gluster

RocksDB. (2020) *A persistent key-value store | RocksDB*. Retrieved November 23, 2020, from https://rocksdb.org/

Saito, Y., & Karamanolis, C. (2002, July). Pangaea: a symbiotic wide-area file system. In Proceedings of the 10th workshop on ACM SIGOPS European workshop (pp. 231-234).

Saito, Y., Frølund, S., Veitch, A., Merchant, A., & Spence, S. (2004). FAB. *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS-XI*, 48. https://doi.org/10.1145/1024393.1024400

Satyanarayanan, M. (1990). Scalable, Secure, and Highly Available Distributed File Access. *Computer*, *23*(5), 9–18. https://doi.org/10.1109/2.53351

Schindler, J. (2004). *Matching Application Access Patterns to Storage Device Characteristics*.

Shankar, V., & Lin, R. (2017, June). Performance study of ceph storage with Intel cache acceleration software: Decoupling hadoop mapreduce and hdfs over ceph storage. In 2017 IEEE 4th International Conference on Cyber Security and Cloud Computing (CSCloud) (pp. 10-13). IEEE.

Shen, K., Park, S., & Zhu, M. (2014). Journaling of Journal Is (almost) free. Proceedings of the 12th USENIX Conference on File and Storage Technologies, FAST 2014, 287–293. https://www.usenix.org/conference/fast14/technical-sessions/presentation/shen

Shepler, S., Callaghan, B., Robinson, D., Thurlow, R., Beame, C., Eisler, M., & Noveck, D. (n.d.). *RFC 7530 - Network File System (NFS) Version 4 Protocol*. Retrieved October 12, 2020, from https://tools.ietf.org/html/rfc7530

Silberschatz, A., & Levy, E. (1990). Distributed File Systems: Concepts and Examples. *ACM Computing Surveys (CSUR)*, *22*(4), 321–374. https://doi.org/10.1145/98163.98169

Silicon Graphics Inc. (2020). *XFS Filesystem Structure*. Retrieved October 12, 2020, from https://xfs.org/docs/xfsdocs-xml-dev/XFS_Filesystem_Structure/tmp/en-US/html/index.html

Singh, K., & Parkes, D. (2019). *Ceph Part - 1 : BlueStore (Default vs. Tuned) Performance Comparison - Ceph*. Retrieved November 16, 2020, from https://ceph.io/community/bluestore-default-vs-tuned-performance-comparison/

Singh, K. (2019). *Red Hat Ceph Storage 3.3 BlueStore compression performance*. https://www.redhat.com/en/blog/red-hat-ceph-storage-33-bluestore-compression-performance

SoftNetStats - file /proc/net/softnet_stat — insights-core 3.0.8 documentation. (2020). Retrieved November 23, 2020, from https://insights-core.readthedocs.io/en/latest/shared_parsers_catalog/softnet_stat.html

Songbin Liu, Xiaomeng Huang, Haohuan Fu, & Guangwen Yang. (2013). *Understanding Data Characteristics and Access Patterns in a Cloud Storage System*. 327–334. https://doi.org/10.1109/ccgrid.2013.11

Stallings, William. (2018). *Operating Systems : Internals and Design Principles.* Global Edition. Upper Saddle River, N.J. :Prentice Hall.

Stonebraker, M. (1981). Operating System Support for Database Management. Communications of the ACM, 24(7), 412–418. https://doi.org/10.1145/358699.358703

Sweeney, A., Doucette, D., Hu, W., Anderson, C., Nishimoto, M., & Peck, G. (1996). *Scalability in the XFS file system* ATEC '96: Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference. https://dl.acm.org/doi/10.5555/1268299.1268300

Tanenbaum, A. & Steen, M. (2016). *Distributed systems : principles and paradigms.* Niederlande: Maarten van Steen.

Tang, H., Gulbeden, A., Zhou, J., Strathearn, W., Yang, T., & Chu, L. (2004). A self-organizing storage cluster for parallel data-intensive applications. *Proceedings of the ACM/IEEE SC 2004 Conference: Bridging Communities*. https://doi.org/10.1109/SC.2004.9

The Apache Software Foundation. (2015). *HDFS Architecture Guide*. Hadoop Apache Project. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

The story of one latency spike. (2015). Retrieved November 23, 2020, from https://blog.cloudflare.com/the-story-of-one-latency-spike/

Thekkath, C. A., & Lee, E. K. (1996). *Petal: Distributed Virtual Disks.*

Thekkath, C. A., Mann, T., & Lee, E. K. (1997). *Frangipani. Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles - SOSP '97, 224–237. https://doi.org/10.1145/268998.266694*

Thereska, E., Ballani, H., O'Shea, G., Karagiannis, T., Rowstron, A., Talpey, T., Black, R., & Zhu, T. (2013). IOFlow: A software-defined storage architecture. *SOSP 2013 - Proceedings of the 24th ACM Symposium on Operating Systems Principles*, 182–196. https://doi.org/10.1145/2517349.2522723

Umrao, V., Hackett, M., & Singh, K. (2017). *Ceph Cookbook - (2nd ed.). Birmingham, England: Packt Publishing.*

van der Ster, D., & Wiebalck, A. (2014). Building an organic block storage service at CERN with Ceph. *Journal of Physics: Conference Series*, *513*(TRACK 4). https://doi.org/10.1088/1742-6596/513/4/042047

Wang, F., Oral, S., Shipman, G., Drokin, O., Wang, T., & Huang, I. (2009). *Understanding lustre filesystem internals*. Oak Ridge National Laboratory, National Center for Computational Sciences, Tech. Rep.

Wang, H., & Yang, Z. (2016). *Accelerate Ceph via SPDK XSKY ' s BlueStore as a case study*.

Wang, R., & Anderson, T. (1993). *xFS: A Wide Area Mass Storage File System* | EECS at UC Berkeley. https://www2.eecs.berkeley.edu/Pubs/TechRpts/1993/6321.html

Weil, S. A., Brandt, S. A., Miller, E. L., & Maltzahn, C. (2006). CRUSH: Controlled, scalable, decentralized placement of replicated data. Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, SC'06. https://doi.org/10.1145/1188455.1188582

Weil, S. (2007). CEPH: Reliable, Scalable and High-Performance Distributed Storage. Retrieved May 10, 2020, from https://ceph.com/wp-content/uploads/2016/08/weil-thesis.pdf

Weil, S. A., Leung, A. W., Brandt, S. A., & Maltzahn, C. (2008). RADOS: A scalable, reliable storage service for petabyte-scale storage clusters. Proceedings of the 2nd International Petascale Data Storage Workshop, PDSW '07, Held in Conjunction with Supercomputing '07, 35–44. https://doi.org/10.1145/1374596.1374606

Weil, S. A. (2019). Ceph Tech Talk - Intro to Ceph - YouTube. https://www.youtube.com/watch?list=PLrBUGiINAakM36YJiTT0qYepZTVncFDdc&v=PmLPbrf-x9g

Welch, B., Unangst, M., Abbasi, Z., Gibson, G. A., Mueller, B., Small, J., ... & Zhou, B. (2008, February). *Scalable Performance of the Panasas Parallel File System*. In FAST (Vol. 8, pp. 1-17).

Western Digital. (2020). Zoned Storage Overview - Zoned Storage. https://zonedstorage.io/introduction/zoned-storage/

Wong, T. M., Golding, R. A., Glider, J. S., Borowsky, E., Becker-Szendy, R. A., Fleiner, C., Kenchammana-Hosekote, D. R., & Zaki, O. A. (2005). *Kybos: Self-management for distributed brick-based storage*.

Wu, F., & Sun, G. (2013). *Software-Defined Storage*. http://fgwu.me/publications/techrepo/WS2013.pdf

Write Anywhere File Layout - Wikipedia. (2020. Retrieved October 12, 2020, from https://en.wikipedia.org/wiki/Write_Anywhere_File_Layout

Yang, C. T., Chen, C. J., & Chen, T. Y. (2017). Implementation of ceph storage with big data for performance comparison. In International Conference on Information Science and Applications (pp. 625-633). Springer, Singapore.

13. SIGLAS y NOMENCLATURA

AoE	Ata over Ethernet
CAP	Consistencia, Disponibilidad y Tolerancia al Particionado (Consistency, Availability, Partition tolerance)
CephFS	Sistema de Archivos de Ceph (Ceph File System)
CERN	Consejo Europeo para la Investigación Nuclear (Conseil Européen pour la Recherche Nucléaire)
CIFS	Common Internet File System
COW	Copiar al Escribir (Copy on Write)
HDD	Disco Rígido (Hard Disk Drive)
HPC	Cómputo de Alto Desempeño (High Performance Computing)
iSCSI	Interfaz de internet para sistemas de computo pequeño (Internet Small Computer System Interface)
LAN	Red de Área Local (Local Area Network)
LFS	Filesystem estructurado como registros (Log-structured File System)
MDS	Servidores de Metadatos (Metadata Servers)
NAS	Almacenamiento conectado a la red (Network Attached Storage)
NFS	File System de Red (Network File System)
NVMe	Memoria "exprés" no volátil (Non-Volatile Memory Express)
OSD	Dispositivo de almacenamiento de Objetos (Object Storage Device)
POSIX	Interfaz portable de Sistema Operativo (Portable Operating System Interface)
QoS	Calidad de Servicio (Quality of Service)
RADOS	Almacenamiento confiable y autónomo de objetos distribuidos (Reliable Autonomic Distributed Object Store)
RAID	Grupo redundante de discos independientes (redundant array of independent disks)
RBD	Bloque de Almacenamiento RADOS (RADOS Block Device)
RGW	Almacenamiento de Objetos RESTful API (Rados Gateway)
SAN	Red de Área de Almacenamiento (Storage Area Network)
SDS	Almacenamiento definido por software (Software Defined Storage)
SLA	Acuerdo de Nivel de Servicios (Service Level Agreement)
SPoF	Punto Único de Falla (Single Point of Failure)
SSD	Disco de Estado Sólido (Solid State Disk)
TIC	Tecnologías de la Información y la Comunicación
UNNOBA	Universidad Nacional del Noroeste de la provincia de Buenos Aires
VPN	Red Privada Virtual (Virtual Private Network)
WAN	Red de Área Amplia (Wide Area Network)

WAL	Registro de escritura anticipada (Write-Ahead Log)
WAF	Factor de amplificación de escritura (Write Amplification Factor)