Universidad de las Ciencias Informáticas

Centro de Informatización Universitaria

Facultad 1

PROPUESTA DE UN SISTEMA DE FICHEROS DISTRIBUIDO PARA LA RED SOCIAL UNIVERSITARIA DE LA UNIVERSIDAD DE LAS CIENCIAS INFORMÁTICAS

Trabajo de diploma para optar por el título de Ingeniero en Ciencias Informáticas

Autor: Julio Jesús Garcia Coste

Tutores: Ing. Alien Gongora Rodríguez
Ing. Daisy Guzmán Arias

Ciudad de La Habana, Cuba
Junio de 2012

DECLARACIÓN DE AUTORÍA

Declaro que soy el único autor de este trabajo y autorizo a la Universidad de las Ciencias Informáticas (UCI) de Ciudad de La Habana, para que hagan el uso que estimen pertinente con este trabajo.

Para que así conste, firmo la presente a los_____ días del mes de _____ de 2012.

_____

Julio Jesús Garcia Coste

_____ _____

Ing. Alien Gongora Rodríguez Ing. Daisy Guzmán Arias

Pensamiento

Todos y cada uno de nosotros paga puntualmente su cuota de sacrificio consciente de recibir el premio en la satisfacción del deber cumplido, conscientes de avanzar con todos hacia el Hombre Nuevo que se vislumbra en el horizonte…

Dedicatoria

A mi Papá especialmente, por haber sido mi guía y apoyo en la vida,

porque siempre estuvo ahí por mí y para mí,

y de alguna manera siempre estará…

Agradecimientos

A mis tutores y consultantes, especialmente a Daisy, por su dedicación, orientación y ayuda.

A mi familia en general, por el apoyo brindado para la realización de este trabajo, pero en especial a:

A mi papá (que Dios lo lleve en la gloria), por confiar en mí siempre que estuvo en vida y porque por él es que hoy me hago ingeniero.

A mi mamá, por su apoyo y ayuda en cada momento de realización de este trabajo.

A mis primas Ana y Amarilis, por su inmenso e incondicional apoyo, por estar siempre en todo momento y porque por ellas he logrado ser perseverante en mis estudios y vida en general.

A "tata" mi hermana, por su inmensa ayuda, por su preocupación, dedicación y por ser mi guía en la vida, a mi hermano Edua, por aguantar mis horas de estudio a mi lado a pesar de no entender nada, solo le bastaba hacerme compañía.

A "tuto" mi apreciada abuela, por creer siempre en mí y apoyarme en todas mis decisiones.

A mis primos Yilian, Lázaro, Marileydis, Eddito y Leylí, a mis tíos Lume, Ramiro, Pipo, Marisol y Fanny, a mi abuela Emma, a todos ellos, por su preocupación constante y ayuda durante mis años de universitario y vida en general.

A todos mis profesores, especialmente a Damián, por haber contribuido grandemente a mi formación como profesional durante estos 5 años.

A mis amigos Yasnelis, Yendry, Marta, Ivis, Yaniel, Javier Maryanis, José Carlos, Dainer e Iván, por estar siempre y por su ayuda incondicional y preocupación durante toda la carrera.

A Alejandro, Alexander, Alexis, Beto, Claudia Durán y Grettel, por apoyarme en todo momento y preocuparse por mi desempeño a diario.

A Argilagos, por su inmensa capacidad e inteligencia a disposición en todo momento que lo necesité.

A Odiel, por su ayuda y orientación durante el desarrollo de este trabajo.

A todos los profes del proyecto, Cesar, Yusliel, Damaris, Jorgito, Yunieky, Miguel y Nayla, por su gran ayuda en la tesis y por los consejos brindados siempre que fue necesario.

A mis compañeros de aula y CDI, Aylena, Zulema, Sandra, Dairelis, Jorge, Ricardo, Yoslandy, David, Eddy

A todos mis compañeros en general, Karelia, Yanisbel, Aristides, Yisel, Yannier, Yaima, Dianela, Lisbethy, Yudith, Irnel, Yusy a todos y otros que me queden por ahí, les agradezco por su preocupación y ayuda siempre que los necesité.

… Y a todos los que de una u otra forma han hecho posible la realización de la Obra de mi Vida… ¡¡¡MUCHAS GRACIAS!!!

Resumen

El rápido desarrollo de la ciencia y la tecnología, ha llevado a la sociedad a entrar en un nuevo milenio inmerso en lo que se ha dado llamar: “Era de la informatización”.

En la línea de Redes Sociales, perteneciente al Centro de Informatización Universitaria (CENIA) de la Universidad de las Ciencias Informáticas, se quiere llevar a cabo el proceso de construcción de una red social universitaria, en la cual queden reflejados todas las funcionalidades y servicios que serían de utilidad para la comunidad y que manifiesten de esta forma la informatización, comunicación social y el uso de archivos multimedia los cuales constituyen un peso sumamente importante en todo este proceso. El propósito de esta investigación se enfoca en proponer un sistema de archivos distribuidos que soporte la gran cantidad de información y datos que pueden ser gestionados en la red social, y de esta misma forma, proveer un servicio eficiente, garantizando a través de este sistema de ficheros, un acceso concurrente de todos los usuarios sin causar carga en el servidor. Mediante un estudio realizado acerca de varios sistemas de ficheros distribuido, se propone para el soporte de los servicios de la red social, el sistema de archivos distribuido *Hadoop*. A lo largo de todo el proceso de investigación, se reflejan algunos elementos de su diseño, arquitectura y se establecen todos los elementos de configuración e instalación del mismo. La escalabilidad, rendimiento, rapidez, robustez y fiabilidad de este sistema de ficheros, son los principales indicadores tomados como punto de partida para la validación del sistema y se ha podido verificar que es altamente viable para el soporte de grandes cantidades de datos y grandes configuraciones de clúster de máquinas.

De manera general se pretende que con la propuesta de este sistema de ficheros distribuido, se pueda concebir un funcionamiento eficiente en la universidad con el uso de la red social universitaria y que las prestaciones de rendimiento sean mayores.

Palabras claves: clúster, *hadoop*, sistema de archivos, sistema de archivos distribuido

Abstract

The fast development of science and technology, has led the society to enter in a new millennium immersed in what has been called "Computerization Age."

In the Social Networks line which forms part of the Academic Informatization Center (CENIA) at the Computer Sciences University, the process of building a University Social Network is carried out, where all the functionalities and services are reflected and they would be useful to the community. Thus, they might manifest the computerization, communication and the use of media files which constitute a very important weight in this process. The purpose in this research is focused on proposing a distributed file system that supports a wealth of information and data that can be managed through the network, and simultaneously, providing an efficient service ensuring a concurrent access to the file through this file system for all users without causing a server overload. By mean of a study of several distributed file systems the Hadoop Distributed File System (HDFS) has been proposed for the support of the social network services. Similarly, in the research is reflected some elements of its design, architecture and all the setup and installation process is specified. The scalability, performance, speed, robustness and reliability of this file system are the main indicators taken as a starting point for the system validation and the fact that it is highly feasible to support large amounts of data and large configurations clusters of machines is verified.

Generally it is intended that with the proposal of the distributed file system, an efficient operation may be conceived at the university with the use of the University Social Network , so the benefits of performance is becoming even greater more and more.

Keywords: *cluster, distributed file system, files system, hadoop*

Índice

Tabla de contenido

Índice de tablas

Índice de figuras

INTRODUCCIÓN

En el mundo actual, a lo largo de los años, el avance tecnológico ha ido evolucionando de manera progresiva y las Tecnologías de la Información y las Comunicaciones (TIC) han tomado las riendas del desarrollo en la socialización mundial a través de la tecnología. Con el surgimiento de Internet, las posibilidades de comunicación fueron aún mayores y el auge de la era digital ha revolucionado en la sociedad, de modo que hoy en día se hace imprescindible el uso de este recurso en el desarrollo social a nivel mundial. Las relaciones entre los individuos de la sociedad están determinadas por las relaciones sociales que se ejercen a través del trabajo, la comunicación y la actividad. Las redes computacionales han servido para satisfacer esta necesidad y han llegado a significar un recurso indispensable en la actual era digital en la que se está inmersa. De aquí surge la llamada “Red Social” la cual complementa y satisface muchas de las necesidades sociales expresadas por la humanidad y que se encargaría entonces de proporcionarle a usuarios y/o grupos de usuarios, una cercanía a disímiles servicios y fuentes personales de una manera totalmente dinámica.

Las redes sociales se caracterizan por tener cientos de usuarios asociados a ella, los cuales pueden en el mundo entero acceder de forma concurrente. Así mismo, el Centro de Informatización Universitaria (CENIA) de la Universidad de las Ciencias Informáticas (UCI), establece la línea de redes sociales en la cual se llevará a cabo la construcción de una Red Social Universitaria (RSU), donde toda la comunidad pueda acceder, crear y modificar perfiles, añadir contactos, publicar información e interactuar con el resto de la comunidad de forma dinámica, para lo cual se hace uso dentro de otras necesidades, de grandes volúmenes de almacenamiento, esencialmente cubiertas por datos, información personal y archivos multimedia. Para ello es necesario un sistema de ficheros capaz de soportar miles de usuarios accediendo simultáneamente a solicitar y usar diferentes servicios que brindará esta red social. Dichos servicios pueden ser comparados con los definidos en las redes sociales universalmente conocidas como Facebook, Twitter, MySpace, Picassa, Flickr, entre otras, las cuales almacenan grandes capacidades de información y archivos, utilizando un sistema de ficheros hacedero para el soporte requerido. Un sistema de archivos es: “un conjunto de datos abstractos, concretamente algoritmos y estructuras lógicas, utilizados para poder acceder a la información que se tiene en el disco y que son implementados para el almacenamiento, la organización jerárquica, la manipulación, el acceso, el direccionamiento y la recuperación de datos”(Díaz 2009). Cada uno de los sistemas operativos utiliza estas estructuras y

algoritmos de diferentes maneras independientemente del *hardware*. El desempeño del disco duro, la fiabilidad, seguridad, capacidad de expansión y la compatibilidad, estará en función de estas estructuras lógicas. Por tanto, dependiendo del tipo de sistemas de archivos que utilice el sistema operativo elegido, así será su eficacia, seguridad y eficiencia.

Para la red social de la comunidad universitaria de la Universidad de las Ciencias Informáticas, no se propicia un almacenamiento de grandes volúmenes de datos, que permita el acceso concurrente a la información personal de los usuarios de esta red y que pueda ser compartida entre el resto de los usuarios, delimitando de esta forma el intercambio social y la comunicación, además de provocar una sobrecarga en el servidor.

De acuerdo a lo anteriormente planteado se define como **problema a resolver**: ¿Cómo contribuir al almacenamiento de grandes capacidades de datos de los recursos gestionados en la red social de la Universidad de las Ciencias Informáticas garantizando el acceso concurrente a los mismos?

El referido problema se manifiesta y constata en el **objeto de estudio** de dicha investigación, el cual se enmarca en los sistemas de ficheros distribuidos. Se precisa como **campo de acción** los sistemas de ficheros distribuidos que soporten los requerimientos de una red social.
Dicha investigación establece como **objetivo general** proponer un sistema de ficheros distribuido que permita almacenar en una red de computadoras, la información que se maneja en la red social universitaria de la Universidad de las Ciencias Informáticas, mejorando el acceso de forma concurrente a los recursos gestionados en la misma.

El objetivo general de esta investigación es desglosado en los siguientes **objetivos específicos:**

- Inferir en el estudio de los sistemas de archivos distribuido.
- Definir cualidades y deficiencias de los sistemas de ficheros partiendo de indicadores como fiabilidad, estabilidad, rapidez, escalabilidad y rendimiento.
- Fundamentar la propuesta final del sistema de ficheros para el desarrollo de la red social de la universidad.
- Establecer pruebas de rendimiento y pruebas de carga para validar la selección del sistema de ficheros en aras de definir la efectividad de la propuesta de solución.

Para dar cumplimiento a los objetivos planteados han sido utilizados los **métodos científicos** de la investigación: **teórico y empírico**.

Los **métodos empíricos** de investigación estudian las características y relaciones esenciales del objeto que son accesibles directamente desde la percepción sensorial. Tienen como elemento principal la búsqueda sistemática del conocimiento a partir de los datos y las informaciones que proporcionan la percepción sensorial.

Dentro de los métodos de investigación empíricos se han empleado:

Entrevistas: para identificar el estado de opiniones de los especialistas del nodo central de la Universidad de las Ciencias Informáticas, sobre el desarrollo tecnológico en la misma, el uso de sistemas de ficheros, así como la posibilidad de probar un sistema de ficheros para la red social universitaria.

Los **métodos teóricos** de investigación se aplican durante el proceso de explicación, predicción, interpretación y comprensión de la esencia del objeto. Además, posibilitan la interpretación conceptual de los datos empíricos, revelan las relaciones esenciales del objeto de investigación que no son observados a simple vista, participan en la construcción del modelo y la hipótesis de la investigación.

Dentro de los métodos de investigación teóricos se emplearon:

Método Analítico-Sintético: para procesar la información en la elaboración de los fundamentos teóricos y las conclusiones de manera general.

Inductivo-Deductivo: es empleado para realizar un análisis de las necesidades de la universidad, de disponer de un sistema de ficheros distribuido para el soporte de grandes cantidades de información y de igual forma, para el acceso concurrente por parte de miles de usuarios interactuando con la red social universitaria.

Histórico-Lógico: se utilizó para el estudio de los antecedentes de las investigaciones relacionadas con el propósito de esta investigación.

Estructura del documento

El presente trabajo de diploma está estructurado de la siguiente forma: Introducción, Capítulo I, Capítulo II, Capítulo III, Conclusiones generales, Recomendaciones, Bibliografía y Anexos.

Capítulo 1: Fundamentos teóricos sobre los sistemas de ficheros: en este capítulo se analizan los conceptos asociados al objeto de estudio para el dominio de la propuesta de solución; se hace un estudio del arte sobre el uso de los sistemas de ficheros en el mundo computacional y su desarrollo evolutivo

hasta nuestros días. Finalmente se define el sistema de archivos que será propuesto para el soporte de los recursos de la red social de la universidad.

Capítulo 2: Sistema de Archivos Distribuido *Hadoop* (HDFS): se caracteriza y se define al sistema de ficheros distribuido seleccionado a través de elementos del diseño y la arquitectura, una vez realizado el análisis mediante los elementos comparativos que definen teóricamente la viabilidad de este sistema para la red social de la Universidad de las Ciencias Informáticas.

Capítulo 3: Resultados y validación del Sistema de Archivos Distribuido *Hadoop*: se realiza una evaluación técnica de la propuesta a través de pruebas al sistema de ficheros con el objetivo de validar los elementos teóricos previamente expuestos y verificar que la solución es idónea en cuanto a fiabilidad, escalabilidad, rendimiento, rapidez, entre otros.

CAPÍTULO 1: FUNDAMENTOS TEÓRICOS

1.1 Introducción del capítulo

En el presente capítulo se desarrolla un análisis de los elementos teóricos que van a servir como base a la problemática y objetivos planteados en este trabajo, caracterizando cada uno de ellos de forma tal que se logre comprender el entorno de la investigación y la teoría que la sustenta. Se describen diferentes sistemas de archivos necesarios a tener en cuenta para determinar cuál será aplicado para el soporte de los servicios de la red social de la UCI, así como un análisis de cada uno de ellos para identificar sus características propias en cuanto a funcionamiento y utilidad.

1.2 Conceptos asociados a los sistemas de ficheros distribuidos

Fichero

Un fichero es una abstracción muy importante en informática. Los ficheros sirven para almacenar datos de forma permanente y ofrecen un pequeño conjunto de primitivas muy potentes (abrir, leer, avanzar puntero, cerrar, etc.). Los ficheros se organizan formalmente en estructuras de árbol, donde los nodos intermedios son directorios capaces de agrupar otros ficheros(Mancha 2002).

Un fichero es un conjunto de información relacionada y organizada como una secuencia de bytes (espacio de direcciones lógico contiguo) que tiene un nombre y se clasifica según criterios del usuario en datos (alfanumérico, binario, etc.), programa (fichero ejecutable). Son dispositivos lógicos gestionados por el sistema operativo en el cual se pueden definir reglas de acceso para que sean compartidos (Silberschatz 2011).

De forma general se puede señalar que un fichero es un conjunto de información que se almacena de forma virtual para ser leído y accedido por medio de una computadora.

Sistema de ficheros

Es uno de los componentes más visibles del sistema operativo. Proporciona el mecanismo para almacenar, localizar y recuperar datos y programas del sistema operativo y de los usuarios. Es una colección de ficheros y una estructura de directorios donde se establece un espacio de nombres a través del cual se puede referenciar unívocamente un fichero. Provee abstracciones para gestionar los

dispositivos de almacenamiento secundario que no son volátiles, que se pueden compartir entre procesos, que se pueden organizar y que son independientes del dispositivo físico(Silberschatz 2011).
De esta forma, es apreciable ver que los sistemas de ficheros son uno de los principales componentes de un sistema operativo y de ellos se espera que sean rápidos y extremadamente fiables.

Por otra parte, el sistema de ficheros es la forma en que el sistema operativo organiza, gestiona y mantiene la jerarquía de ficheros en los dispositivos de almacenamiento, normalmente discos duros. Cada sistema operativo soporta diferentes sistemas de ficheros, para mantener la modularización del sistema operativo y proveer a las aplicaciones con una interfaz de programación (API) uniforme(Mancha 2002). Los diferentes sistemas operativos implementan una capa superior de abstracción denominada sistema de ficheros virtual (*VFS: Virtual File System)*. De esta forma y a consideración propia, en computación, un sistema de archivos es un método para el almacenamiento y organización de archivos de computadora y los datos que estos contienen, para hacer más fácil la tarea de encontrarlos y acceder a ellos.
Así mismo, los archivos son administrados por el sistema operativo como se muestra en la Figura 1. Su estructura, nombre, forma de acceso, uso, protección e implantación son temas fundamentales en el diseño de un sistema operativo. Aquella parte del sistema operativo que trabaja con los archivos se conoce, como un todo, como el sistema de archivos.

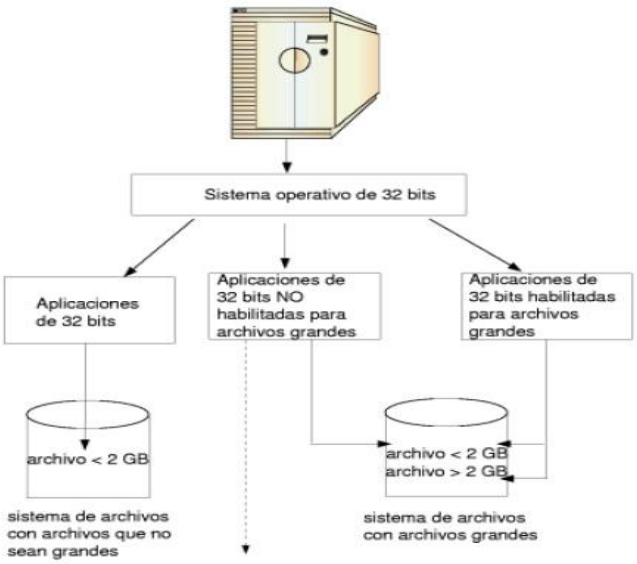


Figura 1: **Administración de archivos por el sistema operativo**.

Sistema de ficheros distribuido

Un sistema de ficheros distribuidos o sistema de archivos de red es un sistema de archivos de computadoras que sirve para compartir archivos, impresoras y otros recursos como un almacenamiento persistente en una red de computadoras. El primer sistema de este tipo fue desarrollado en la década de 1970, y en 1985 *Sun Microsystems* creó el sistema de archivos de red NFS *(Network File System),* el cual fue ampliamente utilizado como sistema de archivos distribuido(Stallings 2006).

La gestión de un sistema de ficheros distribuido se soporta mediante dos funciones a menudo bien diferenciadas: un servicio de nombres o directorios y un servicio de ficheros. Es fundamental proporcionar un rendimiento aceptable, por lo que hay que alcanzar un compromiso entre la disponibilidad local de la información para disminuir los costes de comunicación (*caching* y distribución) y la consistencia, que se refleja en la semántica que muestran los accesos compartidos(Lafuente 2010).

Un sistema de ficheros almacena ficheros en una o más computadoras llamadas servidores y las hacen accesibles a otras computadoras llamadas clientes(Braam 2003).

Un sistema de ficheros se caracteriza por un conjunto de propiedades generales:

- proporciona almacenamiento de información permanente;
- identifica los ficheros en un espacio de nombres (normalmente estructurado);
- es posible el acceso concurrente desde varios procesos;
- proporciona protección de accesos en sistemas multiusuario.

Archivado

Organiza como están estructurados los datos en un espacio de direcciones de almacenamiento, así como son representados a las aplicaciones y los usuarios(Ulf Troppens 2009).

- Existen dos tipos de archivados: sistema de ficheros y bases de datos.
- Un Sistema Operativo (SO) gestiona la secuencia de eventos de procesamiento, mientras los sistemas de ficheros gestionan los espacios de direcciones de almacenamiento: asignación de espacio y control de acceso.
- Proveen una interfaz amigable y estructuras más simples para acceder a los datos.

Ruta de acceso

Componentes que son recorridos (*hardware* y *software*) por peticiones de escritura y lectura(Ulf Troppens 2009).

Caching

Mecanismo designado para acortar las rutas de acceso para los elementos accedidos frecuentemente, para de esta forma mejorar el rendimiento(Jepsen 2003).

Control de acceso

Es el conjunto de técnicas que hacen cumplir las decisiones que encapsulan los permisos de acceso a los datos(Jepsen 2003). (Sistemas operativos, encriptación, subredes y zonas de red). Los objetivos básicos son:

- autenticación (verifica si es quien dice ser).
- autorización (permite realizar una determinada acción).
- privacidad (permite ver el contenido).

Clustering

Es una forma de "servicio de agregación" para mejorar la escalabilidad, la tolerancia a fallos y facilidad de gestión(Jepsen 2003).

En el contexto de sistemas de ficheros, esta técnica es conocida como *cluster file system* o *distributed file system*. La gestión de volúmenes lógicos tienen las mismas propiedades.

1.3 Propiedades de los sistemas de ficheros distribuidos

Un sistema de ficheros se caracteriza por un conjunto de propiedades generales:

- proporciona almacenamiento de información permanente;
- identifica los ficheros en un espacio de nombres (normalmente estructurado);
- es posible el acceso concurrente desde varios procesos;
- en sistemas multiusuario proporciona protección de accesos.

Un sistema de ficheros distribuido posee las siguientes propiedades:

- Transparencia en la identificación: espacio de nombres único e independiente del cliente.
- Transparencia en la ubicación: para permitir la movilidad del fichero de una ubicación a otra, se requiere una correspondencia dinámica nombre-ubicación.
- Escalabilidad: espacios de nombres estructurados, y replicación (*caching*) para evitar cuellos de botella.
- Robustez ante fallos: el servidor no debe verse afectado por los fallos de los clientes, lo que incumbe a la gestión del estado de los clientes en el servidor. Por otra parte, la interfaz ofrecida a los clientes debe proporcionar en lo posible operaciones idempotentes, que garanticen la corrección ante invocaciones repetidas (por sospecha de error) al servidor.
- Disponibilidad y tolerancia a fallos: implican alguna forma de replicación. Un aspecto de la disponibilidad es permitir el funcionamiento en modo desconectado, que requiere *caching* de ficheros enteros.
- Consistencia: el objetivo es mantener en lo posible, la semántica de los sistemas centralizados, por ejemplo preservar la semántica UNIX (*kernel* del sistema operativo Linux) en presencia de *caching* u otras formas de replicación.
- Seguridad: la necesidad de autenticación remota implica nuevos modelos de protección, basados en credenciales en lugar de listas de accesos.

1.4 Caracterización del uso de los ficheros

El diseño de un sistema de ficheros distribuido que proporcione un buen nivel de rendimiento deberá basarse en las características de uso de los ficheros por las aplicaciones. Aunque no es fácil generalizar, sí es posible determinar una serie de patrones de comportamiento(George Coulouris 1994).

- La mayoría de los ficheros son de pequeño tamaño por lo que implica que el fichero puede ser la unidad de recuperación.
- La escritura es poco frecuente, de esta forma alienta el *caching* y la replicación.
- El ratio de búsqueda/uso suele ser bajo, por lo que también favorece el *caching*.
- El acceso suele ser secuencial y existe un alto grado de localidad, promoviendo el *buffering* para proporcionar anticipación en los accesos.

- La mayoría de los ficheros tienen una vida muy corta (ficheros temporales), por lo que es necesario tender a gestionarlos localmente.
- Existen clases de ficheros, con propiedades diferenciadas (por ejemplo ejecutables, que rara vez se modifican).

1.5 Clasificación de los sistemas de ficheros UNIX

Los sistemas de ficheros soportados por Linux se clasifican en tres categorías(Mancha 2002):

1. **basados en disco:** discos duros, disquetes, CD-ROM, etc. (Estos sistemas son ext2, ext3, ReiserFS, XFS, JFS e ISO9660)
2. **sistemas remotos (de red):** NFS, Coda y Samba.
3. **sistemas especiales:** procfs, ramfs y devfs.

El modelo general de ficheros puede ser interpretado como orientado a objetos, donde los objetos son construcciones de *software* (estructura de datos y funciones y métodos asociados.

1.6 ¿Dónde encajan los sistemas de ficheros dentro del sistema operativo?

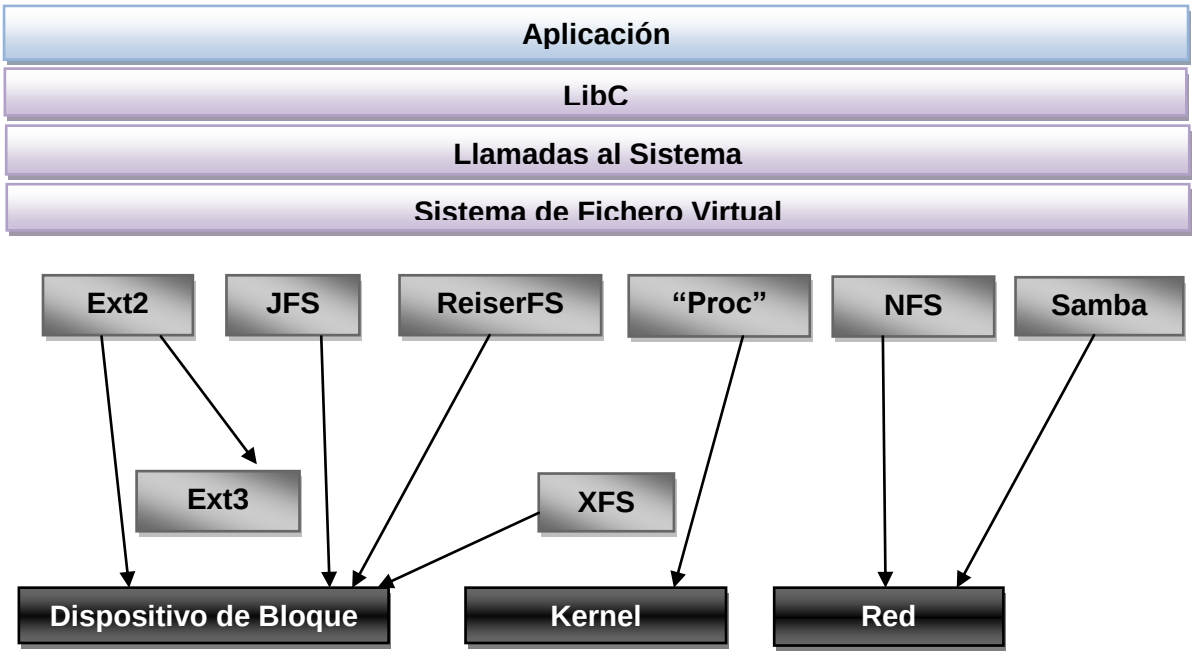


Figura 2: **Organización de los sistemas de archivos en un sistema operativo.**

1.7 Sistemas de ficheros en Linux con *journaling*

Al trabajar con un ordenador, para mejorar el rendimiento de las operaciones de entrada y salida (E/S), los datos del disco son temporalmente almacenados en la memoria RAM (Linux utiliza para ello dos mecanismos: el *page-cache y el buffer-cache*). Los problemas surgen si hay un corte de suministro eléctrico antes de que los datos modificados en la memoria (*dirty buffers*) sean grabados nuevamente al disco. Se generaría una inconsistencia en el estado global del sistema de ficheros. Por ejemplo, un nuevo fichero que todavía no fue “creado'' en el disco u otros que hayan sido borrados pero su i-nodo y bloques de datos todavía permanece como “activos'' en el disco.

Un sistema con *journaling* es un sistema de ficheros tolerante a fallos, en el cual la integridad de los datos está asegurada pues las modificaciones de la meta-información de los ficheros son primero grabadas en un registro cronológico (*log o journal*, que simplemente es una lista de transacciones) antes que los bloques originales sean modificados. En el caso de un fallo del sistema, un sistema con *journaling* asegura que la consistencia del sistema de ficheros sea recuperada. El método más común es el de grabar previamente cualquier modificación de la meta-información en un área especial del disco, el sistema realmente grabará los datos una vez que la actualización de los registros haya sido completada(Mancha 2002).

La demanda de sistemas de ficheros que soportan *terabytes* de datos, miles de ficheros por directorios y compatibilidad con arquitecturas de 64 *bits* ha hecho que en los últimos años haya crecido el interés de la disponibilidad de sistemas con *journaling* en Linux, ya que utilizando estos sistemas de ficheros se simplifican los reinicios de la máquina, se reducen las fragmentaciones y se aceleran las operaciones de entrada/salida. Los primeros sistemas de ficheros con *journaling* fueron creados a mediados de los años 80 e incluyen a Veritas (VxFS), *Tolerant* y JFS (*Journal File System*) de IBM *(International Business Machines)*. Linux tiene ahora disponible cuatro sistemas de ficheros transaccionales: ReiserFS de *Namesys*, XFS de *Silicon Graphic* (SGI), JFS de IBM y el ext3 que fue desarrollado por *Stephen Tweedie*, co-desarrollador del Ext2. Cada uno de ellos tiene características específicas que le diferencian del resto(Mancha 2002).

1.7.1 Sistema de ficheros Ext3

El sistema de ficheros Ext3 es una extensión con *journaling* del sistema de ficheros Ext2. Como ya se ha visto, con el *journaling* se obtiene una enorme reducción en el tiempo necesario para recuperar un sistema de ficheros después de una caída, y es por tanto muy recomendable en entornos donde la alta disponibilidad es muy necesaria, no sólo para reducir el tiempo de recuperación de máquinas independiente, sino también para permitir que un sistema de ficheros de una máquina ante fallo, sea recuperado en otra máquina cuando se tiene un clúster con algún disco compartido. Además se posibilita que el sistema de ficheros caído de una máquina (por ejemplo un servidor) esté disponible cuanto antes para el resto de máquinas a través de la red (NFS, Samba, FTP, HTTP, etc.)(Mancha 2002).

Este sistema es una capa del sistema de archivo Ext2 que sí mantiene un archivo *journal* de la actividad del disco de modo que la recuperación de un apagado imprevisto es más rápida que en el Ext2 solamente. Pero, debido a que está muy atado al Ext2, sufre algunas limitaciones y además no explota todo el potencial de los sistemas de archivos puros con *journaling*, por ejemplo, es aún basado en bloques y usa búsqueda secuencial de archivos en nombres de directorios(Granada 2002).

Journal Ext3 mantiene orden de consistencia en ambos, los datos y los metadatos. A diferencia del sistema de archivos anterior, la consistencia es asegurada para el contenido de los archivos. El nivel de *journaling* puede ser controlado con opciones de montaje.

1.7.2 Sistema de archivos ReiserFS

Reiser es basado en un árbol balanceado rápido (*B-Tree*) para organizar los objetos del sistema de ficheros. Esto proporciona una búsqueda rápida de directorios y rápidas operaciones de borrado. Otra característica de rendimiento de este sistema, es que incluye el soporte de archivos esparcidos y asignación de nodo en disco dinámico(Ray Bryant 2002). Los objetos de los sistemas de ficheros son las estructuras usadas para mantener la información del fichero: tiempo de acceso, permiso a archivos, etc. En otras palabras, es la información contenida dentro de un i-nodo, directorios y datos de archivos. ReiserFS llama a esos objetos, objetos de datos, objetos de directorios y objetos directos e indirectos. ReiserFS solo proporciona *journaling* de metadatos. En caso de un reinicio no planeado del sistema, los datos en bloques que estaban siendo usados en el momento del fallo, podrían haber sido corrompidos. Así ReiserFS no garantiza que el contenido de los archivos no esté corrompido(Granada 2002).

Los nodos no formateados son bloques lógicos sin ningún formato dado, usados para almacenar datos de archivos y el objeto directo consiste en ese propio dato de archivo. También esos objetos son de tamaño variable y almacenados dentro de las hojas del nodo del árbol, a veces con otros en caso que exista suficiente espacio en el nodo(Granada 2002).

La cola de paquetes es una característica especial de ReiserFS. Las colas son archivos más pequeños que un bloque lógico o la pérdida de porción de archivo que no llena un bloque completo. Para salvar espacio en el disco, ReiserFS usa la cola de paquetes para ocupar las colas en un espacio tan pequeño como sea posible. Generalmente esto permite a ReiserFS ocupar acerca del 5% del equivalente al sistema de archivos Ext2. Sin embargo, esta opción tiene un alto costo de rendimiento por lo que la mayoría de los sistemas son corridos con esta opción deshabilitada(Ray Bryant 2002).

1.7.3 Sistema de archivo XFS

El 1ro de mayo de 2001, SGI (*Silicon Graphic*) hizo disponible la liberación de la versión 1.0 del sistema de archivos para Linux XFS. Es reconocido por su soporte en grandes ficheros de disco y también por su alta ejecución de entrada/salida (probado con 7gb/sec). XFS fue desarrollado por el sistema operativo IRIX 5.3 *SGI Unix*. El objetivo del sistema de ficheros es soportar grandes capacidades de archivos y altas ejecuciones de entrada/salida para grabación y ejecución de videos en tiempo real.

Para incrementar la escalabilidad del sistema de ficheros, XFS usa extensivamente el árbol balanceado (*B-Tree*). Este sistema de archivos usa una extensión basada en asignación de espacio y tiene características como asignación retrasada, pre-asignación de espacio y es encaminado a grandes longitudes para distribuir archivos usando las extensiones más grandes posibles. De igual forma usa enormes descriptores de extensiones en el mapa de extensiones de archivos, donde cada descriptor puede describir hasta 2 millones de bloques de sistemas de ficheros. Los descriptores eliminan la sobrecarga del CPU (*Central Processing Unit*) por sus siglas en inglés, de las entradas de escaneo en el mapa de extensiones para determinar si los bloques en el archivo están contiguos. Esto puede simplemente leer la longitud de la extensión más bien que mirar a cada entrada para ver si es contigua con la previamente entrada(Granada 2002).

Según plantea(Ray Bryant 2002) XFS es un sistema de archivos *journaling* que soporta metadatos *journaling*. También usa grupos de asignaciones y extensiones basadas en asignación para mejorar la

localización de los datos en el disco. Esto resulta en una ejecución mejorada, particularmente para grandes transferencias secuenciales.

XFS permite bloques de tamaños variables, de 512 *bytes* a 64 *kilobytes* en las bases de un sistema por archivo. Cambiando el tamaño de bloque del sistema de archivos, puede variar la fragmentación. Los sistemas de archivos con grandes números de pequeños archivos, típicamente usan tamaños de bloques más pequeños para evitar gastar espacio a través de la fragmentación externa. Los sistemas de archivos con grandes archivos tienden a hacer una alternativa contraria y usan grandes tamaños de bloques para reducir la fragmentación externa del sistema de archivos y de sus extensiones de archivos(Granada 2002).

XFS soporta las Listas de Control de Acceso o *Access Control Lists* (ACL´s) (integradas con el servidor Samba) y cuotas transaccionales.

1.7.4 Sistema de ficheros JFS

IBM introdujo su sistema de ficheros UNIX como el Sistema de Fichero *Journaled* (JFS) con la liberación inicial de AIX versión 3.1. Tiene ahora introducido un segundo sistema de archivos que es para correrlo en sistemas AIX llamado “Sistema de Archivo *Journaled* Mejorado” (JFS2) el cual está disponible en sistemas AIX en su versión 5.0 y otras posteriores a esta(Granada 2002).

Las características técnicas de JFS se centran en la inclusión de una asignación de almacenamiento basado en extensiones, tamaño de bloque variable (aunque solo 4096 *bytes* de bloques son soportados actualmente por Linux), la asignación de i-nodos en discos dinámicos y el soporte de archivos densos y esparcidos. JFS es un sistema de archivos *journaling* que soporta entrada de metadatos. No es actualmente soportado en sistemas donde el número de páginas excede el tamaño de bloque del sistema de ficheros(Ray Bryant 2002).

Los *logs* JFS son mantenidos en cada sistema de archivos y usados para archivar información sobre operaciones en metadatos. El *log* tiene un formato que solo es modificado por la utilidad de creación del sistema de archivos. La semántica de entrada de JFS es tal que, cuando una operación del sistema de archivos involucra cambios de metadatos, devuelve un código de respuesta exitoso, el efecto de la operación ya ha sido entregado al sistema de archivos y será visto incluso, si el sistema falla después de

la operación. JFS soporta tamaños de bloque de 512, 1024, 2048 y 4096 *bytes* en un sistema de bases por archivo.

1.8 Descripciones de los sistemas de ficheros distribuidos

1.8.1 Sistema de archivos NFS (*Network File System*)

Introducido por *Sun Microsystems* en 1985 y desarrollado originalmente para UNIX. Se concibió como sistema abierto, lo que le ha permitido ser adoptado por todas las familias UNIX y por otros sistemas operativos: VMS (*Virtual Memory System*), *Windows*, convirtiéndose en un estándar de factor en LANs (*Local Area Network*). NFS ha evolucionado mucho y la versión 4 poco tiene que ver con las anteriores, ya que incluye la posibilidad de implementación en WAN (*Wide Area Network*)(Lafuente 2010).

NFS permite compartir datos entre varios ordenadores de una forma sencilla. Es un sistema de archivos distribuido para un entorno de red de área local. Posibilita que distintos sistemas conectados a una misma red accedan a ficheros remotos como si se tratara de locales.
Por ejemplo: un usuario validado en una red no necesitará hacer *login* a un ordenador específico, vía NFS, accederá a su directorio personal (que se llamará exportado) en la máquina en la que esté trabajando.
El NFS, tiene como objetivo dentro de estos procesos, soportar un sistema heterogéneo donde los clientes y servidores podrían ejecutar distintos sistemas operativos en *hardware* diverso; por ello es esencial que la interfaz entre los clientes y los servidores esté bien definida.

- NFS logra este objetivo definiendo dos protocolos **“cliente - servidor”.**

Un **“protocolo”** es un conjunto de: solicitudes que envían los clientes a los servidores y un conjunto de respuestas que envían los servidores de regreso a los clientes.

Los servidores exportan directorios. Para hacer exportable un directorio se incluye el camino en un determinado fichero de configuración. Los clientes montan los directorios exportados, y estos se ven en el cliente completamente integrados en el sistema de ficheros. El montaje se ejecuta en el *booting* del sistema operativo, o por demanda cuando se abre un fichero mediante un servicio adicional de NFS. Las operaciones sobre ficheros y las peticiones de montaje son atendidas por sendos procesos *daemon* en el servidor (*nfsd* y *mountd r*espectivamente).

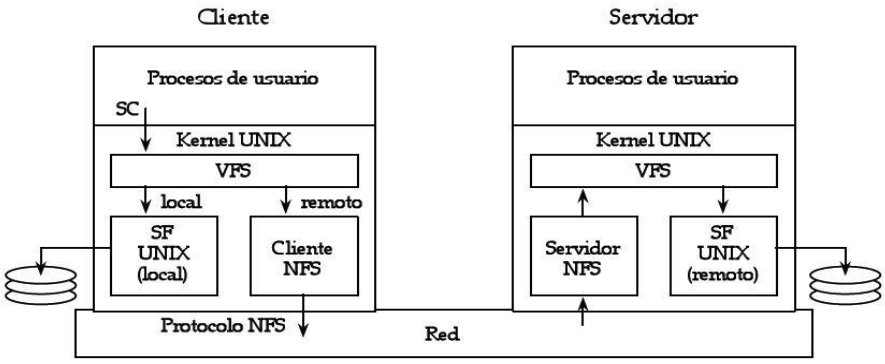


Figura 3: **Estructura del servicio NFS**.

El sistema NFS tiene algunas limitaciones como por ejemplo: el mantenimiento de la consistencia UNIX resulta problemático. La disminución del período de validación para mejorar la consistencia, produce una sobrecarga por la gran cantidad de operaciones *getattr* que se realizan. En principio, el montaje de sistemas de ficheros remotos no era transparente (hay que identificar al servidor). El *automounter* (utilidad que permite el montaje dinámico de sistemas de ficheros por demanda), mejora este aspecto. Debido a la falta de estado, bloquear el acceso a ficheros remotos requiere un mecanismo de exclusión mutua e independiente. En los servidores orientados a UNIX se utiliza un servidor específico (*lockd),* el cual no está diseñado para soportar replicación de servidores. Para incrementar la disponibilidad, las partes del sistema de ficheros que tengan que soportar una tasa muy alta de accesos, pueden replicarse en un conjunto de servidores siempre que sean para lectura.(Melamed 1987)

1.8.2 Sistema de archivos AFS (*Andrew File System*)

Andrew es el nombre de una familia de sistemas de ficheros distribuido para UNIX, desarrollados en la Universidad de Carnegie Mellon a partir de 1983. Los componentes de la familia son:

- AFS-1 (1983). Prototipo no optimizado.
- Coda (1987). Proporciona funcionamiento en modo desconectado.

AFS puede definirse en general como un sistema de ficheros distribuido sin estado, que proporciona semántica de sesión(Lafuente 2010).

La arquitectura de AFS consta de dos componentes, uno en el servidor y otro en el cliente:

- Vice: código de los servidores. Desde el punto de vista del cliente, Vice es un conjunto de servidores de ficheros interconectados en red.
- Venus: código cliente que se ejecuta sobre el sistema operativo en los nodos conectados a Vice.

Los ficheros de Vice se ven como integrados en el sistema local de cada puesto cliente. Soporta replicación de subconjuntos del sistema de ficheros (*volúmenes*) de actualización poco frecuente (AFS-2). Esta técnica también se usa para *back-ups* (mediante copias de sólo lectura de un volumen).

La seguridad está dada en definición de dominios de acceso como UNIX. Los derechos de acceso se definen de modo compatible con UNIX. Proporciona autenticación por medio de un servidor de autenticación que expide *token* (fichas) ante la presentación de la clave del usuario en el *login* para acceder al sistema de ficheros durante un plazo preestablecido (típicamente 24 horas)(Lafuente 2010).

Disponibilidad: Coda

Coda es una versión de AFS para proporcionar disponibilidad en entornos sujetos a fallos, tanto en la red como en los servidores, por lo que resulta adecuado para dispositivos móviles (sujetos a desconexiones frecuentes) y en general en sistemas replicados que requieran tolerancia a fallos. Gestiona réplicas de volúmenes siguiendo una estrategia optimista.

Coda es un sistema de archivos distribuido avanzado y desarrollado desde 1987. Cuenta con bastantes características muy deseables para un sistema distribuido (especialmente para un clúster). El sistema de archivos distribuido CODA es un estado del arte experimental de los sistemas de ficheros desarrollado en el grupo de *M. Satyanarayanan at Carnegie Mellon University*(Braam 2003). Si CODA está funcionando sobre una PC cliente, la cual se tomaría que fuera en una estación de trabajo de Linux con un montaje manual, mostraría que un sistema de ficheros de tipo “CODA” está montado bajo CODA. Un cliente se conecta a “CODA” y no a un servidor individual. Esta es una gran diferencia de montar el sistema de archivos NFS el cual está hecho para servidores(Braam 2003).

1.8.3 Sistema de archivos LUSTRE

Es un sistema de archivos distribuido, creado por *Cluster Files System* y luego adquirido por *Sun Microsystems*. Es de gran escalabilidad, independiente del *hardware*, capaz de usar una gran variedad de redes, es un sistema robusto, usado por los clúster más potentes del mundo.

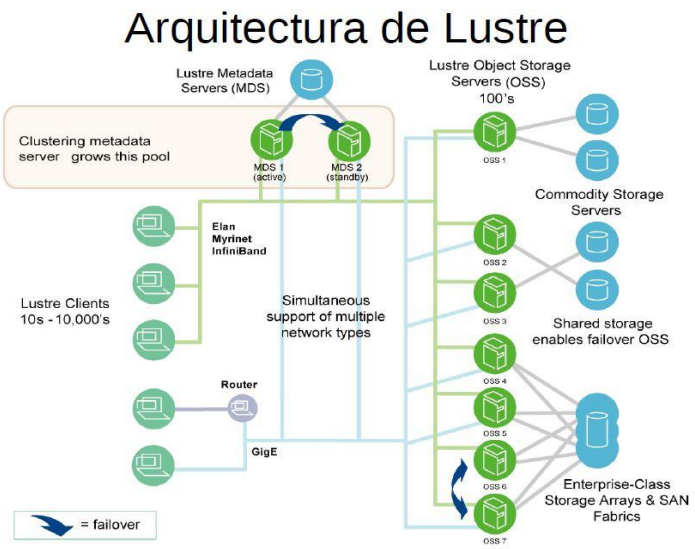


Figura 4: **Arquitectura LUSTRE** (Barton 2008)**.**

El *Metadata Server* (MDS) administra los metadatos asociados a cada archivo (ubicación, permisos, jerarquía, etc.). Todos los metadatos son guardados en MDTs (*Metadata Target*) que son unidades de almacenamiento local o compartidos. Las peticiones de archivos, listados de directorios, ubicación de archivos son procesados primero por el MDS(Barton 2008).

El *Object Server Storage* (OSS) atiende las peticiones de acceso a archivos (o partes de archivos) a los clientes. No guarda información de propietarios, jerarquía etc. El almacenamiento lo hace en los *Object Storage Target* que no es más que un medio de almacenamiento local o compartido.

Lustre puede correr sobre distintos tipos de redes e incluso combinaciones de estas (Elan, Myrinet, Infiniband, Ethernet). Para tener esta compatibilidad y otras características, Lustre usa una capa de red llamada LNET (*Lustre Networking*)(An Oak Ridge National Laboratory 2008).

El *File Over* está dado por la capacidad del sistema para tolerar caídas de servidores, recuperándose de manera transparente. Esta capacidad también permite realizar actualizaciones de equipos. Otras características que presenta este sistema de archivos son:

Interoperabilidad: Lustre puede ser utilizado en diversas plataformas (IA-32, IA-64, x86_64, PPC). También es compatible con versiones anteriores y posteriores a la utilizada y da la ventaja de escalar OSS's.

Listas de Control de Acceso (ACL´s): sigue el modelo de seguridad de UNIX, implementando características adicionales ACL POSIX y conexión desde puertos específicos. Las ACL's son atributos extra en los archivos.

Las cuotas: Lustre trabaja con restricciones a usuarios y grupos en cuanto a espacio en disco o ancho de banda.

Adición de OSS: la capacidad del sistema de archivos Lustre, de añadir OSS's sin interrumpir las operaciones del clúster.

***Striping* controlado:** el sistema de archivos tiene una configuración por defecto que determina cómo los archivos son distribuidos entre los OSS. Pero a un directorio se le pueden agregar atributos para modificar el tamaño del *stripe* a conveniencia, incluso las librerías pueden modificar el tamaño de *stripe*.

***Snapshot*:** es como "una foto instantánea" del sistema de archivos tomada en cierto momento para respaldo, permitiendo al usuario seguir trabajando sobre el sistema de archivos de manera transparente.

1.8.4 Sistema de archivos GFS (*Google File System*)

El sistema de archivos de *Google*, GFS (siglas de «*Google File System*»), es un sistema de almacenamiento basado en las necesidades de *Google*, diseñado por Sanjay Ghemawat, Howard Gobioff y Shun-Tak Leung y presentado por primera vez en *«19th ACM Symposium on Operating Systems Principles», Lake George, Nueva York*, octubre de 2003. Al no ser un sistema de archivos de uso generalista, GFS ha sido diseñado teniendo en cuenta las siguientes premisas: que un componente falle es la norma no la excepción, los archivos son enormes (archivos de muchos GB son comunes), es muy común que un archivo cambie porque se le añaden datos pero es muy raro que se sobrescriban los datos existentes, el diseño de las aplicaciones y de la API del sistema de archivos proporciona un beneficio global(Calvo 2004).

Como otros grandes sistemas de archivos tales como AFS(John Howard 2001), GFS proporciona una localización independiente de espacios de nombres, el cual habilita los datos para ser movidos transparentemente para cargar el balance de carga o la tolerancia de fallo. A diferencia de AFS, GFS despliega los datos del fichero sobre los servidores de almacenamiento en una manera más superficial que XFS(Thomas Anderson 1995) y *Swift*(Luis-Felipe Cabrera 2000) para entregar rendimiento agregado y tolerancia de fallo incrementada. Como los discos son relativamente baratos y la replicación es más simple que sofisticada (David A. Patterson 1988), GFS actualmente usa replicación para redundancia y

por lo tanto consume más rango de almacenamiento que XFS y *Swift*. Al contrario de los sistemas como AFS, XFS, *Frangipani*(Chandramohan A. Thekkath 1997) e *Intermezzo(Jacob Gorm Hansen 2001)* GFS no proporciona ningún *caching* debajo de la interfaz del sistema de ficheros.
Un clúster GFS consiste en un máster simple y múltiples *chunkservers* (trozos de servidores) y es accedido por múltiples clientes. Cada uno de estos es un producto de una máquina de Linux corriendo un proceso de servidor a nivel de usuario(Sanjay Ghemawat 2001).

¿Qué es un *chunk*?
Es análogo a los bloques, excepto que es más grande. Soporta un tamaño de 64MB, guardado en *chunkservers*(Sanjay Ghemawat 2001).

Existen muchos sistemas de archivos distribuido (AFS, XFS, *Swift*, *Intermezzo*, NASD, etc.) usando una variedad de técnicas (*peer-to-peer*, RAID, entre otras).
GFS difiere de estos en varios aspectos los cuales son importantes tener en cuenta. GFS es diseñado para clases restringidas de aplicación, modelo de consistencia abierto, uso de producto de *hardware* poco fiable, enfoque a tolerancia de fallos: aquí este usa esquemas simples *"fool-proof"* como centralización, replicación de datos y rápido proceso de recuperación en lugar de algoritmos complejos(Zaharia 2007).

1.8.5 Sistema de ficheros *Hadoop* (*Hadoop File System*)

Google trabaja sobre un sistema de ficheros diseñado para satisfacer sus propias necesidades, el cual es llamado *Google File System* (GFS) y es altamente optimizado para su propósito. A partir de la creación de este sistema de ficheros *Doug Cutting* y *Yahoo* llevaron a cabo el proceso de ingeniería inversa sobre el GFS y el sistema resultante fue el *Hadoop Distributed File System* (HDFS). Este sistema de ficheros es bajo código abierto y altamente tolerante a fallos, altamente soportable a acceso de entrada y salida al sistema, adecuado para aplicaciones con grandes cantidades de datos configurables, gran acceso de entrada y salida a datos del sistema, puede ser construido con productos de *hardware* de bajo costo(J. Dean 2008).
HDFS fue originalmente diseñado para ser un sistema de ficheros para aplicaciones de *MapReduce*, que son inherentemente un sistema de lotes, donde la escalabilidad y el proceso de entrada y salida son los más críticos. Se han visto las ventajas de usar HDFS y se pueden apreciar en su escalabilidad lineal y resultados de tolerancia ante fallos en enormes ahorros de costos en una empresa. El uso nuevo, más

actual y online de HDFS provee nuevos requerimientos y cambios en el núcleo como un propósito general de un sistema de ficheros(Dhruba Borthakur 2011).

Para gestionar los recursos de almacenamiento en todo el clúster, *Hadoop* utiliza un sistema distribuido a nivel de usuario del sistema de archivos. Este sistema de archivos - HDFS - está escrito en Java y diseñado para la portabilidad a través de *hardware* heterogéneo y plataformas de *software*. En este trabajo se analiza el desempeño de las cuestiones de HDFS y se analizan varios elementos de su rendimiento.

- en la arquitectura *hadoop* existen cuellos de botella en la aplicación *Hadoop*, que resultan del uso ineficiente de HDFS debido a los retrasos en la programación de nuevas tareas *MapReduce*.
- las limitaciones de la portabilidad previenen la implementación en Java de las características de explotación en la plataforma nativa.
- HDFS implícitamente hace suposiciones de portabilidad acerca de cómo la plataforma nativa gestiona los recursos de almacenamiento, incluso aunque los sistemas de archivos nativos y la programación de E / S varían ampliamente en diseño y comportamiento(Jeffrey Shafer 2010).

1.9 Conclusiones del capítulo

Durante este capítulo se llevó a cabo el estudio de algunos conceptos necesarios a tener en cuenta para la realización de la investigación en general, comprendiéndose de esta forma la idea fundamental a tener en cuenta para el desarrollo del trabajo. Se realizó un análisis sobre los sistemas de archivos existentes a través de elementos comparativos entre ellos, lo cual permitió determinar de acuerdo a indicadores cualitativos y cuantitativos, cuál de ellos sería capaz de darle solución a la problemática planteada en la investigación. Algunas características de cada sistema de ficheros fueron argumentadas, de modo que sirvió para definir las cualidades técnicas de cada uno de ellos y finalmente se determinó que: el sistema de ficheros viable para el soporte de los recursos y servicios de la red social de la Universidad de las Ciencias Informáticas, es el sistema de archivos *Hadoop* (HDFS), tomándose como punto de partida el análisis realizado sobre este y mediante los resultados obtenidos a través de los elementos comparativos mencionados anteriormente. De esta forma el sistema de ficheros distribuido *Hadoop*, constituye en esta investigación, la propuesta de solución para la red social, en el cual se profundizará aún más en el próximo capítulo.

CAPÍTULO 2: SISTEMA DE ARCHIVOS DISTRIBUIDO *HADOOP* (HDFS)

2.1 Introducción del capítulo

Cuando en un conjunto de datos crece la capacidad de almacenamiento de una sola máquina física, se hace necesaria la partición a través de un número de máquinas separadas. Los sistemas de ficheros que gestionan el almacenamiento a través de una red de máquinas, son llamados sistemas de archivos distribuido(White 2009). Uno de los mayores retos es hacer que el sistema de archivos tolere el fracaso del nodo sin sufrir pérdida de datos. HDFS es el sistema de archivos propuesto y es el elemento fundamental de este capítulo, el cual en realidad tiene una abstracción del sistema de archivos de propósito general, por lo que se analizará a lo largo de este capítulo su diseño, arquitectura y elementos de su configuración e instalación.

El Sistema de Archivos Distribuido *Hadoop* (HDFS) fue creado por *Doug Cutting*, el creador de *Apache Lucene*, la biblioteca de búsqueda de texto ampliamente utilizada. *Hadoop* tiene sus orígenes en *Apache Nutch*, un motor de búsqueda *web* de código abierto, en sí una parte del proyecto Lucene(White 2009). Es un sistema de archivos distribuido diseñado para ejecutarse sobre *hardware* de bajo costo. Tiene muchas similitudes con sistemas de archivos existentes, sin embargo, las diferencias con respecto a otros sistemas de archivos son significantes. Proporciona un alto acceso de entrada y salida a los datos de aplicación y es adecuado para aplicaciones que tienen grandes configuraciones de datos. Este sistema de archivos además fue construido como una infraestructura para el proyecto de motor de búsqueda *web* de *Apache Nutch(BORTHAKUR 2007).*

Antecedentes

Hadoop(Tien Duc Dinh 2009) es un *framework* de código abierto que implementa el modelo *MapReduce* de programación paralela(GHEMAWAT 2004).

Hadoop está compuesto por un motor *MapReduce* y un sistema de ficheros de nivel de usuario que gestiona los recursos de almacenamiento en todo el clúster. Para un transporte a través de una variedad de plataformas - Linux, FreeBSD, Mac OS / X, Solaris y *Windows* - los dos componentes se escriben en Java, y sólo requieren de un *hardware* de bajo costo(Jeffrey Shafer 2010).

Apache Hadoop es un *framework* de código abierto hecho en java para ejecutar aplicaciones sobre grandes clúster. *Hadoop* es un proyecto de *Apache* de alto nivel. Se basa en la colaboración de una extensa comunidad activa de contribuyentes alrededor del mundo para su éxito(Julio 2009).

2.2 Motor *MapReduce*

MapReduce es un modelo de programación introducido por *Google* para brindar soporte al procesamiento en paralelo. Es utilizado para escribir aplicaciones que procesen rápidamente grandes cantidades de datos en paralelo sobre grandes grupos de computadoras. Se especifica una función *map* que procesa un par clave/valor para generar una colección de pares clave/valor, y una función *reduce* que agrupa todos los valores intermedios asociados con la misma clave intermedia(ÁNGEL MERCHAN). *MapReduce* es un modelo igualmente empleado para el procesamiento de datos. En *Hadoop*, *MapReduce* puede ejecutar programas escritos en varios lenguajes. Los programas de *MapReduce* son inherentemente paralelos, poniendo a gran escala de análisis de datos en las manos de cualquier persona con suficientes máquinas a su disposición.(White 2009).

Por otra parte, *Tom White* plantea su criterio al respecto. *MapReduce* trabaja llevar a cabo el proceso en dos fases: la fase de mapa y la fase de reducción. Cada fase tiene pares clave-valor como entrada y salida, los tipos de los cuales pueden ser elegidos por el programador. El programador también especifica dos funciones: la función de *map* y la función de *reduce*(White 2009).

De manera similar, la función *reduce* puede funcionar independientemente en cada par clave-valor, exponiendo también paralelismo significativo. En *Hadoop*, un servicio centralizado *jobtracker* es responsable para dividir los datos de entrada en trozos, para su procesamiento por mapa independiente y reducir las tareas, reducir la programación de cada tarea en un nodo de clúster para su ejecución, y la recuperación de fallas por volver a ejecutar las tareas. En cada nodo, un servicio *tasktracker* ejecuta tareas *MapReduce* y periódicamente contacta el j*obtracker* para reportar la completitud de tareas y solicitar nuevas tareas. De forma predeterminada, cuando una tarea nueva es recibida, una nueva instancia de JVM (*Java Virtual Machine*) o Máquina Virtual de Java, será generada para ejecutarlo(Jeffrey Shafer 2010). La entrada a la fase *map* son los datos NCDC. Se elige un formato de entrada de texto el cual da a cada línea en el conjunto de datos como un valor de texto(White 2009).

La función *map* es simple, es sólo una fase de preparación de datos donde la creación de los estos es de tal manera que la función *reduce* puede hacer su trabajo, de acuerdo a la necesidad que establezca en cada caso. La función de *map* es también un buen lugar para colocar los registros malos: aquí se filtran las variables que falten, se sospechan, o erróneas indistintamente.

2.2.1 Flujo de datos

Un trabajo *MapReduce* o *MapReduce Job*, es una unidad de trabajo que el cliente desea llevar a cabo: se trata de los datos de entrada, el programa de *MapReduce* y la información de configuración. *Hadoop* ejecuta el trabajo dividiéndolo en tareas, de las cuales hay dos tipos: tareas reducir y tareas mapa.

Hay dos tipos de nodos que controlan el proceso de ejecución del trabajo: un *jobtracker* y un número de *tasktracker*s. El *jobtracker* coordina todos los trabajos que se ejecutan en el sistema mediante la programación de tareas para ejecutarse en el *tasktracker*. Los *tasktrackers* ejecutan tareas y envían los informes de progreso al *jobtracker*, que mantiene un registro de la evolución general de cada puesto de trabajo. Si una tarea falla, el *jobtracker* puede reprogramar en un *tasktracker* diferente. *Hadoop* divide la entrada a un trabajo *MapReduce* en trozos de tamaño fijo llamados división de entrada. *Hadoop* crea un mapa de tareas para cada división, el cual ejecuta la función *map* por el usuario definido para cada registro en la división(White 2009).

Hadoop hace lo mejor para ejecutar la tarea de *mapa* en un nodo donde los datos de entrada residan en HDFS. Esto se le llama optimización de localidad de datos. Ahora debería quedar claro por qué el tamaño de la división óptima es el mismo que el tamaño de bloque: es el mayor tamaño de entrada que puede ser garantizado a ser almacenado en un solo nodo. Si la división se extendió a dos bloques, sería poco probable que cualquier nodo HDFS haya almacenado ambos bloques, por lo que parte de la división tendría que ser transferida a través de la red para el nodo que ejecuta la tarea mapa, que es claramente menos eficaz que correr toda la tarea mapa utilizando los datos locales.

Las tareas *mapas* escriben su salida en un disco local, no así en HDFS. Esto es debido a que las salidas de mapa son salidas intermediarias: son procesadas por tareas *reducir* para producir la salida final, y una vez el trabajo esté completo el mapa de salida puede ser lanzado. Por tanto, almacenándolo en HDFS, con replicación, sería una exageración.

Si el nodo que ejecuta la tarea mapa falla antes de que la salida del mapa haya sido consumida por la tarea de reducir, entonces *Hadoop* automáticamente volverá a ejecutar el mapa de tareas en otro nodo para recrear el mapa de salida. Las tareas reducir no tienen la ventaja de localidad de datos a la entrada de una sola tarea, es normalmente reducir la salida de todos los mapeadores. El flujo de datos completo con una simple tarea reducir es ilustrada en la Figura 5. Los cuadros discontinuos representan los nodos, las flechas delgadas representan la transferencia de datos en un nodo y las flechas gruesas muestran la transferencia de datos entre nodos.

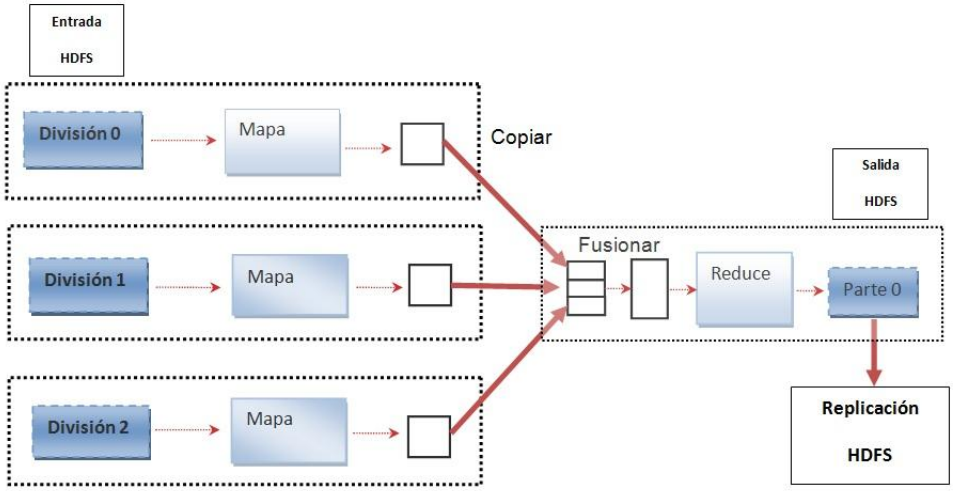


Figura 5: **Flujo de datos de *MapReduce* con una simple tarea reducir**.

Cuando hay varios reductores, la partición de las tareas *mapa* de su producción, cada una crea una partición para cada tarea *reducir*. No puede haber muchas claves en cada partición, pero los registros de todas las claves están todos en una sola partición. La partición puede ser controlada por un usuario de una función definida por el particionado. El flujo de datos para el caso general de múltiples tareas reducir se ilustra en la Figura 6.

Este esquema deja claro por qué ocurre el flujo de datos entre las tareas mapa y reducir respectivamente, que coloquialmente se conoce como "fusión", ya que cada tarea reducir es alimentada por muchas tareas mapa(White 2009).

Por último, también es posible tener cero tareas reducir. Esto puede ser apropiado cuando no necesita la "fusión" ya que el proceso puede llevarse a cabo completamente en paralelo. En este caso, el único nodo fuera de transferencia de datos es cuando las tareas del mapa escriben en HDFS (véase la Figura 7).

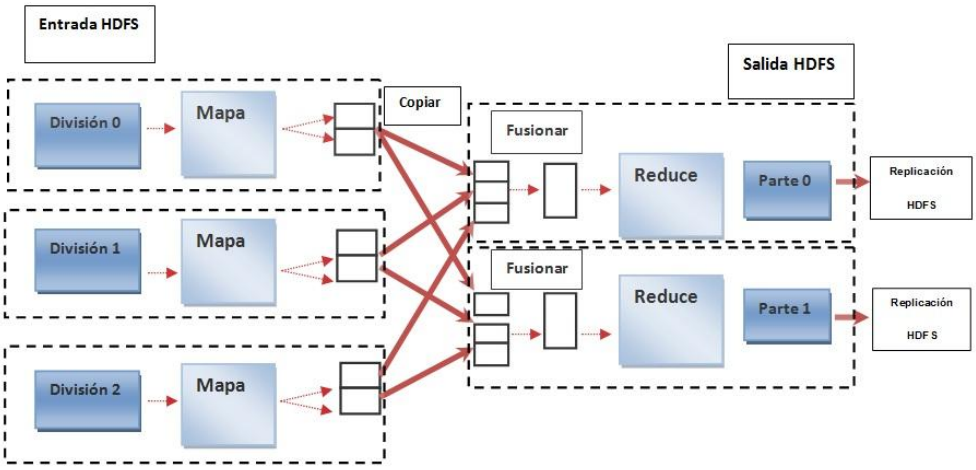


Figura 6: **Flujo de datos *MapReduce* con múltiples tareas reducir**.

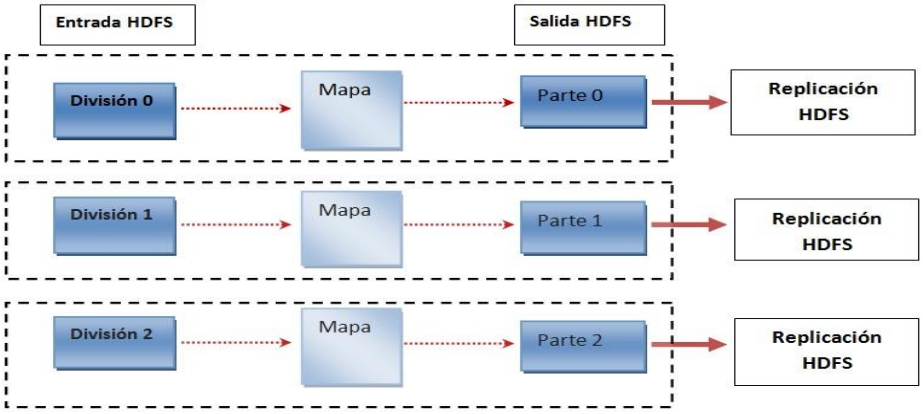


Figura 7: **Flujo de datos *MapReduce* sin tareas reducir**.

2.2.2 Entrada y salida de *Hadoop*

Hadoop proporciona una API para *MapReduce* que le permite escribir su mapa y reducir las funciones en lenguajes distintos de Java. La entrada y salida de *Hadoop* utiliza los estándares de Unix como interfaz

entre *Hadoop* y su programa, así que se puede utilizar cualquier lenguaje que pueda leer la entrada estándar y escriba en la salida estándar para escribir su programa de *MapReduce*.

La entrada y salida es, naturalmente, ideal para el procesamiento de textos (aunque en la versión 0.21.0 puede también manejar secuencias binarias), y cuando se utiliza en modo texto, tiene una visión orientada a las líneas de datos. Los datos del mapa de entrada se pasan sobre la entrada estándar a la función de su mapa, lo que procesa línea por línea y escribe las líneas de la salida estándar. Un mapa de salida par clave-valor se escribe como una sola ficha delimitado por la línea. La entrada para reducir la función es en el mismo formato - una ficha separada par clave-valor - pasada por la entrada estándar. La función *reduce* lee líneas desde la entrada estándar, con la cual la garantía del *framework* está ordenada por clave, y escribe sus resultados a la salida estándar(White 2009).

2.3 Sistema de Archivos Distribuido *Hadoop* (HDFS)

2.3.1 Diseño de HDFS

El sistema de archivos distribuido de *Hadoop* es un sistema de ficheros distribuidos inspirado en el ya existente Sistema de Archivos de *Google* (GFS) y está diseñado para ejecutarse en *hardware* de bajo costo. Tiene muchas similitudes con los actuales sistemas de archivos distribuidos, sin embargo las diferencias con los otros sistemas de archivos distribuidos son significativas al contar con características implementadas para hacer frente a los problemas que se presentan en el almacenamiento y transferencias de archivos en un ambiente distribuido. HDFS es altamente tolerante a fallos y está diseñado para ser implementado en *hardware* de bajo costo, proporciona acceso de alto rendimiento de datos de aplicación y es adecuado para aplicaciones que tienen grandes conjuntos de datos. HDFS está diseñado para soportar archivos de gran tamaño, los tamaños de bloques de archivos usados por HDFS son de 64 MB(ÁNGEL MERCHAN). Algunos elementos que confirman este criterio son los siguientes:

Archivos muy grandes

“Muy grande” en este contexto significa archivos que tienen cientos de *megabytes*, *gigabytes* o *terabytes* de tamaño. Actualmente, existen clúster de *Hadoop* corriendo sobre este sistema, almacenando *petabytes* de datos(White 2009).

Acceso de entrada y salida a datos

HDFS es construido sobre la idea de que la mayoría de los patrones eficientes de procesamiento de datos sea un patrón de escribir-una vez, leer-muchas veces. Un conjunto de datos es típicamente generado o copiado desde la fuente. Cada análisis implicará una extensa proporción de conjuntos de datos, de modo que el tiempo para leer este conjunto de datos completo es más importante que la latencia en la lectura del primer registro(White 2009).

Hardware básico

Hadoop no requiere *hardware* de alto costo para ser ejecutado. Está diseñado para ser corrido en un clúster de *hardware* básico (comúnmente *hardware* disponible en múltiples vendedores) por lo que la posibilidad de fallo de un nodo es altamente probable, al menos en grandes clúster(White 2009). HDFS está diseñado entonces para llevar a cabo trabajos sin interrupción perceptible al usuario en la fase de dicho fallo.

Baja latencia de acceso a datos

Las aplicaciones que requieren acceso a datos de baja latencia, en los 10 milisegundos de rango, no trabajarán bien sin HDFS(White 2009). Se debe recordar que HDFS está optimizado a entregar un alto rendimiento de datos y esto puede estar en el costo de la latencia.

Cantidades de achivos pequeños

Puesto que el *namenode* soporta metadatos de sistema de archivos en memoria, el límite al número de archivos en un sistema de archivos es regido por la cantidad de memoria en el *namenode*. Cada archivo, directorio, y bloque abarca sobre 150 *bytes*. Entonces, por ejemplo, si se tiene 1 millón de archivos, cada uno tomando un bloque, se necesitaría al menos 300 MB de memoria. Mientras el almacenamiento de millones de archivos sea factible, billones es más allá de la capacidad de *software* actual(White 2009).

Múltiples escritores, modificación de archivos arbitrarios

Los archivos en HDFS pueden ser escritos por un usuario (escritor) en singular. Las escrituras son siempre hechas al final del archivo. No hay un soporte para múltiples escritores, o para modificaciones de desplazamiento de archivos arbitrarios(White 2009).

El sistema de archivos distribuido *Hadoop* ofrece el acceso global a los archivos del clúster(M. ZUKOWSKI 2007). Para obtener la máxima portabilidad, HDFS se implementa como un sistema de archivos a nivel de usuario en lenguaje Java, que explota el sistema de ficheros nativo en cada nodo, como por ejemplo ext3 o NTFS, para almacenar datos. Los archivos en HDFS se dividen en grandes bloques, típicamente de 64MB, y cada bloque se almacena como un archivo separado en el sistema de ficheros local. HDFS se desarrolla en dos servicios: el *namenode* y *datanode*. El *namenode* es responsable de mantener el árbol de directorios HDFS, y es un servicio centralizado en el funcionamiento del clúster en un nodo único. Los clientes contactan con el *namenode* con el fin de realizar operaciones comunes del sistema de archivos, tales como abrir, cerrar, renombrar y borrar(W. TANTISIRIROJ 2008).

El *namenode* no almacena los datos de HDFS en sí, sino más bien mantiene una asignación entre el nombre del archivo HDFS, una lista de bloques en el archivo, y el *datanode*, en la que dichos bloques se almacenan. Cada *datanode* almacena bloques HDFS en nombre de los clientes locales o remotos. Cada bloque se guarda como un archivo separado en el sistema de archivos local del nodo. Debido a que el *datanode* abstrae los detalles del arreglo local de almacenamiento, todos los nodos no tienen que usar el mismo sistema de archivos local. Los bloques son creados y destruidos en los d*atanodes* a petición del *namenode*, el cual valida y procesa peticiones de los clientes. Aunque el *namenode* administra el espacio de nombres, los clientes se comunican directamente con los *datanodes* para poder leer o escribir datos a nivel de bloque HDFS.

Las aplicaciones *MapReduce* de *Hadoop* utilizan el almacenamiento de una manera que es diferente de la computación de propósito general(W. TANTISIRIROJ 2008).

En primer lugar, acceder a los archivos de grandes datos, típicamente de decenas a cientos de *gigabytes* de tamaño. En segundo lugar, estos archivos son manipulados a través de patrones de acceso de transmisión típica de procesamiento por lotes cuando la lectura de los archivos, los segmentos de datos de gran tamaño (varios cientos de *kilobytes* o más) se recuperan por operación, con solicitudes sucesivas de un mismo cliente iterando a través de una región de archivo secuencial. De manera similar, los

archivos también se escriben en una manera secuencial. Este énfasis en las cargas de trabajo de transmisión es evidente en el diseño de HDFS. La coherencia de un modelo simple (una sola escritura, varias lecturas) se utiliza de modo que no permita que los datos sean modificados una vez escritos. Esto está bien adaptado para el patrón de acceso de transmisión de aplicaciones de destino, y mejora la escala del clúster mediante la simplificación de los requerimientos de sincronización. En segundo lugar, cada archivo en el sistema se divide en bloques grandes para el almacenamiento y acceso, típicamente 64MB de tamaño. Las porciones del archivo pueden ser almacenadas en diferentes nodos del clúster, equilibrando los recursos de almacenamiento y demanda. La manipulación de los datos en este sistema es eficiente debido a que las aplicaciones de estilo de transmisión son más propensas a leer o escribir todo el bloque antes de pasar al siguiente bloque. Además, esta elección de diseño mejora el rendimiento al disminuir la cantidad de metadatos que deben ser rastreados en el sistema de archivos, y permite acceder a la latencia para ser amortizados en un gran volumen de datos.

De este modo, el sistema de archivos está optimizado para gran ancho de banda en lugar de baja latencia. Esto permite que aplicaciones no interactivas puedan procesar datos a velocidad más rápida. Para leer un archivo de HDFS, las aplicaciones clientes sólo tienen que utilizar un archivo estándar de Java para el flujo de entrada, como si el archivo fuese el sistema de archivos nativo(Jeffrey Shafer 2010).

Primeramente, el *namenode* solicita permiso de acceso. Si es concedido, este se traducirá en el nombre del archivo HDFS a una lista de identificadores de bloque HDFS que comprende ese archivo y una lista de *datanodes* que almacenan cada bloque, y devuelve las listas al cliente. Luego, el cliente abre una conexión al *datanode* "más cercano" y pide un ID de bloque específico. Este bloque HDFS se devuelve en la misma conexión, y se entregan los datos a la aplicación(Jeffrey Shafer 2010).

Para escribir datos sobre HDFS, las aplicaciones clientes ven el sistema HDFS como una secuencia de salida estándar. Internamente, sin embargo, los datos de acceso primero se fragmentan en bloques de tamaño HDFS y a continuación, los paquetes más pequeños (64KB) se fragmentan por el subproceso de cliente. Cada paquete se pone en cola en una FIFO (*First In First Out*) (primero en entrar, primero en salir) que puede almacenar hasta 5 MB de datos, por lo tanto la disociación del subproceso de la aplicación de la latencia del sistema de almacenamiento durante el funcionamiento es normal.

2.3.2 Replicación HDFS

Para la fiabilidad, HDFS implementa un sistema de replicación automática. Por defecto dos copias de cada bloque son almacenadas por *datanodes* diferentes en el mismo bastidor y la tercera copia es almacenada en un bastidor diferente. De esta forma, en una operación normal de clúster, cada *datanode* está sirviendo simultáneamente a ambos, los clientes locales y los clientes remotos.

La replicación del sistema HDFS es transparente a la aplicación cliente. Cuando se escribe un bloque, un ducto se establece mediante el cual el cliente sólo se comunica con el primer *datanode*, que luego refleja los datos a un segundo *datanode*, y así sucesivamente, hasta que el número deseado de réplicas hayan sido creadas. El bloque sólo se termina cuando todos los nodos en este ducto de la replicación hayan entregado todos los datos al disco. Los *datanodes* reportan periódicamente una lista de todos los bloques almacenados en el *namenode*, que verificará que cada archivo es lo suficientemente replicado y, en caso de fallo, instruir a los *datanodes* para hacer copias adicionales(Jeffrey Shafer 2010).

HDFS está diseñado para almacenar de forma segura, archivos muy grandes a través de las máquinas en grandes clúster. Almacena cada archivo como una secuencia de bloques; todos los bloques en un archivo, excepto el último bloque que es del mismo tamaño.

Los bloques de un archivo se replican para tolerancia a fallos. El tamaño de bloque y el factor de replicación son configurables por archivo. Una aplicación puede especificar el número de réplicas de un archivo. El factor de replicación puede ser especificado en el momento de creación del archivo y puede ser cambiado más adelante. Los archivos en HDFS son de una sola escritura y tienen estrictamente un escritor en cualquier momento. El *namenode* toma todas las decisiones con respecto a la replicación de los bloques. Se recibe periódicamente un “latido del corazón” y un reporte de bloque de cada uno de los *datanodes* en el clúster. La recepción de un latido del corazón implica que el *datanode* está funcionando correctamente. Un *blockreport* o reporte de bloques contiene una lista de todos los bloques en un *datanode*(BORTHAKUR 2007).

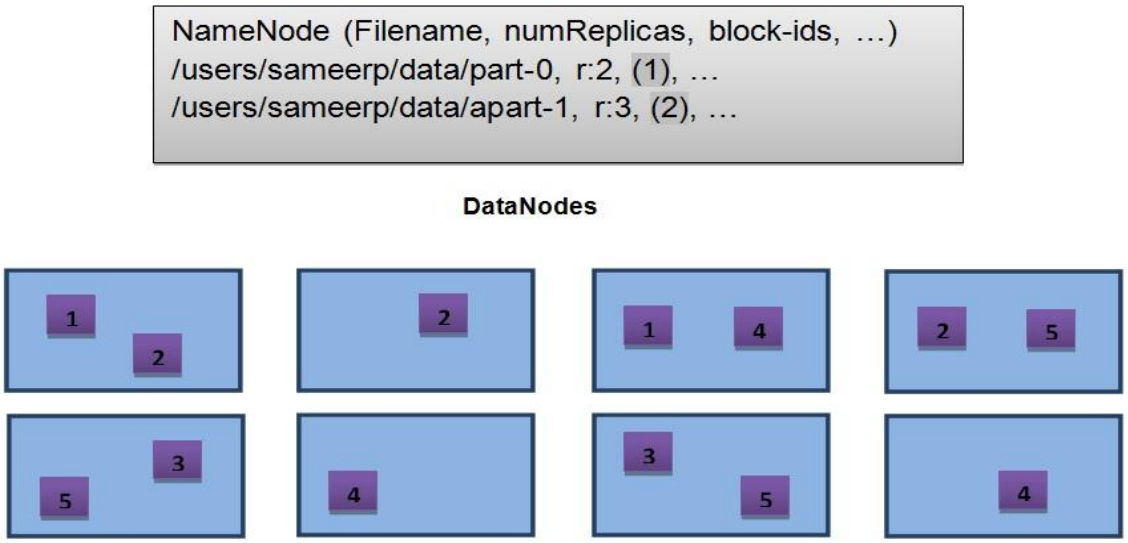


Figura 8: **Replicación de bloques.**

Réplica de colocación

La réplica de colocación es crítica para la fiabilidad y el rendimiento de HDFS. La optimización de la réplica de colocación distingue HDFS de la mayoría de otros sistemas de archivos distribuido. Esta es una característica que necesita mucha afinación y experiencia. El propósito de una política de réplica de colocación en un sistema de ficheros es para mejorar la fiabilidad de los datos, la disponibilidad y utilización del ancho de banda de red. La mayoría de los casos de HDFS se ejecutan en un clúster de equipos que se propaga a través de muchos bastidores. El *namenode* determina el ID del canal al cual pertenece cada *datanode* a través del proceso descrito anteriormente. Una política simple, pero no óptima, es colocar réplicas en bastidores únicos. Esto previene la pérdida de datos cuando un bastidor completo, falla y permite el uso de ancho de banda a partir de múltiples bastidores al leer los datos. Esta política distribuye réplicas uniformemente en el grupo que hace fácil equilibrar la carga de fallo de un componente. Sin embargo, esta política aumenta el costo de operaciones de escritura porque una escritura necesita transferir bloques a varios bastidores(BORTHAKUR 2007).

Para el común de los casos, cuando el factor de replicación es de tres, la política de colocación de HDFS es poner una réplica en un nodo en el bastidor local, otra en un nodo diferente en el bastidor local, y el último en un nodo diferente en un bastidor diferente. Esta política reduce el tráfico de escritura inter-bastidor que en general mejora el rendimiento de escritura. La probabilidad de fallo del bastidor es mucho

menor que la probabilidad de fallo en el nodo, para lo cual la política no afecta la confiabilidad de los datos y garantías de disponibilidad. Sin embargo, se reduce el ancho de banda agregado utilizado cuando la lectura de datos desde un bloque se coloca en sólo dos bastidores únicos en lugar de tres. Con esta política, las réplicas de un archivo no presentan una distribución uniforme a través de los bastidores. Una tercera parte de las réplicas están en un nodo, dos terceras partes de las réplicas se encuentran en un bastidor, y el otro tercio se distribuyen uniformemente a través de los bastidores restantes. Esta política mejora el rendimiento de escritura, sin comprometer la fiabilidad de los datos y el rendimiento de lectura(BORTHAKUR 2007).

Réplica de selección

Para minimizar el consumo de ancho de banda global y la latencia de lectura, HDFS trata de satisfacer una solicitud de lectura a partir de una réplica que está más cerca del lector. Si existe una réplica en el mismo bastidor que el nodo lector, entonces se prefiere esa réplica para satisfacer la petición de lectura. Si el clúster HDFS se extiende por múltiples centros de datos, entonces, una réplica que es residente en el centro de datos local es preferida sobre cualquier réplica remota(BORTHAKUR 2007).

2.4 Suposiciones y metas

2.4.1 Fallas de *hardware*

Las fallas de *hardware*, son más bien la norma que la excepción. Una instancia de HDFS puede consistir en cientos y miles de máquinas servidores, cada una almacenando parte de los datos de los sistemas de archivos. El hecho que haya un amplio número de componentes y que cada componente tiene una probabilidad de fallo no trivial, significa que algunos componentes de HDFS son siempre no funcionales. Además, la detección rápida de fallos y la recuperación automática de ellos es una meta arquitectónica central del sistema de ficheros HDFS.(BORTHAKUR 2007)

2.4.2 Entrada y salida de datos

Las aplicaciones que corren sobre HDFS necesitan un acceso de entrada y salida a sus configuraciones de datos. No existen aplicaciones de propósito general que típicamente corren en sistemas de archivos de propósito general. HDFS es diseñado más para procesamiento por lotes que para uso interactivo por los usuarios. El énfasis es más alto durante todo el acceso a datos que la baja latencia del mismo.

2.4.3 Modelo de coherencia simple

Las aplicaciones de HDFS necesitan un modelo de acceso de una sola escritura, muchas lecturas para los archivos. Un archivo, una vez que haya sido creado, escrito, y una vez cerrado no necesita ser cambiado. Este supuesto simplifica los problemas de coherencia de datos y permite un alto rendimiento de acceso a datos. Una aplicación de *MapReduce* o una aplicación rastreador *web* se ajusta perfectamente a este modelo.

2.5 *Namenode* y *datanode*

Un clúster HDFS tiene dos tipos de nodos operando en un patrón maestro-esclavo. Un *namenode* (maestro) y una serie de *datanodes* (esclavos). El *namenode* gestiona el espacio de nombres en el sistema de archivos y mantiene el árbol del sistema de archivos y los metadatos para todos los archivos y directorios del árbol. Esta información es almacenada persistentemente en un disco local en forma de dos archivos: la imagen del nombre de espacio y el *edit log* o registro editor(White 2009).

El *namenode* también identifica los datanode*s* en el cual todos los bloques para un archivo dado están localizados, sin embargo, no almacena la localización de los bloques persistentemente, puesto que esta información es reconstruida desde los datanode*s* cuando el sistema arranca. Sin el namenode, el sistema de archivos no puede ser usado. De hecho, si la ejecución del *namenode* en la máquina fuera eliminada, todos los archivos del sistema de archivos serían perdidos puesto que no habría forma de saber cómo reconstruir los archivos desde los bloques en los *datanodes*(White 2009). Hadoop crea mecanismos para que el *namenode* sea resistente a fallos. El primero es respaldar los archivos y hacer el estado persistente de los metadatos en el sistema de archivos. *Hadoop* puede ser configurado de forma tal que el *namenode* escriba su estado persistente en múltiples sistemas de archivos. La alternativa de configuración usual es escribir sobre un disco local de la misma forma que en un montaje de NFS remoto.

HDFS expone un espacio de nombres en el sistema de archivos y permite que los datos de usuarios se almacenen en archivos. Internamente, un archivo se divide en uno o más bloques y los bloques de estos se almacenan en un conjunto de *datanodes*. El *namenode* ejecuta operaciones de espacios de nombres en el sistema de archivos como abrir, cerrar y renombrar nombres y directorios. Los *datanodes* son responsables de que se lean y escriban las peticiones de los clientes sobre el sistema de archivos. Los

datanodes también llevan a cabo la creación de bloques, eliminación, y la replicación en la instrucción desde el n*amenode*(BORTHAKUR 2007).

2.6 Arquitectura del sistema HDFS

El *namenode* y *datanode* son piezas de *software* diseñados para funcionar con los productos básicos de máquinas. Estas máquinas suelen ejecutar el sistema operativo GNU / Linux. HDFS se construye utilizando el lenguaje Java, cualquier máquina que soporte Java puede ejecutar el *namenode* o el *software* de *datanode*. El uso del lenguaje Java altamente portátil significa que HDFS puede ser desplegado en una amplia gama de máquinas. Una implementación típica tiene un equipo dedicado que sólo ejecuta el *software namenode*. Cada una de las otras máquinas en el clúster ejecuta una instancia del *software* de d*atanode*. La arquitectura no se opone a ejecutar varios *datanodes* en la misma máquina pero sí en una implementación real, que es raramente el caso. La existencia de un solo *namenode* en un grupo de máquinas, simplifica enormemente la arquitectura del sistema. El *namenode* es el árbitro y el repositorio de todos los metadatos de HDFS. El sistema es diseñado de tal forma que nunca los datos de usuario fluyen a través del *namenode*.

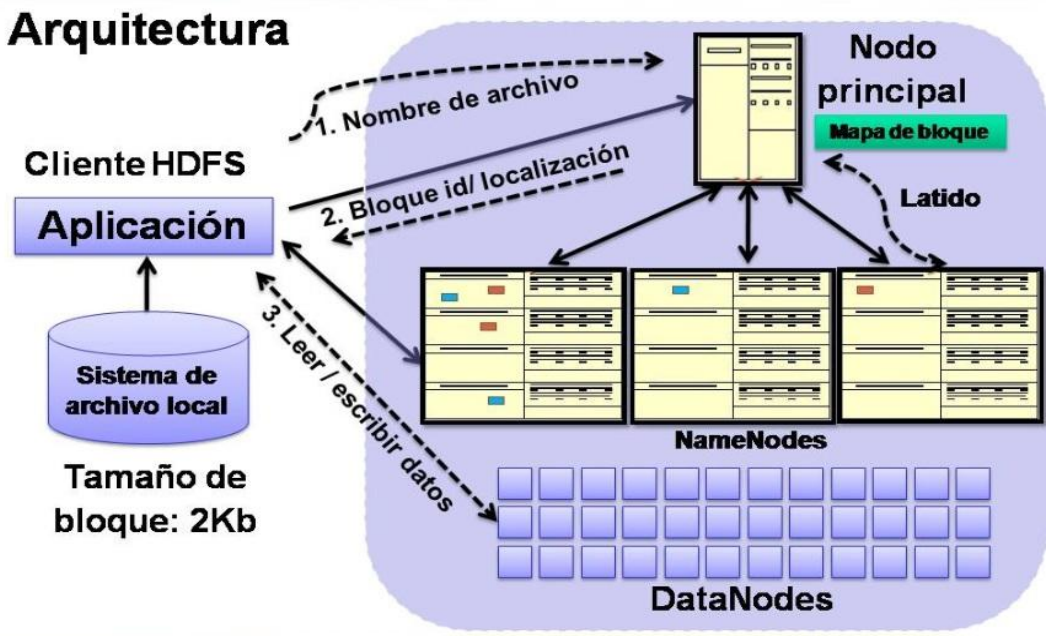


Figura 9: **Arquitectura del sistema HDFS**.

2.7 Espacio de nombres de archivos del sistema

HDFS es compatible con una organización tradicional de archivos jerárquicos. Un usuario o una aplicación pueden crear directorios y almacenar archivos dentro de estos. El espacio de nombres del sistema de archivos jerárquico es similar a la mayoría de los otros sistemas de archivos existentes, se puede crear y eliminar archivos, mover un archivo de un directorio a otro, o cambiar el nombre de un archivo. HDFS aún no implementa cuotas de usuarios o permisos de acceso. Sin embargo, la arquitectura de HDFS no se opone a la aplicación de estas características. El *namenode* mantiene el espacio de nombres del sistema de archivos. Cualquier cambio en el espacio de nombres del sistema de archivos o en sus propiedades es registrado por el *namenode*. Una aplicación puede especificar el número de réplicas de un archivo que deben ser mantenidas por HDFS. El número de copias de un archivo es llamado el factor de replicación de ese archivo. Esta información es almacenada por el *namenode*(BORTHAKUR 2007).

2.8 Modo seguro

En el arranque, el *namenode* entra en un estado especial llamado “modo seguro”. La replicación de bloques de datos no se produce cuando el *namenode* está en el estado “modo seguro”. El *namenode* recibe *latidos del corazón* (proceso de información que permite conocer si todas las tareas fueron ejecutadas correctamente) y mensajes de reporte de bloques desde los *datanodes*. Un reporte de bloques contiene la lista de bloques de datos que el *datanode* está organizando. Cada bloque tiene un número mínimo especificado de réplicas. Un bloque se considera replicado de modo seguro cuando el número mínimo de réplicas de bloques de datos se ha comprobado en el *namenode*. Después de un porcentaje configurable de bloques de datos replicados de manera segura, se chequea en los controles con el *namenode* (más un adicional de 30 segundos), el *namenode* sale del estado modo seguro. Se determina entonces la lista de bloques de datos (si existe) que todavía tiene un poco menos que el número especificado de réplicas. El *namenode* entonces replica estos bloques a otros *datanodes*(GHEMAWAT 2004; BORTHAKUR 2007).

2.9 Persistencia de los metadatos del sistema de archivos

El espacio de nombres HDFS es almacenado por el *namenode*. El n*amenode* utiliza un registro de transacciones llamado (editor de registro) el cual se usa para registrar persistentemente cada cambio que se produce en los metadatos del sistema de archivos. Por ejemplo, crear un nuevo archivo en HDFS hace

que el n*amenode* inserte un registro en el editor de registros. De manera similar, cambiando el factor de replicación de un archivo hace un nuevo registro para ser insertado en el editor de registros. El *namenode* utiliza un archivo en el sistema operativo de su máquina local para almacenarlo en el editor de registros. El archivo completo de espacio de nombres del sistema, incluyendo la asignación de bloques a los archivos y las propiedades del sistema de archivos, se almacenan en un archivo llamado *FsImage*. El *FsImage* se almacena de igual forma como un archivo en el sistema de archivos local del *namenode*(Jeffrey Shafer 2010).

El *namenode* mantiene una imagen de todo el espacio de nombres del sistema de archivos y del archivo del mapa de bloques (*Blockmap*) en memoria. Este elemento de metadatos claves, está diseñado para ser compacto, de tal manera que un *namenode* con 4GB de RAM es suficiente para apoyar un gran número de archivos y directorios. Cuando el *namenode* arranca, lee el *FsImage* y el editor de registro desde el disco, aplica todas las transacciones desde el editor de registro hasta la representación de la memoria interna del *FsImage*. Puede entonces truncar el antiguo editor de registros debido a que sus transacciones se han aplicado al *FsImage* persistente. Este proceso se llama “punto de control”. En la implementación actual, un “punto de control” se produce cuando el *namenode* se pone en marcha. El *datanode* no tiene el conocimiento acerca de los archivos HDFS. Almacena cada bloque de datos HDFS en un archivo separado en su sistema de archivos local.

El *datanode* no crea todos los archivos en el mismo directorio. En su lugar, utiliza una heurística para determinar el número óptimo de archivos por directorio y crea subdirectorios apropiadamente. No es óptimo para crear todos los archivos locales en el mismo directorio ya que el sistema de archivos local no podría ser capaz de soportar de manera eficiente un gran número de archivos en un solo directorio. Cuando un *datanode* arranca, escanea a través de su sistema de archivos local, genera una lista de todos los bloques de datos HDFS que corresponden a cada uno de estos archivos locales y envía este informe al *namenode*: este es el *blockreport* o reporte de bloques(BORTHAKUR 2007).

2.10 Protocolos de comunicación

Todos los protocolos de comunicación del sistema de archivos HDFS se superponen en la parte superior del protocolo TCP / IP. Un cliente establece una conexión a un puerto TCP configurable en la máquina *namenode*. Los *datanodes* comunican con el *namenode* utilizando el protocolo *datanode*. Una llamada de

procedimiento remoto (RPC) de la abstracción se envuelve tanto en el cliente "protocolo" como en el protocolo del *datanode*. Por diseño, el *namenode* nunca inicia ningún RPC. Por el contrario, sólo responde a las peticiones RPC emitidas por *datanodes* o clientes(Jeffrey Shafer 2010).

2.11 Robustez

El objetivo principal de HDFS es el de almacenar datos de forma fiable, incluso en presencia de fallos. Los tres tipos comunes de fallos son: fallos del *namenode*, fallos del *datanode* y las particiones de red.

2.11.1 Fallos del disco duro, *heartbeats* y la re-replicación

Cada *datanode* envía periódicamente un mensaje de "*heartbeat*" (forma en que se transmite la información de que todas las tareas procesadas en los datanodes, fueron llevadas a cabo sin problemas) al *namenode*. Una partición de red puede causar un subconjunto de *datanodes* a perder la conectividad con el *namenode*. El *namenode* detecta esta condición por la ausencia de mensajes de *heartbeat*. El *namenode* lleva a cabo un seguimiento constante con los bloques que deben ser replicados e inicia la replicación cuando sea necesario. La necesidad de re-replicación puede surgir debido a muchas razones: un *datanode* puede llegar a no estar disponible, una réplica puede estar dañada, un disco duro en un *datanode* puede fallar, o el factor de reproducción de un archivo puede ser mayor(Jeffrey Shafer 2010).

2.11.2 Clúster de reequilibrio

La arquitectura del sistema de archivos *Hadoop* es compatible con los datos de planes de reequilibrio. Un esquema podría automáticamente mover los datos de un *datanode* a otro, si el espacio libre en un uno de ellos cae debajo de cierto umbral. En el caso de una demanda alta repentina para un archivo en particular, un esquema dinámico podría crear réplicas adicionales y volver a equilibrar los datos del clúster. Estos tipos de datos de planes de reequilibrio no se han implementado todavía(Jeffrey Shafer 2010).

2.11.3 Integridad de los datos

Es posible que un bloque de datos obtenidos a partir de un *datanode* llegue dañado. Esta corrupción puede ocurrir debido a fallas en un dispositivo de almacenamiento, fallas de red o *software* defectuoso. El *software* del cliente de control del sistema de archivos *Hadoop,* implementa la comprobación de los contenidos de los archivos HDFS. Cuando un cliente crea un archivo HDFS, se calcula una suma de comprobación de cada bloque del archivo y almacena estos al archivo de sumas de comprobación

separado en el mismo espacio de nombres HDFS. Cuando un cliente recupera el contenido de los archivos, se comprueba que los datos que se reciben de cada *datanode* corresponden con la suma de comprobación y almacena estos en un archivo oculto separado en el mismo espacio de nombres del HDFS(Jeffrey Shafer 2010).

2.11.4 Fallos de disco en los metadatos

El *FsImage* y el *EditLog* (editor de registro) son estructuras de datos centrales del sistema de archivos HDFS. Una corrupción de estos archivos puede causar una instancia de HDFS a ser no funcional. Por esta razón, el *namenode* puede ser configurado para soportar el mantenimiento de múltiples copias del *FsImage* y el *EditLog*. Cualquier actualización ya sea al *FsImage* o al *EditLog* hace que cada uno de los *FsImages* y *EditLogs* se actualizan sincrónicamente. Esta actualización sincrónica de varias copias del *FsImage* y del *EditLog* puede degradar el ritmo de las transacciones de espacio de nombres por segundo que un *namenode* puede soportar. Sin embargo, esta degradación es aceptable, ya que aunque las aplicaciones del sistema HDFS son muy intensivas en datos, no hay presencia de metadatos intensivos. Cuando se reinicia un *namenode*, se selecciona el último *FsImage* coherente y el *EditLog* a usar. La máquina *namenode* es un punto único de fallo para un clúster de HDFS. Si el n*amenode* de la máquina falla, la intervención manual es necesaria. En la actualidad, el reinicio automático y conmutación por error del *software namenode* a otro equipo no es compatible(Jeffrey Shafer 2010).

2.12 Organización de datos

2.12.1 Bloques de datos

HDFS está diseñado para soportar archivos de gran tamaño. Las aplicaciones que sean compatibles con HDFS son aquellas que se ocupan de grandes conjuntos de datos. Estas aplicaciones escriben sus datos una sola vez, pero ellas las leen una o más veces y se requieren de estas lecturas para satisfacer las velocidades de transmisión. HDFS es compatible con la semántica de una sola escritura, muchas lecturas en los archivos. Un tamaño de bloque típico usado por HDFS es 64MB. De este modo, un archivo de HDFS es cortado en trozos de 64MB, y si es posible, cada trozo residirá en un *datanode* diferente(Jeffrey Shafer 2010).

2.13 Accesibilidad

El sistema de archivos HDFS puede ser accedido desde aplicaciones en muchas formas diferentes. De forma nativa, HDFS proporciona una API de Java para las aplicaciones a utilizar. También está disponible un paquete del lenguaje C para esta API de Java. Además, también se puede utilizar un navegador HTTP para navegar por los archivos de una instancia del sistema HDFS.

2.13.1 FS Shell

HDFS permite que los datos de los usuarios se organicen en forma de archivos y directorios. Se proporciona una interfaz de línea de comandos de Shell llamada FS que permite al usuario interactuar con los datos de HDFS. La sintaxis de este conjunto de comandos es similar a otros *shells* (por ejemplo *bash*, csh) que los usuarios ya están familiarizados. Estos son algunos ejemplos de acción de comandos / pares:

Tabla 1: **Acciones de comandos pares FS**.

Acción Comando	
Crear un directorio llamado /foodir	bin/Hadoop dfs –mkdir /foodir
Vista de contenidos de un archivo llamado: /foodir/miarhivo.txt	bin/Hadoop dfs -cat/foodir/miarchivo.txt

FS *shell* está orientado a aplicaciones que necesitan un lenguaje de programación para interactuar con el almacenado de datos.

2.13.2 DFSAdmin

El conjunto de comandos *DFSAdmin* se utiliza para administrar un clúster HDFS. Estos son comandos que se utilizan sólo por un administrador HDFS. A continuación se muestran algunos ejemplos de la acción de comandos:

Tabla 2: **Ejemplos de conjuntos de comandos para administrar clúster HDFS.**

Acción Comando	
Poner el clúster en modo seguro	bin/hadoop dfsadmin -safemode enter
Generar una lista de *datanodes*	bin/hadoop dfsadmin -report
Decomission Datanode Datanodename	bin/hadoop dfsadmin –decomission datanodename

2.13.3 Interfaz del navegador

Una típica instalación HDFS configura un servidor web para exponer el espacio de nombres a través de un puerto TCP HDFS configurable. Esto permite a un usuario navegar por el espacio de nombres HDFS y ver el contenido de sus archivos a través de un navegador web.

2.14 Borrado de archivos y recuperación

Cuando se elimina un archivo por un usuario o una aplicación, no se retira inmediatamente de HDFS. En su lugar, HDFS primero cambia el nombre a un archivo en el directorio /trash. El archivo se puede restaurar rápidamente siempre y cuando se mantenga en este directorio. Un archivo se mantiene en /trash durante una cantidad de tiempo configurable. Después de la expiración de su vida, el *namenode* elimina el archivo del espacio de nombres HDFS. La eliminación de un archivo hace que los bloques asociados con el archivo sean liberados. Hay que tener en cuenta que podría haber un retraso de tiempo considerable entre el momento en que se elimina un archivo por un usuario y el tiempo del aumento correspondiente en el espacio libre en HDFS. Un usuario puede recuperar un archivo después de eliminarlo, siempre y cuando se mantenga en el directorio /trash. Si un usuario desea recuperar un archivo que ha eliminado, entonces, este puede navegar por el directorio /trash y recuperar el archivo. El directorio sólo contiene la copia más reciente del archivo que se ha eliminado. El directorio /trash es igual que cualquier otro directorio, con una especial característica: HDFS aplica políticas específicas para eliminar automáticamente los archivos de este directorio. La política por defecto actual es eliminar los archivos de /trash que tengan más de 6 horas de vida. En posteriores implementaciones, esta política se puede configurar a través de una interfaz bien definida(Jeffrey Shafer 2010).

2.14.1 Disminución del factor de replicación

Cuando el factor de replicación de un archivo se reduce, el *namenode* selecciona réplicas en exceso que se pueden eliminar. El siguiente *heartbeat* transfiere esta información al *datanode*. El *datanode* a continuación, elimina los bloques correspondientes y el espacio libre correspondiente que aparece en el clúster. Una vez más, podría haber un retardo de tiempo entre la realización de la llamada del *setReplication* a la API y la aparición de espacio libre en el clúster(Jeffrey Shafer 2010).

2.15 Instalación y configuración de HDFS.

Para el presente trabajo de diploma, la instalación y configuración se ha establecido con las siguientes versiones de *software*.

- Ubuntu, Linux 10.10 *Desktop Edition*
- *Hadoop* 0.18.3.0 el cual se puede encontrar en:

(http://www.apache.org/dyn/closer.cgi/hadoop/core).

2.15.1 Requisitos previos

Los requisitos para la instalación de *Hadoop* 0.18.3.0

- *Sun Java* 1.6 *update* 30 o superior,
- El servidor SSH deben de estar instalado,
- *Apache Hadoop* 0.18.3.0

Instalación de Java JDK

Paso 1: Lo primero es descargar el SUN/Oracle Java JDK del siguiente enlace:

http://www.oracle.com/technetwork/java/javase/downloads/index.html

Paso 2: Posteriormente es necesario establecerse como usuarios *root* con el siguiente comando:

```
1. user@ubuntu:~# su
```

Paso 3: Ejecutar los siguientes comandos para comenzar la instalación del JDK.

```
1. user@ubuntu:~# chmod +x /path/to/file/jdk-6u22-linux-i586-rpm.bin
2. user@ubuntu:~# /path/to/binary/jdk-6u22-linux-i586-rpm.bin
```

Paso 4: Instalar Java JDK de Oracle con el siguiente comando.

```
1. $ sudo apt-get install sun-java6-jdk
```

Paso 5: Verificar las versiones instaladas del JDK.

```
1. user@ubuntu:~#javac -version
2. javac 1.6.0.22
3. user@ubuntu:~#java –version
4. java version “1.6.0.22”
5. Java(TM) SE Runtime Environment (build 1.6.0.22-b04)
6. Java HotSpot (TM) Client VM (build 17.1-b03, mixed mode, sharging)
```

Creación del usuario *Hadoop*

Es necesario crear una cuenta de usuario para ejecutar *Hadoop*. Aunque esta operación no es requerida, sí se recomienda pues ayuda a separar la instalación de *Hadoop* de otras aplicaciones de *software* y de los usuarios ejecutándose en la misma máquina (tener en cuenta: seguridad, permisos, *backups*). Para realizar la operación se puede acceder mediante el siguiente comando:

```
1. $sudo addgroup hadoop
2. $sudo adduser --ingroup hadoop hduser
```

De esta forma se agregará el usuario *hduser* al grupo *hadoop.*

Instalación de SSH

Para instalar SSH solo se ejecuta el siguiente comando:

```
1. $ sudo apt-get install ssh
```

Configuración de SSH

Hadoop requiere acceso SSH para administrar sus nodos. Para el caso del nodo simple, se necesita además configurar el acceso SSH al *localhost* para el usuario *hduser* creado anteriormente. Para ello se utiliza:

```
1. user@ubuntu:~$ su - hduser
2. hduser@ubuntu:~$ ssh –keygen –t rsa -P “”
```

La segunda línea crea una llave RSA con una contraseña vacía. Luego se debe habilitar el acceso SSH a la máquina local con la nueva llave creada a través del comando:

```
1. hduser@ubuntu:~$ cat $HOME/.ssh/id_rsa.pub >> $HOME/.ssh/authorized_keys
```

Deshabilitando IPv6

Para deshabilitar IPv6 en Ubuntu 10.10 *Desktop Edition*, se abre: /etc/sysctl.conf en el editor pertinente y se agregan las siguientes líneas al final del archivo:

```
1. #disable ipv6
2. net.ipv6.conf.all.disable_ipv6=1
3. net.ipv6.conf.default.disable_ipv6=1
4. net.ipv6.conf.lo.disable_ipv6=1
```

Se debe reiniciar la máquina de modo que los cambios sufran efectos. Luego se puede a probar si IPv6 está deshabilitado utilizando el siguiente comando. De estar deshabilitado devolverá el valor 1:

```
1. $cat /proc/sys/net/ipv6/conf/all/disable_ipv6
```

Una vez garantizada dichas instalaciones y configuraciones, pues se procede a la instalación del sistema.

2.15.2 Instalación del sistema

Paso 1: Descargar el paquete *Hadoop* del sitio de *Apache Download Mirrors*

(http://www.apache.org/dyn/closer.cgi/hadoop/core). En este caso se utiliza el paquete con la versión 0.18.3.0.tar.gz.

Paso 2: Una vez descargado el archivo *.tar*, se instala el paquete con el comando tar xzf, y posteriormente se cambia el propietario del directorio con el comando *chown.*

```
1. $tar xzf hadoop-0.18.3.0.tar.gz
2. $chown –R hadoop hadoop-0.18.3.0
```

Actualizando $HOME/.bashrc

Se agregan las siguientes líneas al final del archivo $HOME/.bashrc:

```
1. export HADOOP_HOME=/usr/lib/hadoop
2. export JAVA_HOME=/usr/lib/jvm/java-6-sun
```

Configuración de directorios

A continuación se configurará el directorio donde *Hadoop* almacena sus archivos de datos, puertos de red, etc. Se usa para esta investigación el directorio */app/hadoop/tmp*, por lo que se crea el directorio y se configuran los propietarios y permisos requeridos con el siguiente comando:

```
1. $sudo mkdir -p /app/hadoop/tmp
2. $sudo chown hduser:hadoop /app/hadoop/tmp
3. $sudo chmod 750 /app/hadoop/tmp
```

Configurando archivos XML

Para configurar los archivos XML necesarios, se añaden las siguientes líneas de código en cada archivo correspondiente:

En el archivo **conf/core-site.xml**

[sourcecode]

```
<configuration>
 <property>
 <name>fs.default.name</name>
 <value>hdfs://namenode:9000</value>
```

```
</property>
</configuration>
```

En el archivo **conf/mapred-site.xml**

[sourcecode]

```
<configuration>
<property>
<name>mapred.job.tracker</name>
<value>namenode:9001</value>
</property>
</configuration>
```

[/sourcecode]

En el archivo **conf/hdfs-site.xml**

[sourcecode]

```
<configuration>
<property>
<name>dfs.replication</name>
<value>3</value>
</property>
</configuration>
```

[/sourcecode]

Formateando el sistema de archivos HDFS a través del *namenode*

Para comenzar con la instalación del sistema, se procede a formatear el sistema de archivos *Hadoop*. No se debe formatear el sistema de archivos mientras esté corriendo, pues se perderán todos los datos actuales del clúster (en HDFS).

Para formatear el sistema de archivos (el cual simplemente inicializa el directorio especificado por la variable dfs.name.dir), se ejecuta el siguiente comando:

```
1. $/usr/lib/hadoop-0.18/bin/hadoop-0.18 namenode -format
```

Arrancando el sistema

Para arrancar el sistema después de haber formateado el *namenode,* se ejecuta el siguiente comando:

```
1. $/usr/lib/hadoop-0.18/bin/start-all.sh
```

Esto arrancará el *namenode*, d*atanode*, *jobtracker*, y el *tasktracker* del sistema.

También se puede chequear con el comando *netstat,* la lista de procesos de java que están ejecutándose dentro del sistema de archivos. Para ellos se utiliza el siguiente comando.

```
1. $sudo netstat –plten | grep java
```

Deteniendo el sistema

Para detener todos los servicios del sistema, se procede a través del siguiente comando:

```
1. $/usr/lib/hadoop-0.18/bin/stop-all.sh
```

Interfaces *web* de *Hadoop*

El sistema de archivos *Hadoop* viene con varias interfaces *web* las cuales se encuentran por defecto en las direcciones:

- http://localohost:50030/ - interfaz *web* para *MapReduce jobtracker*(s)
- http://localohost:50060/ - interfaz *web* para *tasktracker*(s)
- http://localohost:50070/ - interfaz *web* para el *namenode* HDFS

Estas interfaces *web* proveen información concisa sobre lo que está ocurriendo en el clúster HDFS.

Interfaz *web MapReduce jobtracker*(s)

La interfaz *web jobtracker* provee información sobre estadísticas generales de trabajos *MapReduce* en el clúster HDFS. También proporciona acceso a los *log* de archivos HDFS de la máquina local (la máquina sobre la cual está corriendo la interfaz *web*).Ver Anexo 10

Interfaz web *tasktracker*

La interfaz *web tasktracker* muestra las tareas que están corriendo y las que no están corriendo en el clúster. También proporciona acceso a los logs de archivos HDFS de la máquina local. Ver Anexo 11

Interfaz *web namenode* HDFS

Esta interfaz muestra un sumario del clúster incluyendo información sobre la capacidad total/restante, los nodos vivos/muertos. Adicionalmente permite buscar los espacios de nombres HDFS y permite ver el

contenido de los archivos en el buscador *web*. También proporciona acceso a los *logs* de archivos HDFS de la máquina local. Ver Anexo 12

2.15.3 Instalación del sistema en modo multi-modo

Una vez se tiene instalado y configurado *Hadoop* en modo *single-node* o modo simple, se configura para poder ejecutarlo en varias máquinas a la vez o en un clúster. Para este caso, se hace uso de tres máquinas en las cuales una trabajará como *namenode* (maestro) y las otras como *datanodes* (esclavos).

Configuración del sistema en modo simple

Se procede a la configuración de las 3 máquinas siguiendo los pasos descritos anteriormente de *Hadoop* en modo *single-node* o modo simple. A partir de aquí se le llamará a la máquina maestra "*master"* y a las esclavos, "*slave".* Antes de empezar a hacer nada, es necesario detener los servicios de *Hadoop* que se tengan arrancados, ejecutando el comando */stop-all.sh* en las tres máquinas.

Configuración de la red

La configuración descrita para esta investigación, es la de una red interna, con los ordenadores conectados por un *switch*. Para simplificar, se tiene que el *master* tiene la dirección de IP 10.53.7.229 y los dos *slave* con la dirección de IP 10.53.7.8 y 10.53.7.250 respectivamente. Para que las tres máquinas se reconozcan mutuamente, se actualiza el fichero */etc/hosts* en las tres máquinas respectivamente de la siguiente manera:

```
1. #/etc/hosts (for master AND slave)

10.53.7.229 master

10.53.7.8 slave

10.53.7.250 slave
```

Luego la arquitectura y configuración de los nodos quedaría como se muestra en la Figura 10.

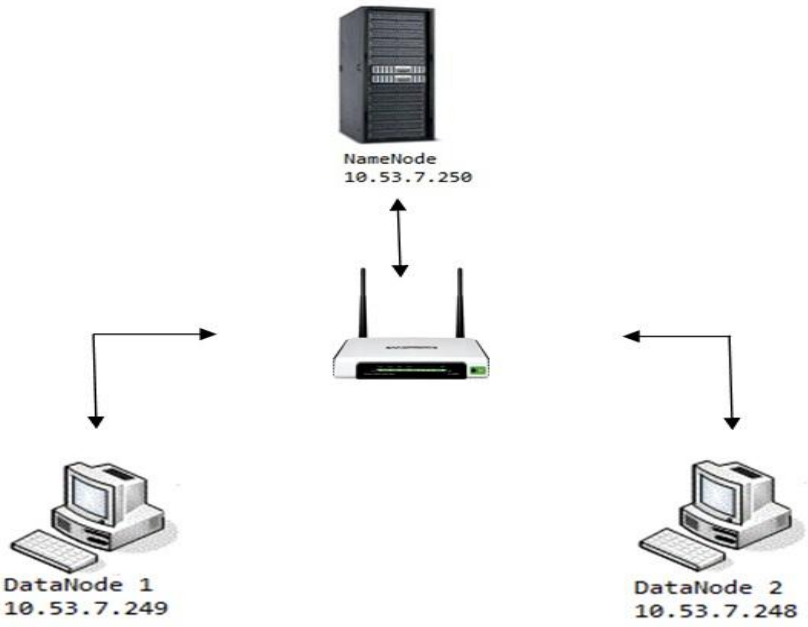


Figura 10: **Configuración de nodos en modo multi-nodo**.

Configuración SHH

Es necesario que el usuario *Hadoop* se pueda conectar desde el *master* al *slave* sin tener que utilizar *password*. Para ello se intercambia las claves entre el *master* y *slave* con el siguiente comando:

```
1.  hduser@master:~$ ssh-copy-id -i $HOME/.ssh/id_rsa.pub hadoop@slave
```

Una vez hecho esto, se comprueba desde el *master* que es posible conectarse al *slave* y al *master* con ssh *slave* y ssh *master* respectivamente. Si no solicita contraseña, es que se ha realizado todo correctamente.

De manera general, la estructura del nuevo clúster va a ser la siguiente: En el *master* se ejecutarán las tareas de *namenode*, d*atanode*, j*obtracker* y *tasktracker*, y en el *slave*, solamente *tasktracker* y *datanode*.

Configuración del *master*

Para configurar el *master*, se procede a modificar los siguientes ficheros:

conf/masters (solo el *master*)

A pesar de su nombre, el archivo *conf/masters* define en cuáles máquinas *Hadoop* arrancará el *secondary namenode* en el clúster multi-nodo. Para la configuración antes vista, en este caso será la máquina *master.* El *namenode* primario y el *jobtracker* siempre serán las máquinas en las cuales se ejecutan los *scripts bin/start-dfs.sh* y *bin/start-mapred.sh* respectivamente.

En el *master* actualizar *conf/masters* para lo cual quedaría:

```
master
```

conf/slaves (solo en el *master*)

En este fichero se definen los *hosts* línea por línea donde los demonios esclavos de *hadoop* (*datanodes* y *tasktrackers*) serán ejecutados. Cuando se arranque el clúster desde el *master*, se conectará a todas las máquinas definidas aquí y arrancará los procesos esclavos necesarios para ejecutar el clúster. En este fichero se ha de introducir una máquina por línea.

En el *master* actualizar el fichero *conf/slaves* para lo cual quedaría de la siguiente forma:

```
master
slave
```

Si se desean más nodos esclavos en el clúster, solo se agregan en el archivo *conf/masters* línea por línea (hacerlo en todas las máquinas en el clúster)

```
master
slave
nuevoslave1
nuevoslave2
```

Configuración en todas las máquinas

conf/core-site.xml

Se cambia la variable *fs.default.name* para que apunte al *master* como *namenode* del clúster:

[sourcecode]

```
<configuration>
 <property>
 <name>fs.default.name</name>
 <value>hdfs://master:54310</value>
 </property>
</configuration>
```

[/sourcecode]

conf/mapred-site.xml

Se cambia la variable *mapred.job.tracker* para que apunte también al *master* como *jobtracker*:

[sourcecode]

```
<configuration>
<property>
<name>mapred.job.tracker</name>
<value>master:54311</value>
</property>
</configuration>
```

[/sourcecode]

conf/hdfs-site.xml

Se cambia la variable *dfs.replication* que especifica el número de nodos en el que se va a replicar la información del servidor. Por defecto estará a 3 y es exactamente lo que se necesita para el ejemplo de configuración con el que se estará trabajando en esta investigación.

[sourcecode]

```
<configuration>
 <property>
 <name>dfs.replication</name>
 <value>3</value>
```

```
 </property>
</configuration>
```

[/sourcecode]

Formateando el *namenode*

Antes de empezar a ejecutar el nuevo clúster multi-nodo, es necesario formatear el *namenode*. Para ello, se ejecuta el siguiente comando en el *master*:

```
1. hduser@master:/usr/lib/hadoop-0.18$ bin/hadoop-0.18 namenode -format
```

Finalmente con el siguiente comando se arranca el sistema con el nuevo clúster donde para este caso solo se usará *bin/start-dfs.sh* en el *master*

```
1. hduser@master:/usr/lib/hadoop-0.18$ bin/start-dfs.sh
```

Para detener el sistema con el clúster es suficiente con el siguiente comando donde para este caso se usará *bin/stop-dfs.sh*:

```
1. hduser@master:/usr/lib/hadoop-0.18$ bin/stop-dfs.sh
```

2.16 Conclusiones del capítulo

Durante este capítulo, se ha llevado a cabo un análisis de cada uno de los elementos que caracterizan al sistema de archivos *Hadoop* y que lo acreditan de igual forma, como un sistema de archivos altamente tolerante ante grandes capacidades de datos e información. Esto permitió entender su funcionamiento y apreciar las potencialidades del mismo, las cuales estarían a disposición para la red social. Esta premisa, igualmente es una de las más importantes a tener en cuenta en la selección del sistema para la red social de la Universidad de las Ciencias Informáticas. Al mismo tiempo se han establecido, algunas de las características de su funcionamiento interno a la hora de ejecutar tareas, llevar a cabo transacciones, accesos simultáneos de usuarios, entre otras, que responden al proceso de ejecución del sistema para lograr y ser capaz de valorar su funcionamiento. Los elementos más importantes de su instalación también fueron dados a conocer en este capítulo, de modo que sirvió de guía para la instalación y configuración del sistema y que servirá más adelante en la ejecución de las pruebas pertinentes. Igualmente se establecen algunos parámetros de configuración a tener en cuenta para su correcto funcionamiento. Las

pruebas de este sistema de ficheros, la validación de los argumentos teóricos antes analizados y los resultados arrojados durante el proceso de pruebas, constituyen las principales fuentes de análisis a tener en cuenta en el desarrollo del próximo capítulo.

CAPÍTULO 3: RESULTADOS Y VALIDACIÓN DEL SISTEMA DE ARCHIVOS DISTRIBUIDO *HADOOP* (HDFS)

3.1 Introducción del capítulo

Todo sistema o *software* informático una vez concluida su fase de elaboración y construcción, requiere las pruebas necesarias para validar de una forma u otra su correcto funcionamiento. La prueba es un proceso de ejecución de un programa con la intención de descubrir errores. Una buena prueba es aquella que tiene una alta probabilidad de mostrar un error no descubierto hasta entonces. A lo largo de este capítulo, se pretende demostrar una vez realizadas las pruebas correspondientes, la alta posibilidad de aplicar el Sistema de Archivos Distribuido *Hadoop* (HDFS) a la red social universitaria de la Universidad de las Ciencias Informáticas. Existen diferentes tipos de pruebas que permiten corroborar el correcto funcionamiento de un *software*. Para el caso de este trabajo de diploma, se pretende aplicar las pruebas que se abarcan en las dimensiones de funcionalidad, fiabilidad, rendimiento y soportabilidad las cuales están enfocadas en verificar las habilidades de los programas para manejar grandes cantidades de datos. De esta misma forma, se enfoca a la valoración de la robustez (resistencia ante fallos), a evaluar como el sistema responde bajo condiciones anormales (extrema sobrecarga, insuficiente memoria, servicios y *hardware* no disponible, entre otras).

La distribución de *Hadoop* viene con un número de *benchmarks* (especies de programas para establecer pruebas al sistema) los cuales están empaquetados en *hadoop-*test*.jar* y *hadoop-*examples*.jar.* Los tres *benchmarks* principales de los que se estarán haciendo uso son: *TestDFSIO*, *nnbench*, *mrbench* (en *hadoop-*test*.jar*) y *randomwriter* (en *hadoop-*examples*.jar*).

Es bueno resaltar, que existen otros *benchmarks* muy importantes para llevar a cabo pruebas al sistema de ficheros (*TeraGen*/*TeraSort*/*TeraValidate*), pero solo se enfocará todo el proceso de pruebas en los *benchmarks* anteriormente mencionados.

3.2 Comprensión del proceso de pruebas

Las pruebas fueron desarrolladas en un inicio, solamente con un nodo, en un sistema de archivos local. Luego se desarrollaron con una simulación de 3 nodos en un clúster del sistema de archivos distribuidos HDFS, utilizando para ellos un factor de replicación igual a 3 donde se asumiría que, un nodo sería el *master* (*namenode* y *jobtracker*) y los otros dos nodos serían el *slave* (*datanodes* y *tasktracker*) (ver

configuración en epígrafe 2.15). Las pruebas que aquí se ejecutan están basadas fundamentalmente en los programas *benchmarks* de HDFS incluidos en el paquete de pruebas de TestDFSIO.java, con la idea de comprender cómo un usuario crea y solicita un trabajo *MapReduce* y cómo este trabajo es procesado en HDFS.

3.2.1 *TestDFSIO*

El *benchmark TestDFSIO* es una prueba de escritura y lectura para el sistema HDFS. Es útil en tareas tales como pruebas de *stress* HDFS, para descubrir la formación de cuellos de botellas en la red, para acelerar el funcionamiento de *hardware*, del sistema operativo y de las configuraciones de *Hadoop* en las máquinas del clúster (particularmente en los *namenode* y *datanodes*) y para dar la impresión de cuán rápido es el clúster en términos de entrada/salida (I/O)(Tien Duc Dinh 2009). Las funciones de este programa se pueden apreciar a través del siguiente comando:

```
/usr/lib/hadooo-o.18/hadoop-0.18 jar hadoop-0.18.3+76.2-test.jar TestDFSIO
Usage: TestDFSIO -read j -write j -clean [-nrFiles N] [-fileSize MB] [-resFile resultFile-
Name] [-bufferSize Bytes]
```

Cada tarea *MapReduce* se ejecutará en un *datanode* y la distribución de estas tareas es realizada por el *jobtracker*.

TestDFSIO está diseñado de tal forma que se usará una tarea mapa por archivo por lo que si se tienen 5 tareas mapa y la medida es igual a “X” Mb/s se tendrá en total 5*X Mb/s.

El comando para ejecutar una prueba de escritura que genere 10 archivos con un tamaño de 1GB para un total de 10 GB es el siguiente:

```
1. sudo hadoop-o.18 jar hadoop-0.18.3+76.2-test.jar TestDFSIO –write –nrFiles 10 –fileSize 1000
```

El comando para ejecutar una prueba de lectura que genere 10 archivos con un tamaño de 1GB para un total de 10 GB es el siguiente:

```
1. sudo hadoop-o.18 jar hadoop-0.18.3+76.2-test.jar TestDFSIO –read –nrFiles 10 –fileSize 1000
```

Para limpiar y borrar los datos de prueba se ejecuta el siguiente comando:

1. sudo hadoop-0.18 jar hadoop-0.18.3+76.2-test.jar TestDFSIO –clean

3.2.2 Escenarios de pruebas

Las pruebas se llevaron a cabo a través de los procesos de lectura/escritura con las capacidades de archivos de datos de 1000MB (1GB), 4000MB (4GB) y 8000MB (8GB) respectivamente.

Abreviatura: Px= Prueba X (por ej. P1= 1ra prueba)

Las medidas corresponden a una tarea por mapa. Esto significa que si se tienen 5 tareas y la medida es igual a “X” Mb/s se tendrá entonces un total de 5*X Mb/s.

Para todos los escenarios de pruebas, el proceso de escritura/lectura fue probado 3 veces y un valor medio fue comparado con el resto de las pruebas, para evitar datos atípicos. Estas pruebas se realizaron en un sistema de archivos local y en un clúster del sistema para un factor de replicación igual 3, lo que permitió establecer una comparación y determinar el rendimiento, escalabilidad y rapidez del sistema de archivos *Hadoop*.

3.3 Resultados de las pruebas

Prueba WRITE/READ (1000MB, tamaño de bloque: 64MB)

nrFiles=10, Factor de replicación=1

Tabla 3: **Resultados de las pruebas WRITE/READ para 1GB y Fr =1.**

	WRITE				READ			
	P1	**P2**	**P3**	**Promedio**	**P1**	**P2**	**P3**	**Promedio**
Throughput (mb/s)	**27.77**	**26.64**	**26.58**	**29.99**	**39.65**	**39.65**	**37.42**	**38.91**
Average IO rate (mb/s)	**28.66**	**26.66**	**26.60**	**27.31**	**39.83**	**39.83**	**37.48**	**39.05**
IO rate std deviation	**6.24**	**0.70**	**0.66**	**2.53**	**2.66**	**2.66**	**1.58**	**2.3**

Ver Anexo 1

nrFiles=10, Factor de replicación=3

Tabla 4: **Resultados de las pruebas WRITE/READ para 1GB y Fr = 3.**

	WRITE				READ			
	P1	**P2**	**P3**	**Promedio**	**P1**	**P2**	**P3**	**Promedio**
Throughput (mb/s)	**25.01**	**26.32**	**28.87**	**26.73**	**36.82**	**36.82**	**40.81**	**38.15**
Average IO rate (mb/s)	**25.08**	**26.33**	**30.44**	**27.52**	**36.84**	**36.84**	**46.15**	**39.94**
IO rate std deviation	**1.31**	**0.47**	**8.21**	**2.99**	**0.94**	**0.94**	**19.86**	**7.25**

Ver Anexo 2

Prueba WRITE/READ (4000MB, tamaño de bloque: 64MB)

nrFiles=5, Factor de replicación=1

Tabla 5: **Resultados de las pruebas WRITE/READ para 4GB y Fr = 1.**

	WRITE				READ			
	P1	**P2**	**P3**	**Promedio**	**P1**	**P2**	**P3**	**Promedio**
Throughput (mb/s)	**29.34**	**29.42**	**29.29**	**29.35**	**43.54**	**43.54**	**40.93**	**42.67**
Average IO rate (mb/s)	**31.27**	**31.32**	**31.18**	**31.26**	**48.43**	**48.43**	**45.87**	**47.58**
IO rate std deviation	**9.34**	**9.24**	**9.19**	**9.26**	**19.13**	**19.13**	**19.14**	**19.13**

Ver Anexo 3

nrFiles=5, Factor de replicación=3

Tabla 6: **Resultados de las pruebas WRITE/READ para 4GB y Fr = 3.**

	WRITE				READ			
	P1	**P2**	**P3**	**Promedio**	**P1**	**P2**	**P3**	**Promedio**
Throughput (mb/s)	**29.44**	**29.04**	**29.18**	**29.22**	**44.65**	**44.65**	**41.22**	**43.51**
Average IO rate (mb/s)	**31.47**	**30.62**	**31.09**	**31.06**	**48.87**	**48.87**	**46.13**	**47.96**
IO rate std deviation	**9.63**	**8.25**	**9.27**	**9.05**	**17.74**	**17.74**	**19.00**	**18.16**

Ver Anexo 4

Prueba WRITE/READ (8000MB, tamaño de bloque: 64MB)

nrFiles=5, Factor de replicación=1

Tabla 7: **Resultados de las pruebas WRITE/READ para 8GB y Fr = 1.**

	WRITE				READ			
	P1	**P2**	**P3**	**Promedio**	**P1**	**P2**	**P3**	**Promedio**
Throughput (mb/s)	**40.23**	**29.12**	**29.08**	**32.81**	**40.23**	**40.23**	**39.09**	**39.85**
Average IO rate (mb/s)	**44.62**	**31.09**	**30.72**	**35.48**	**44.62**	**44.62**	**43.23**	**44.16**
IO rate std deviation	**17.67**	**9.43**	**8.42**	**11.84**	**17.67**	**17.67**	**16.69**	**17.34**

Ver Anexo 5

nrFiles=5, Factor de Replicación=3

Tabla 8: **Resultados de las pruebas WRITE/READ para 8GB y Fr = 3.**

	WRITE				READ			
	P1	**P2**	**P3**	**Promedio**	**P1**	**P2**	**P3**	**Promedio**
Throughput (mb/s)	**28.32**	**28.87**	**29.18**	**28.79**	**40.81**	**40.81**	**41.08**	**40.9**
Average IO rate (mb/s)	**29.96**	**30.44**	**31.03**	**30.48**	**46.15**	**46.15**	**44.84**	**45.71**
IO rate std deviation	**8.35**	**8.21**	**9.08**	**8.55**	**19.86**	**19.86**	**16.08**	**18.06**

Ver Anexo 6

Evaluación de escritura

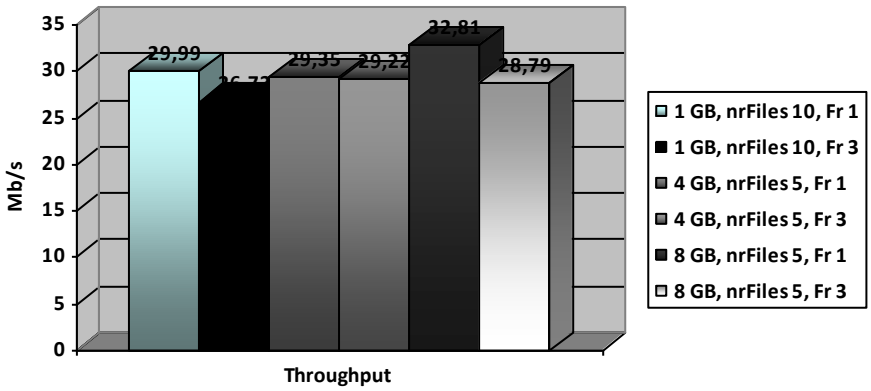


Figura 11: **Evaluación de escritura.**

La ejecución del proceso de escritura con un tamaño de bloque de 64MB, podría ser similar a otra ejecución con un tamaño diferente (128MB) en los bloques. Su escalabilidad es muy buena en ambos, tanto en los datos pequeños como en grandes tamaños de datos. El proceso de escritura con archivos replicados, lógicamente produce un rendimiento más bajo en cuanto a capacidad de los archivos expresado en *megabytes* por segundos.

Evaluación de lectura

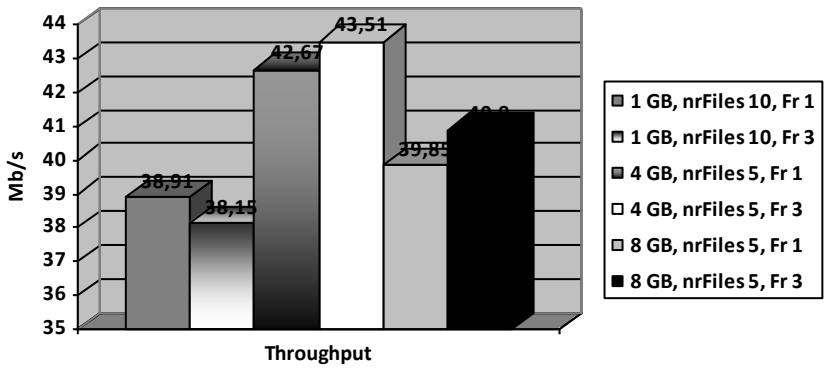


Figura 12: **Evaluación de lectura.**

La ejecución del proceso de lectura con bloques de tamaño de 64MB, pudiera ser similar de igual manera a otras ejecuciones con diferente tamaño de bloque (128MB), además de ser más rápido que el proceso de escritura, apreciándose en los valores obtenidos en cada una de las ejecuciones de prueba. La ejecución de lectura con pequeños archivos (1GB) es más rápida que con grandes tamaños de archivos (4 y 8GB). El proceso de lectura, lógicamente con archivos replicados en determinados factores, produce un rendimiento más bajo en términos de tiempo y más alta en términos de tamaño en la cantidad de datos transferidos. Por tanto se puede constatar, que el sistema de ficheros recoge en su diseño y arquitectura, la gran funcionalidad de operar bajo grandes clúster de máquinas y transferencias de datos respectivamente.

3.3.1 Comparación HDFS: sistema de archivo local VS clúster del sistema

Las pruebas realizadas a través del *benchmark TestDFSIO* y cuyos resultados estadísticos se vieron anteriormente, fueron llevadas a cabo desde diferentes puntos de referencia. Un elemento importante a tener en cuenta a la hora de realizar las pruebas *TestDFSIO* y de analizar los resultados respectivamente, es el factor de replicación HDFS, el cual juega un papel importante dentro de todo este proceso. Si se comparan dos ejecuciones de escritura con factor de replicación 2 y 3 respectivamente, se podrá apreciar un mayor rendimiento y un mayor promedio de entrada/salida en la ejecución realizada con menos factor de replicación. Esto se puede ver en las representaciones siguientes:

Tabla 9: **Comparación de los resultados de escritura para Fr = 1 y Fr = 3.**

WRITE-4GB	FR=1	FR=3
Throughput (Mb/s)	29.35	29.22
Average IO rate (Mb/s)	31.26	31.06

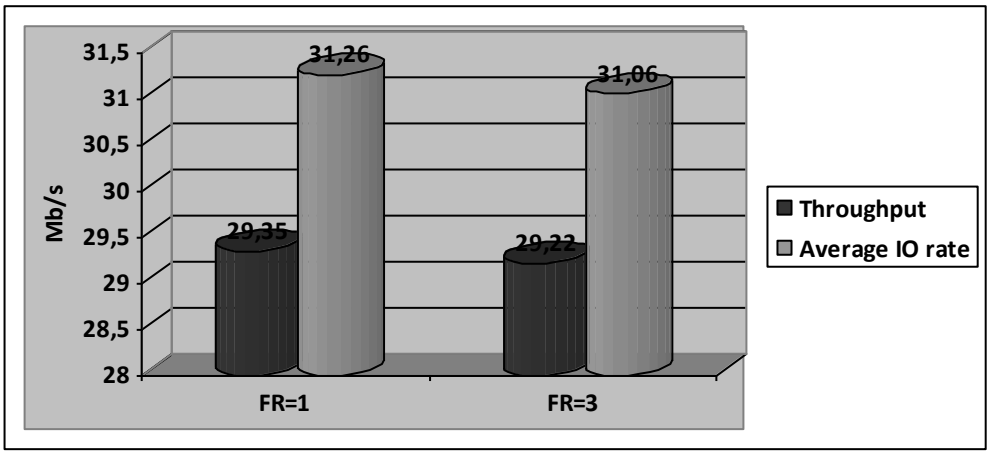


Figura 13: **Comparación de los resultados de escritura para Fr = 1 y Fr = 3.**

3.3.2 NNBENCH y MRBENCH

NNBENCH

Otro de los *benchmarks* que se utilizaron en el proceso de pruebas durante la investigación, fue el programa *NNBENCH*, el cual está incluido igualmente dentro de la paquetería de *Test.jar* del sistema

HDFS. Este *benchmark* es útil para llevar a cabo pruebas de carga sobre la configuración y el *hardware* del *namenode.* Genera una gran cantidad de solicitudes HDFS, normalmente con pequeñas "cargas útiles" con el único propósito de provocar *stress* HDFS sobre el *namenode*. Este *benchmark* puede simular solicitudes para crear, leer, renombrar y borrar archivos en el sistema HDFS. Con el siguiente comando se puede apreciar la lista de operaciones que se pueden llevar a cabo:

```
1. sudo jar hadoop-0.18 jar hadoop-0.18.3+76.2-test.jar nnbench
```

Luego el siguiente comando se ejecutó en la aplicación sobre el *namenode* donde creó 1000 archivos usando 12 *maps* y 6 *reducers.*

```
1. sudo hadoop-0.18 jar hadoop-0.18.3+76.2-test.jar nnbench –operation create_write \-maps 12 –reduces 6 –blockSize 1 –bytesToWrite 0 –numberOfFiles 100 –replicationFactorPerFile 3 –readFileAfterOpen true \-baseDir /benchmarks/NNBench-julio-System-Product-Name
```

La salida para la ejecución de este comando se puede ver en el Anexo 7

Se pueden ver los resultados de completitud de tareas almacenados en la interfaz de administración del *MapReduce* en el *localhost* de *Hadoop*. Esto muestra cómo fue ejecutándose el proceso de creación de escritura a través del *nnbench* y todas las estadísticas durante su ejecución.

Hadoop job_201205031759_0097 on localhost

User: root
Job Name: NNBench-create_write
Job File: hdfs://localhost:54310/tmp/hadoop-hadoop/mapred/system/job_201205031759_0097/job.xml
Status: Succeeded
Started at: Mon May 14 16:30:24 EDT 2012
Finished at: Mon May 14 16:34:35 EDT 2012
Finished in: 4mins, 11sec

Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	100.00%	12	0	0	12	0	0 / 0
reduce	100.00%	6	0	0	6	0	0 / 0

Figura 14: **Interfaz del proceso ejecutado con NNBENCH.**

Figura 15: **Resultado de completitud de tareas reduce.**

MRBENCH

Es una aplicación que se encuentra dentro de la paquetería del *test.jar* en el sistema de archivos HDFS. Constituye otro de los *benchmarks* utilizados para llevar a cabo pruebas de *stress* y carga en el sistema y en este caso, se realizó la prueba con el objetivo de trabajar sobre el *framework* de *MapReduce* como modelo de programación que utiliza el sistema de archivos. *MRBENCH* es de igual forma muy complementario a las "grandes escalas" del *TeraSort* debido a que este verifica si las ejecuciones de trabajos pequeños son sensibles y si está corriendo eficientemente en el clúster sobre el cual se trabaje. Esta aplicación se enfoca en la capa de *MapReduce* ya que su impacto sobre la capa de HDFS es limitada. Para ver en detalles las operaciones que se pueden realizar con esta aplicación se ejecuta el siguiente comando:

```
1. sudo hadoop-0.18 jar hadoop-0.18.3-76.2-test.jar mrbench --help
```

La pruebas llevada a cabo con este *benchmark* fue la de ejecutar una pequeña prueba de 50 trabajos sobre el sistema mediante la operación de *–numRums* concebida a través del siguiente comando.

```
1. sudo hadoop-0.18 jar hadoop-0.18.3-76.2-test.jar mrbench –numRuns 50.
```

Se fueron ejecutando 50 trabajos en el sistema con cada uno de los valores por defecto que tiene del *framework MapReduce*. Finalmente se puede ver el tiempo que tardó en la ejecución, expresado en (milisegundos) (ver Anexo 8). De igual forma si se analiza el resultado obtenido para 13.119 segundos, dicho resultado muestra el rendimiento del sistema en dicha ejecución y la escalabilidad del mismo es aún

mayor pues garantiza más prestación de ejecución con una gran cantidad de trabajos *MapReduce* sobre el sistema HDFS**.**

3.4 Conclusiones del capítulo

Hasta este momento se ha podido apreciar todo el proceso de pruebas al sistema de archivos *Hadoop*, basándose fundamentalmente en los diferentes *benchmarks* que contiene en su paquetería. Al provocar *stress* y carga sobre HDFS se pudo ver como el rendimiento es altamente apreciable con unos índices de ejecución bien bajos, principalmente en la simulación de varios nodos, no siendo así en las pruebas realizadas en un sistema de archivos local. Esto da la medida que el sistema de archivos distribuidos *Hadoop*, está fundamentado en el soporte de clúster con grandes cantidades de nodos así como el soporte de inmensas capacidades de información cubiertas en datos, archivos, información, entre otras. Se pudo concebir el correcto funcionamiento del sistema mostrado en cada una de las interfaces de trabajo además de ver cómo la escalabilidad, fiabilidad, rendimiento, rapidez y seguridad se ven implícitos en dicho sistema de archivos.

CONCLUSIONES GENERALES

Con el desarrollo de este trabajo de diploma, se da cumplimiento a cada uno de los objetivos trazados para su realización y finalmente se cuenta con la propuesta del sistema de ficheros distribuido el cual hace factible el proceso de soporte de los recursos que se gestionarán en la red social universitaria de la Universidad de las Ciencias Informáticas.

Sobre la base de los resultados principales obtenidos se puede apreciar que:

- Se llevó a cabo la inferencia en el estudio de los sistemas de archivos distribuido, permitiendo de esta forma la centralización en el objeto de estudio y la comprensión del funcionamiento de cada uno de estos sobre las diferentes tecnologías informáticas.

- Se identificaron cualidades y diferencias entre los sistemas de archivos distribuido estudiados, de modo que permitió conocer su entorno en cuanto a rendimiento, escalabilidad, fiabilidad, robustez, entre otros, y que contribuyeran a la determinación final de la propuesta de solución.

- Se estableció una fundamentación del Sistema de Ficheros Distribuido *Hadoop* (HDFS) basado en elementos de su arquitectura y diseño, lo cual facilitó la comprensión de cómo este sistema funciona y cómo puede ser aplicado para el soporte de los recursos de la red social universitaria.

- Se realizaron las pruebas correspondientes para la validación del sistema, donde se obtuvo como resultado, una correcta funcionalidad de forma general pero esencialmente a través de un clúster, una alta rapidez y rendimiento en cuanto a ejecuciones de lectura y escritura de archivos sobre el clúster, percibiéndose finalmente, la viabilidad del sistema de archivos para el soporte de los servicios de la red social universitaria.

De manera general, se pudo constatar que la propuesta de solución enmarcada en la aplicación del sistema de archivos *Hadoop*, constituye una excelente propuesta para el soporte de forma eficiente los servicios que brindará la red social universitaria de la Universidad de las Ciencias Informáticas.

RECOMENDACIONES

Tomando como punto de partida la investigación realizada y la experiencia acumulada durante la realización de este trabajo de diploma, se proponen las siguientes recomendaciones:

- Usar el sistema de ficheros distribuido en el desarrollo de la red social universitaria, como soporte de los servicios gestionados en la misma.
- Dar seguimiento al sistema de ficheros para alcanzar una funcionalidad completa en clúster con un mayor número de nodos operando.
- Investigar sobre herramientas que permitan automatizar los procesos de *backups* o salvas de información sobre el disco duro.
- Analizar la posibilidad de crear una aplicación con una interfaz gráfica que permita realizar la configuración del clúster de forma automática una vez instalado el mismo.
- Presentar el resultado en sesiones de eventos de corte científico dentro de la universidad.

BIBLIOGRAFÍA CONSULTADA

(Julio 2009). "The Apache Software Foundation, Hadoop,[Online]."

An Oak Ridge National Laboratory, L. C. o. E. P. (2008) PETA-SCALE I/O WITH THE LUSTRE™ FILE SYSTEM.

ÁNGEL MERCHAN, J. P., JUAN MORENO Implementación de un módulo de búsqueda de personas dentro de una base de datos de rostros en un ambiente distribuido usando Hadoop y los Servicios Web de Amazon (AWS). Facultad de Ingeniería en Electricidad y Computación., Escuela Superior Politécnica del litoral (ESPOL).

Barton, E. (2008). The Lustre File System.

BORTHAKUR, D. (2007). The Hadoop Distributed File System: Architecture and Design. Estados Unidos, The Apache Software Foundation**:** 3-14.

Braam, P. J. (2003). "The Coda distributed file system.": 1,2,5.

Calvo, A. M. (2004). "Google File System: el sistema de archivos de Google.".

Chandramohan A. Thekkath, T. M., Edward K. Lee. (1997). Frangipani: A scalable distributed file system. In Proceedings of the 16th ACM Symposium on Operating System Principles.**:** 224-237.

David A. Patterson, G. A. G., Randy H. Katz. (1988). A case for redundant arrays of inexpensive disks (RAID). In Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data. **:** 109-116.

Dhruba Borthakur, K. M., Karthik Ranganathan,Samuel Rash,Joydeep Sen Sarma,Nicolas Spiegelberg,Dmytro Molkov,Rodrigo Schmidt,Jonathan Gray,Hairong ,Kuang, Aravind Menon, Amitanand Aiyer (2011). Apache Hadoop Goes Realtime at Facebook.

Díaz, R. G. (2009) Sistemas de Archivos. Concepto de Sistema de Archivos. 1-6

George Coulouris, J. D., Tim Kindberg (1994) Distributed Systems. Concepts and Design.

BIBLIOGRAFÍA REFERENCIADA

GHEMAWAT, J. D. A. S. (2004). Mapreduce: simplified data processing on large clusters. In OSDI'04: Proceedings of the 6th Symposium on Opearting Systems Design & Implementation.

Granada, R. G. (2002). "Journal File Systems in Linux.", BLUMA.

J. Dean, S. G. (2008) MapReduce: Simplified data processing on large clusters. 107-113

Jacob Gorm Hansen, A. K. H. (2001). Improving Security in the Intermezzo File System, University of Copenhagen.

Jeffrey Shafer, S. R., and Alan L. Cox (2010). The Hadoop Distributed Filesystem: Balancing Portability and Performance.**:** 3-12.

Jepsen, T. C. (2003). Distributed Storage Networks: Architecture, Protocols and Management. United States, John Wiley & Sons Ltd.

John Howard, M. K., Sherri Menees, David Nichols, Mahadev Satyanarayanan, Robert Sidebotham, Michael West (2001) Scale and performance in a distributed file system. 51-81

Lafuente, A. (2010) Sistemas de Ficheros Distribuidos. 2,3,4

Luis-Felipe Cabrera , D. D. E. L. (2000). Swift: Using distributed disks triping to provide high I/O data rates. Computer Systems. **:** 405-436.

M. ZUKOWSKI, S. H. E., N. NES, AND P. BONCZ. (2007). Cooperative scans: dynamic bandwidth sharing in a DBMS. In VLDB '07: Proceedings of the 33rd international conference on Very large data bases**:** 723-734.

Mancha, C. (2002) El Sistema de Fichero Ext3. 2,3,5,6

Melamed, A. S. (1987). Performance Analysis of Unix-based Network File System. United States, AT&T Bell Laboratories.

Ray Bryant, R. F., John Hawkes (2002). Filesystem Performance and Scalability in Linux 2.4.17**:** 2,3,4,5.

Sanjay Ghemawat, H. G., and Shun-Tak Leung (2001) The Google File System.

Silberschatz, G. (2011) Sistemas de Ficheros. 3-4

Stallings, W. (2006) Sistemas Operativos Distribuidos. 3-14

Thomas Anderson, M. D., Jeanna Neefe, David Patterson, Drew Roselli, Randolph Wang (1995) Serverless networkfil e systems. In Proceedings of the 15th ACM Symposium on Operating System Principles. 109-126

Tien Duc Dinh, O. M., Julian Kunkel (2009). Hadoop Performance Evaluation. Alemania, Institute of Computer Science Research Group Parallel and Distributed Systems.

Ulf Troppens, R. E., Wolfgang Müller-Friedt, Rainer Wolafka, Nils Haustein (2009). Storage Networks Explained: Basics and Application of Fibre Channel SAN, NAS, iSCSI, InfiniBand and FcoE. J. W. S. Ltd. United Kingdom.

W. TANTISIRIROJ, S. P., AND G. GIBSON. (2008) Data-intensive file systems for internet services: A rose by any other name. Technical report.

White, T. (2009). Hadoop: The Definitive Guide. Estados Unidos, O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Zaharia, M. (2007) The Google File System.

ANEXOS

```
----- TestDFSIO ----- : read
           Date & time: Sun May 13 17:12:13 EDT 2012
       Number of files: 10
Total MBytes processed: 10000
     Throughput mb/sec: 39.64604015350947
Average IO rate mb/sec: 39.830780029296875
 IO rate std deviation: 2.6570407558179707
    Test exec time sec: 163.101
```

```
----- TestDFSIO ----- : write
           Date & time: Sun May 13 17:08:40 EDT 2012
       Number of files: 10
Total MBytes processed: 10000
     Throughput mb/sec: 26.642794722595223
Average IO rate mb/sec: 26.6610050201416
 IO rate std deviation: 0.6962446613686311
    Test exec time sec: 216.426
```

Anexo 1 **Pruebas READ/WRITE nrFiles=10, Fr=1 y tamaño=1GB.**

```
----- TestDFSIO ----- : read
           Date & time: Sun May 13 17:21:15 EDT 2012
       Number of files: 10
Total MBytes processed: 10000
     Throughput mb/sec: 36.818173450415124
Average IO rate mb/sec: 36.8420524597168
 IO rate std deviation: 0.9399507803805565
    Test exec time sec: 165.054
```

```
----- TestDFSIO ----- : write
           Date & time: Sun May 13 17:18:15 EDT 2012
       Number of files: 10
Total MBytes processed: 10000
     Throughput mb/sec: 26.31897545492349
Average IO rate mb/sec: 26.3275089263916
 IO rate std deviation: 0.47415457278775586
    Test exec time sec: 218.761
```

Anexo 2 **Pruebas READ/WRITE nrFiles=10, Fr=3 y tamaño=1GB.**

```
----- TestDFSIO ----- : read
           Date & time: Sun May 13 17:38:51 EDT 2012
       Number of files: 5
Total MBytes processed: 20000
     Throughput mb/sec: 43.538552299597484
Average IO rate mb/sec: 48.430294036865234
 IO rate std deviation: 19.13236501290812
    Test exec time sec: 275.409
```

```
----- TestDFSIO ----- : write
           Date & time: Sun May 13 17:33:31 EDT 2012
       Number of files: 5
Total MBytes processed: 20000
     Throughput mb/sec: 29.423750557212276
Average IO rate mb/sec: 31.315059661865234
 IO rate std deviation: 9.241515708614623
    Test exec time sec: 403.594
```

Anexo 3 **Pruebas READ/WRITE nrFiles=5, Fr=1 y tamaño=4GB.**

```
----- TestDFSIO ----- : write
           Date & time: Sun May 13 17:46:56 EDT 2012
       Number of files: 5
Total MBytes processed: 20000
     Throughput mb/sec: 29.041738786984656
Average IO rate mb/sec: 30.624771118164062
 IO rate std deviation: 8.246842904157052
    Test exec time sec: 415.863
```

```
----- TestDFSIO ----- : read
           Date & time: Sun May 13 17:52:17 EDT 2012
       Number of files: 5
Total MBytes processed: 20000
     Throughput mb/sec: 44.645647495825635
Average IO rate mb/sec: 48.86790084838867
 IO rate std deviation: 17.736248388721577
    Test exec time sec: 272.514
```

Anexo 4 **Pruebas READ/WRITE nrFiles=5, Fr=3 y tamaño=4GB**

```
----- TestDFSIO ----- : read
           Date & time: Sun May 13 18:16:40 EDT 2012
       Number of files: 5
Total MBytes processed: 40000
     Throughput mb/sec: 40.227891002529326
Average IO rate mb/sec: 44.618587493896484
 IO rate std deviation: 17.669726871283387
    Test exec time sec: 568.486
```

```
----- TestDFSIO ----- : write
           Date & time: Sun May 13 18:06:44 EDT 2012
       Number of files: 5
Total MBytes processed: 40000
     Throughput mb/sec: 29.122382443862968
Average IO rate mb/sec: 31.089792251586914
 IO rate std deviation: 9.433063080531756
    Test exec time sec: 796.366
```

Anexo 5 **Pruebas READ/WRITE nrFiles=5, Fr=1 y tamaño=8GB.**

```
----- TestDFSIO ----- : read
           Date & time: Sun May 13 18:46:38 EDT 2012
       Number of files: 5
Total MBytes processed: 40000
     Throughput mb/sec: 40.80862286201074
Average IO rate mb/sec: 46.1486930847168
 IO rate std deviation: 19.8594126575753
    Test exec time sec: 561.12
```

```
----- TestDFSIO ----- : write
           Date & time: Sun May 13 18:34:55 EDT 2012
       Number of files: 5
Total MBytes processed: 40000
     Throughput mb/sec: 28.865568716651246
Average IO rate mb/sec: 30.44292640686035
 IO rate std deviation: 8.208221443586512
    Test exec time sec: 808.804
```

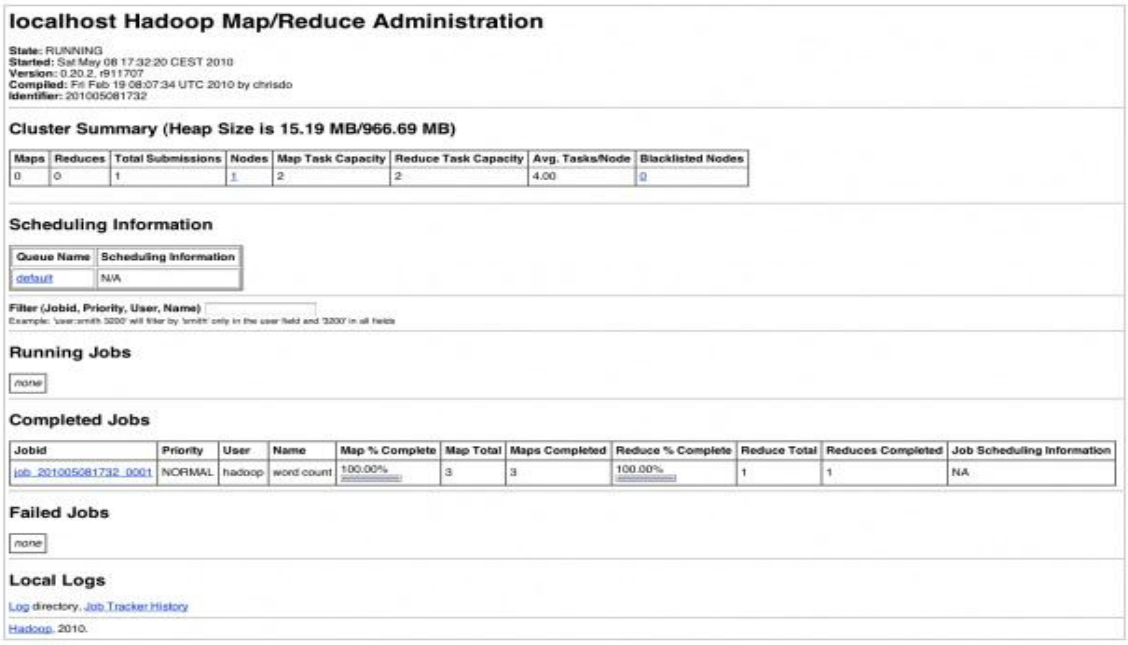
Anexo 6 **Pruebas READ/WRITE nrFiles=5, Fr=3 y tamaño=8GB.**

```
12/05/12 17:14:08 INFO mapred.JobClient:     Reduce output records=8
12/05/12 17:14:08 INFO mapred.JobClient:     Map output bytes=1859
12/05/12 17:14:08 INFO mapred.JobClient:     Map input bytes=458
12/05/12 17:14:08 INFO mapred.JobClient:     Combine input records=0
12/05/12 17:14:08 INFO mapred.JobClient:     Map output records=94
12/05/12 17:14:08 INFO mapred.JobClient:     Reduce input records=94
12/05/12 17:14:08 INFO dfs.NNBench: ------------- NNBench ------------- :
12/05/12 17:14:08 INFO dfs.NNBench:                                Version: NameNode Benchmark 0.4
12/05/12 17:14:08 INFO dfs.NNBench:                            Date & time: 2012-05-12 17:14:08,930
12/05/12 17:14:08 INFO dfs.NNBench:
12/05/12 17:14:08 INFO dfs.NNBench:                         Test Operation: create_write
12/05/12 17:14:08 INFO dfs.NNBench:                             Start time: 2012-05-12 17:11:52,255
12/05/12 17:14:08 INFO dfs.NNBench:                            Maps to run: 12
12/05/12 17:14:08 INFO dfs.NNBench:                         Reduces to run: 6
12/05/12 17:14:08 INFO dfs.NNBench:                     Block Size (bytes): 1
12/05/12 17:14:08 INFO dfs.NNBench:                         Bytes to write: 0
12/05/12 17:14:08 INFO dfs.NNBench:                     Bytes per checksum: 1
12/05/12 17:14:08 INFO dfs.NNBench:                        Number of files: 100
12/05/12 17:14:08 INFO dfs.NNBench:                     Replication factor: 3
12/05/12 17:14:08 INFO dfs.NNBench:             Successful file operations: 0
12/05/12 17:14:08 INFO dfs.NNBench:
12/05/12 17:14:08 INFO dfs.NNBench:          # maps that missed the barrier: 0
12/05/12 17:14:08 INFO dfs.NNBench:                        # exceptions: 0
12/05/12 17:14:08 INFO dfs.NNBench:
12/05/12 17:14:08 INFO dfs.NNBench:                 TPS: Create/Write/Close: 0
12/05/12 17:14:08 INFO dfs.NNBench: Avg exec time (ms): Create/Write/Close: 0.0
12/05/12 17:14:08 INFO dfs.NNBench:               Avg Lat (ms): Create/Write: Infinity
12/05/12 17:14:08 INFO dfs.NNBench:                      Avg Lat (ms): Close: NaN
12/05/12 17:14:08 INFO dfs.NNBench:
12/05/12 17:14:08 INFO dfs.NNBench:                    RAW DATA: AL Total #1: 76192
12/05/12 17:14:08 INFO dfs.NNBench:                    RAW DATA: AL Total #2: 0
12/05/12 17:14:08 INFO dfs.NNBench:                 RAW DATA: TPS Total (ms): 0
12/05/12 17:14:08 INFO dfs.NNBench:          RAW DATA: Longest Map Time (ms): 0.0
12/05/12 17:14:08 INFO dfs.NNBench:                      RAW DATA: Late maps: 0
12/05/12 17:14:08 INFO dfs.NNBench:                RAW DATA: # of exceptions: 0
12/05/12 17:14:08 INFO dfs.NNBench:
julio@julio-System-Product-Name:/usr/lib/hadoop-0.18$
```

Anexo 7 **Anexo 7 Prueba al sistema utilizando NNBENCH.**

```
12/05/12 17:33:02 INFO mapred.JobClient:     Map input bytes=2
12/05/12 17:33:02 INFO mapred.JobClient:     Combine input records=0
12/05/12 17:33:02 INFO mapred.JobClient:     Map output records=1
12/05/12 17:33:02 INFO mapred.JobClient:     Reduce input records=1
12/05/12 17:33:02 INFO mapred.MRBench: Running job 49: input=hdfs://localhost:54310/benchmarks/MRBench/mr_input output=hdfs:/
/localhost:54310/benchmarks/MRBench/mr_output/output_1442141735
12/05/12 17:33:02 WARN mapred.JobClient: Use GenericOptionsParser for parsing the arguments. Applications should implement To
ol for the same.
12/05/12 17:33:03 INFO mapred.FileInputFormat: Total input paths to process : 1
12/05/12 17:33:03 INFO mapred.FileInputFormat: Total input paths to process : 1
12/05/12 17:33:03 INFO mapred.JobClient: Running job: job_201205031759_0057
12/05/12 17:33:04 INFO mapred.JobClient:  map 0% reduce 0%
12/05/12 17:33:10 INFO mapred.JobClient:  map 100% reduce 0%
12/05/12 17:33:15 INFO mapred.JobClient: Job complete: job_201205031759_0057
12/05/12 17:33:15 INFO mapred.JobClient: Counters: 16
12/05/12 17:33:15 INFO mapred.JobClient:   File Systems
12/05/12 17:33:15 INFO mapred.JobClient:     HDFS bytes read=4
12/05/12 17:33:15 INFO mapred.JobClient:     HDFS bytes written=3
12/05/12 17:33:15 INFO mapred.JobClient:     Local bytes read=21
12/05/12 17:33:15 INFO mapred.JobClient:     Local bytes written=128
12/05/12 17:33:15 INFO mapred.JobClient:   Job Counters
12/05/12 17:33:15 INFO mapred.JobClient:     Launched reduce tasks=1
12/05/12 17:33:15 INFO mapred.JobClient:     Launched map tasks=2
12/05/12 17:33:15 INFO mapred.JobClient:     Data-local map tasks=2
12/05/12 17:33:15 INFO mapred.JobClient:   Map-Reduce Framework
12/05/12 17:33:15 INFO mapred.JobClient:     Reduce input groups=1
12/05/12 17:33:15 INFO mapred.JobClient:     Combine output records=0
12/05/12 17:33:15 INFO mapred.JobClient:     Map input records=1
12/05/12 17:33:15 INFO mapred.JobClient:     Reduce output records=1
12/05/12 17:33:15 INFO mapred.JobClient:     Map output bytes=5
12/05/12 17:33:15 INFO mapred.JobClient:     Map input bytes=2
12/05/12 17:33:15 INFO mapred.JobClient:     Combine input records=0
12/05/12 17:33:15 INFO mapred.JobClient:     Map output records=1
12/05/12 17:33:15 INFO mapred.JobClient:     Reduce input records=1
DataLines       Maps    Reduces AvgTime (milliseconds)
1               2       1       13329
julio@julio-System-Product-Name:/usr/lib/hadoop-0.18$
```

Anexo 8 **Prueba al sistema utilizando MRBENCH.**



localhost Hadoop Map/Reduce Administration

State: RUNNING
Started: Sat May 08 17:32:20 CEST 2010
Version: 0.20.2, r911707
Compiled: Fri Feb 19 08:07:34 UTC 2010 by chrisdo
Identifier: 201005081732

Cluster Summary (Heap Size is 15.19 MB/966.69 MB)

Maps	Reduces	Total Submissions	Nodes	Map Task Capacity	Reduce Task Capacity	Avg. Tasks/Node	Blacklisted Nodes
0	0	1	1	2	2	4.00	0

Scheduling Information

Queue Name	Scheduling Information
default	N/A

Filter (Jobid, Priority, User, Name)
Example: 'user:smith 3200' will filter by 'smith' only in the user field and '3200' in all fields

Running Jobs

none

Completed Jobs

Jobid	Priority	User	Name	Map % Complete	Map Total	Maps Completed	Reduce % Complete	Reduce Total	Reduces Completed	Job Scheduling Information
job_201005081732_0001	NORMAL	hadoop	word count	100.00%	3	3	100.00%	1	1	NA

Failed Jobs

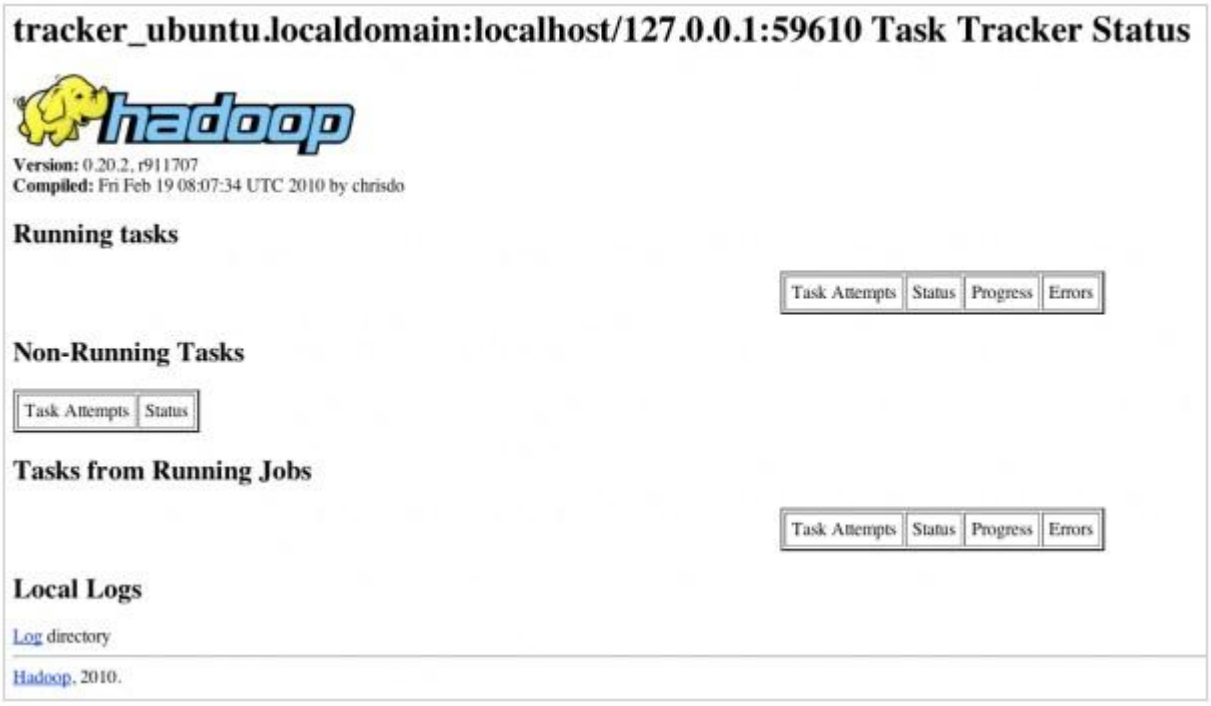
none

Local Logs

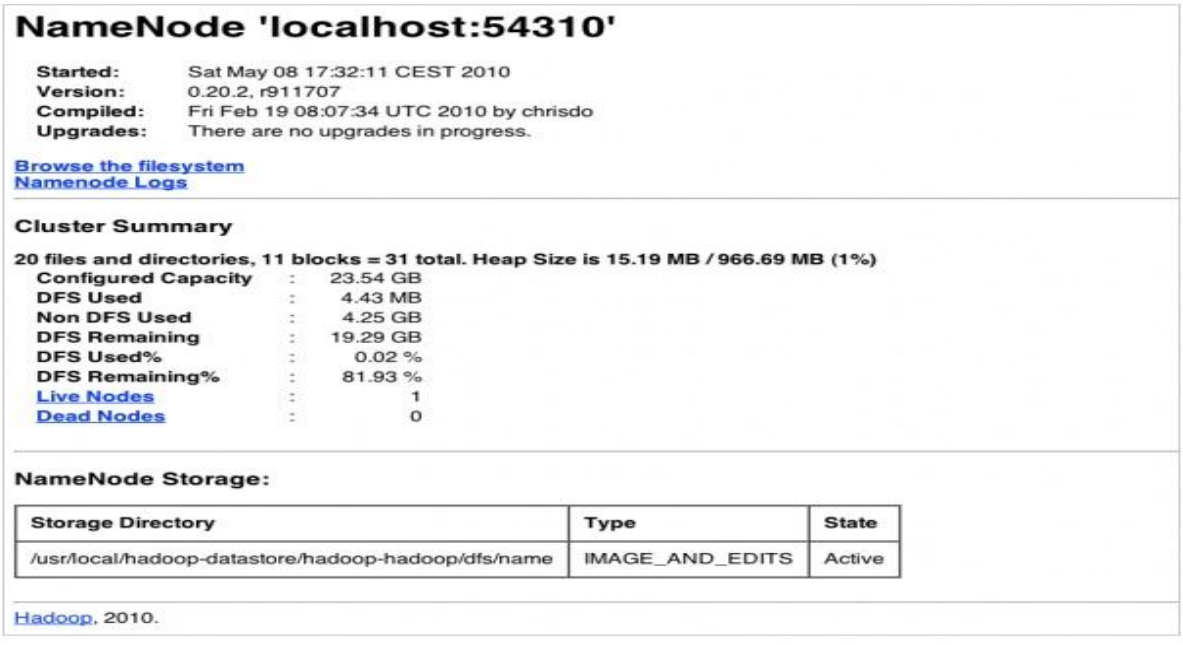
Log directory, Job Tracker History

Hadoop, 2010.

Anexo 9 **Interfaz web MapReduce jobtracker(s).**



Anexo 10 **Interfaz web tasktracker(s).**



Anexo 11 **Interfaz web namenode HDFS**