Instituto de Computación
Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay

PROYECTO DE GRADO

Cloud computing sobre entornos dinámicos tolerantes a fallos

Javier Rey, Matias Cogorno
javirey@gmail.com, matiascogorno@gmail.com

Noviembre de 2014

Tutor: Sergio Nesmachnow

Resumen

En este documento se presenta el proyecto de grado “Cloud Computing sobre entornos dinámicos tolerantes a fallos”, desarrollado en el Centro de Cálculo del Instituto de Computación de la Facultad de Ingeniería de la Universidad de la República.

Apache Hadoop [17] es un framework de código abierto para el procesamiento de grandes cantidades de datos utilizando el paradigma de computación distribuida. Está desarrollado en el lenguaje de programación Java y es mantenido por la fundación Apache. Hadoop permitiendo atacar problemas que utilizan grandes cantidades de datos a un bajo costo y con alto desempeño. Por esta razón compañías como Yahoo, Facebook, Twitter y LinkedIn utilizan Hadoop para procesar datos [15].

Hadoop posee varios mecanismos de tolerancia a fallos para recuperarse de fallas en algunos de sus nodos, pero el nodo encargado de administrar, controlar y distribuír las tareas entre los demás nodos, es un punto único de fallo, por lo que si este nodo (llamado JobTracker) falla durante la ejecución de un trabajo, este trabajo es cancelado y deberá ser comenzado nuevamene, perdiéndose todos los datos intermedios.

En este proyecto de grado, se propone desarrollar una versión de Hadoop tolerante a fallos en el JobTracker, realizando un respaldo de la información del JobTracker periódicamente. Cuando el JobTracker falla, uno de los demás nodos del sistema se transforma en el nuevo JobTracker, recuperando el último respaldo del JobTracker que falló. Para minimizar el tiempo perdido mientras el JobTracker está caído, las tareas que ya habían sido asignadas por el JobTracker y se estaban ejecutando, se siguen ejecutando mientras el JobTracker está caído.

La versión modificada de Hadoop desarrollada en este proyecto logró soportar repetidas fallas en el JobTracker durante la misma ejecución, con un overhead del 1.77 % del total del tiempo de ejecución.

Índice general

Capítulo 1

Introducción

Un sistema distribuido es un conjunto de computadoras separadas físicamente, conectadas entre sí mediante una red de comunicaciones, que son capaces de colaborar a fin de realizar una tarea. Cada máquina posee sus componentes de hardware y software, y no comparten memoria ni espacio de ejecución de programas. En este tipo de sistemas, el programador percibe todos los recursos como una sola unidad, sin saber qué recursos están en qué máquina. Se llama computación distribuida a la computación llevada a cabo sobre sistemas distribuidos.

Según el National Institute of Standards and Technology (NIST) [33], Cloud computing es un modelo que permite el acceso bajo demanda a través de la red a un conjunto compartido de recursos de computación configurables (redes, servidores, almacenamiento, aplicaciones y servicios) que pueden ser aprovisionados rápidamente y con mínimo esfuerzo de gestión o interacción del proveedor del servicio.

En los últimos años, el auge de las infraestructuras en la nube ha abierto nuevas perspectivas y oportunidades, pero las empresas todavía dudan en migrar sus aplicaciones a la nube [47] [58]. Esto se debe principalmente a que en la nube hay más problemas de seguridad y confidencialidad de los datos en comparación a clusters manejados en forma privada. Estos temores de las empresas en relación a la seguridad de los datos que depositan en la nube se vieron agravados en el 2013 a partir de las revelaciones de Edward Snowden, hecho que desencadenó en la búsqueda de alternativas a la nube tradicional por parte de múltiples organizaciones.

Una de las opciones que brinda la nube es la posibilidad de rentar servidores para llevar a cabo cálculos sobre extensas cantidades de datos, distribuyendo los cálculos en varios nodos para disminuir el tiempo de procesamiento [14]. A pesar de los beneficios que esto podría traer en

el análisis de datos, las empresas son reticentes a usar estas herramientas provistas por la nube debido a que la protección de datos sensibles disminuiría.

La alternativa a utilizar la nube para este tipo de tareas reside en infraestructuras privadas manejadas por la empresa, como un cluster. El inconveniente de esta opción es que la empresa debería incurrir en un gran gasto tanto monetario como de tiempo para tener este tipo de infraestructura funcionando correctamente. Por otro lado, en caso de que la empresa necesite más poder de procesamiento, debería actualizar el hardware, lo que añade otra inversión considerable.

Otra alternativa más eficiente sería utilizar el hardware que la empresa ya posee, como computadoras de trabajo que pueden ceder su poder de procesamiento ocioso a otras tareas. Esta alternativa tiene un costo en hardware mínimo para la empresa, debido que utiliza los recursos que ya posee. Por otro lado, al utilizarse cualquier clase de recursos (notebooks, celulares, tablets, etc) es necesario un sistema capaz de aprovechar aquellos que tiene disponibles, incluir nuevos recursos y soportar pérdidas repentinas de nodos (por ejemplo cuando se apaga una tablet o un celular abandona la red).

En este contexto, sería interesante utilizar algún framework que permita a las empresas llevar a cabo procesamientos distribuídos de forma sencilla, haciendo transparente para el usuario el manejo de los nodos que pueden aparecer y desaparecer en cualquier momento. Un framework que cumple con algunas de estas características es Hadoop. Hadoop es un framework de código abierto para el almacenamiento y procesamiento de grandes cantidades de datos. Fue desarrollado en Java y es mantenido por la fundación Apache [53]. Debido al sencillo modelo de programación de Hadoop, una vez configurado es accesible y simple utilizarlo, lo que hace que usuarios sin grandes conocimientos de programación puedan utilizar Hadoop para resolver problemas; por esta razón, Hadoop ha ganado gran popularidad en los últimos años.

Uno de los problemas que presenta Hadoop es que fue desarrollado para ser ejecutado sobre un cluster dedicado, donde los recursos son bien conocidos, relativamente invariantes y sin fallas. Hadoop implementa mecanismos de tolerancia a fallos, cómo réplica de datos, ejecución especulativa de tareas, mecanismos de heartbeat para detectar problemas en los nodos, etc [3][10]. Pero, a pesar de todos estos mecanismos, Hadoop posee un punto único de fallo (en inglés es Single Point Of Failure, por lo que se utilizará la sigla SPOF en el resto del documento) en el nodo maestro encargado de asignar tareas a los nodos esclavos, este nodo

se llama JobTracker. Si el JobTracker falla, también falla todo el sistema y se pierde todo el trabajo realizado ya que Hadoop no posee ningún mecanismo para tolerar una falla en este nodo [11]. Cabe señalar que esto no es un problema cuando Hadoop es ejecutado en un cluster dedicado (ambiente para el cual Hadoop fue diseñado) ya que la probabilidad de que falle un nodo es muy baja, pero sí es un gran problema a la hora de ejecutar Hadoop en un ambiente pervasivo donde los nodos son volátiles, incluido el nodo que contiene al JobTracker, por lo que este nodo podría fallar en cualquier momento.

El objetivo de este proyecto de grado es desarrollar una versión de Hadoop tolerante a fallos en el JobTracker. Para conseguir este objetivo, se realizarán respaldos de la información del JobTracker periódicamente. Se pretende que cuando el JobTracker falle, uno de los demás nodos del sistema se transforme en el nuevo JobTracker, recuperando el último respaldo del JobTracker que falló. Este mecanismo será implementado intentando mantener el overhead en el tiempo de ejecución lo más bajo posible. Para minimizar el tiempo perdido mientras el JobTracker está caído, las tareas que ya habían sido asignadas por el JobTracker y se estaban ejecutando, se siguen ejecutando mientras el JobTracker está caído. Cuando el nuevo JobTracker se integra al sistema, los nodos que estaban realizando tareas le informan al nuevo JobTracker el estado actualizado de las tareas y el sistema continúa con la ejecución como si nunca se hubiese producido una falla.

A lo largo del proyecto se presentó el problema de cómo verificar la solución tanto durante el desarrollo como en la etapa de análisis experimental, siendo necesario hacer fallar nodos en el momento deseado. Por esta razón se desarrolló Docker-Hadoop, una herramienta que permite empaquetar una distribución de Hadoop dentro del ambiente virtual de Docker [25], permitiendo así testear y validar las implementaciones de forma sencilla pudiendo simular varios escenarios y situaciones de falla. Docker-Hadoop no sólo facilitó la tarea de desarrollo y testing en el contexto de este proyecto, sino que también puede ayudar a otros investigadores que estén trabajando con Hadoop a desarrollar y testear sus propias implementaciones de Hadoop.

Este proyecto de grado se enmarca dentro del proyecto PERMARE [46], llevado a cabo por las universidades Universidade Federal de Santa Maria (Brasil), (Université Paris 1 – Panthéon Sorbonne (Francia), Université de Reims Champagne-Ardenne (Francia) y la Universidad de la República (Uruguay). El proyecto PERMARE tiene como objetivo adaptar el paradigma de MapReduce [12] a redes pervasivas y desktop grids.

Para lograr este objetivo se decidió tomar como base una implementación conocida de MapReduce y adaptarla para poder ser ejecutada en este tipo de redes. MapReduce es un modelo de programación para procesar y generar grandes cantidades de datos que facilita la paralelización del procesamiento y la implementación de mecanismos de tolerancia a fallos en los nodos. La implementación de MapReduce seleccionada para el proyecto PERMARE fue Hadoop, implementación en la que, como se mencionó anteriormente, se enfoca este proyecto de grado.

La contribución principal de este trabajo es una versión de Hadoop tolerante a fallos en el JobTracker que pueda ser utilizada en el contexto del proyecto PERMARE. También se contribuyó al proyecto PERMARE y a la comunidad de Hadoop en general mediante el desarrollo de Docker-Hadoop, herramienta que facilita el desarrollo y testeo de Hadoop. Por otro lado, este proyecto contribuyó con el desarrollo de dos papers: *MapReduce challenges on pervasive grids* [45] y *Efficient Prototyping of Fault Tolerant Map-Reduce Applications with Docker-Hadoop* [40].

El presente documento se divide en seis capítulos: la introducción, presentando brevemente la temática, problemas, soluciones y aportes de este proyecto; el segundo capítulo describe la problemática central del proyecto; el tercer capítulo introduce algunos trabajos relacionados con este proyecto realizados por otros investigadores; el siguiente capítulo describe la solución implementada y las herramientas utilizadas para lograrla; en el quinto capítulo se presenta el análisis experimental y los problemas que hubo que superar para realizar las pruebas y, finalmente, en el sexto capítulo se presentan las conclusiones de este proyecto de grado y los trabajos pendientes para el futuro.

Capítulo 2

Cloud computing y MapReduce

Este capítulo presenta una descripción de cloud computing, computación distribuida, MapReduce y Hadoop para finalmente describir en qué consiste el proyecto PERMARE.

2.1. Computación Distribuida

La computación distribuida es un modelo para resolver problemas de computación masiva utilizando varias computadoras o nodos interactuando entre sí mediante una red de comunicaciones. Este tipo de computación tiene usos variados que van desde la realización de cálculos científicos hasta aplicaciones punto a punto o P2P (peer-to-peer en inglés). Este último concepto refiere a una red donde todos los nodos son considerados iguales comunicándose entre sí de forma independiente.

En general, el objetivo de un sistema distribuido es la resolución de un problema computacional de gran escala; pero también se puede utilizar para coordinar recursos compartidos o para proveer servicios de comunicación entre usuarios.

Algunas características de los sistemas distribuidos son [30] [35]:

- Existen varias entidades computacionales autónomas, cada una contiene su propia memoria local.
- Las entidades se comunican entre sí mediante el envío de mensajes.
- El sistema tolera fallas en nodos individuales.

- La estructura del sistema (topología de la red, latencia de la red, número de nodos) no es conocida de antemano, el sistema puede consistir en diferentes tipos de nodos y puede cambiar durante la ejecución del programa distribuido.

- Cada entidad tiene una visión limitada e incompleta del sistema.

Existen dos arquitecturas principales de computación distribuida [29]:

- Sistema Cliente-Servidor: Consiste en un solo servidor que provee un servicio, y hay muchos clientes que se comunican con el servidor para consumir sus productos. En esta arquitectura, los clientes y el servidor tienen distintos trabajos: el servidor tiene que responder a los pedidos de los clientes, sin realizar nunca un pedido, mientras que el trabajo de los clientes es usar los datos obtenidos como respuesta para llevar a cabo alguna tarea. Por otro lado, el servidor no necesita conocer a los clientes, pero estos últimos sí deben conocer al servidor para poder consumir sus productos. Un ejemplo de este tipo de arquitectura sería un servidor web, donde los clientes realizan pedidos de páginas web a un servidor que se encarga de responderles con las páginas pedidas.

- Sistema P2P: Es un sistema descentralizado donde los cálculos pueden ser llevados a cabo por cualquier nodo de la red. En este tipo de sistemas, el trabajo es dividido entre todos los componentes del sistema. Todas los nodos envían y reciben datos y contribuyen con el procesamiento y memoria. En algunos casos se puede usar un nodo central que ayuda a los demás nodos a encontrarse, este nodo puede mejorar sensiblemente la eficiencia del sistema, pero no es estrictamente necesario ya que los nodos pueden encontrarse los unos a los otros analizando a sus vecinos. Ejemplos de este tipo de arquitectura pueden encontrarse en las aplicaciones de intercambio de archivos (como Napster), donde los archivos son intercambiados entre cualquier tipo de nodos.

A partir de los ejemplos dados para estos dos tipos de arquitectura de computación distribuida, se puede decir que en cloud computing se utilizan varios tipos de arquitectura distribuida. A continuación se procederá a definir cloud computing y a detallar algunas de sus características.

2.2. Cloud Computing

Según el National Institute of Standards and Technology (NIST) [33], Cloud computing es un modelo que permite el acceso bajo demanda a través de la red a un conjunto compartido de recursos de computación configurables (redes, servidores, almacenamiento, aplicaciones y servicios) que pueden ser aprovisionados rápidamente y con mínimo esfuerzo de gestión o interacción del proveedor del servicio.

En este modelo, las computadoras locales no tienen que realizar todo el trabajo pesado al ejecutar aplicaciones, la red de computadores que forman la nube se encarga de ello. De esta forma, el hardware y software que el usuario necesita disminuye, lo único que necesita la computadora del usuario es poder ejecutar la interfaz del sistema de cloud computing, que puede ser simplemente un navegador web.

Uno de los ejemplos clásicos de cloud computing es un servicio de email basado en la web, como pueden ser Yahoo!, Hotmail o Gmail. Pero un sistema de cloud computing podría manejar cualquier tipo de programa, desde un procesador de textos hasta juegos de video.

El término cloud computing proviene del símbolo utilizado por ingenieros en los diagramas de red para representar la parte desconocida de la red. Luego se popularizó el término para referirse a software, plataformas e infraestructuras que son vendidos como un servicio, por ejemplo a través de internet.

El modelo de Cloud Computing no es algo nuevo, el origen del concepto de cloud computing se remonta a la década del 50, cuándo las grandes computadoras centrales eran vistas como el futuro de la computación y comenzaron a estar disponibles en academias y corporaciones. Estas computadoras centrales eran accedidas por pequeñas computadoras clientes o terminales que eran usadas para la comunicación pero no tenían capacidad de procesamiento. Para mejorar la eficiencia de los computadores centrales, se desarrollaron técnicas que permitían a los usuarios compartir tanto el acceso a la computadora central desde múltiples terminales como también el tiempo de CPU. De este modo se eliminaban los períodos de inactividad en la computadora central.

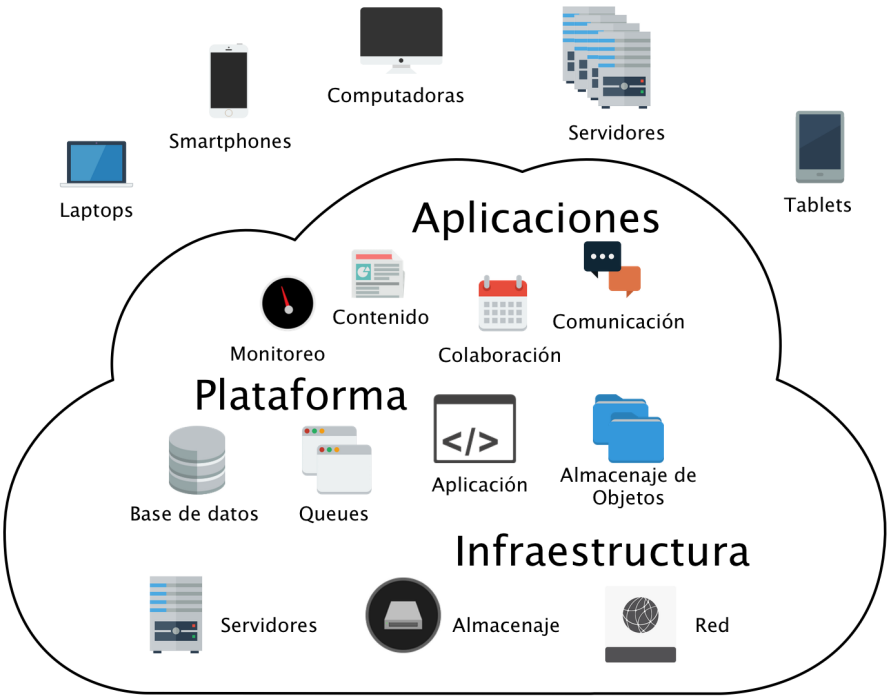

Figura 2.1: Esquema del cloud

En la actualidad, el modelo de Cloud Computing también busca maximizar la eficiencia de los recursos. Por ejemplo, una computadora en la nube (o cloud) que se utiliza para alojar un servicio utilizado por usuarios europeos durante las horas de negocio (por ejemplo correo electrónico) puede alojar más tarde otro servicio utilizado por usuarios americanos en su horario de negocio. De esta forma se utiliza un solo recurso físico continuamente en lugar de usar dos recursos pasando la mitad del tiempo sin ser utilizados. Esto tiene tanto beneficios económicos ya que se necesita menos hardware, así como también beneficios ambientales.

2.2.1. Modelos de Servicio

Existen tres modelos de servicios fundamentales que pueden ser provistos mediante Cloud Computing. Estos servicios están representados en la figura 2.2 y son descritos a continuación.

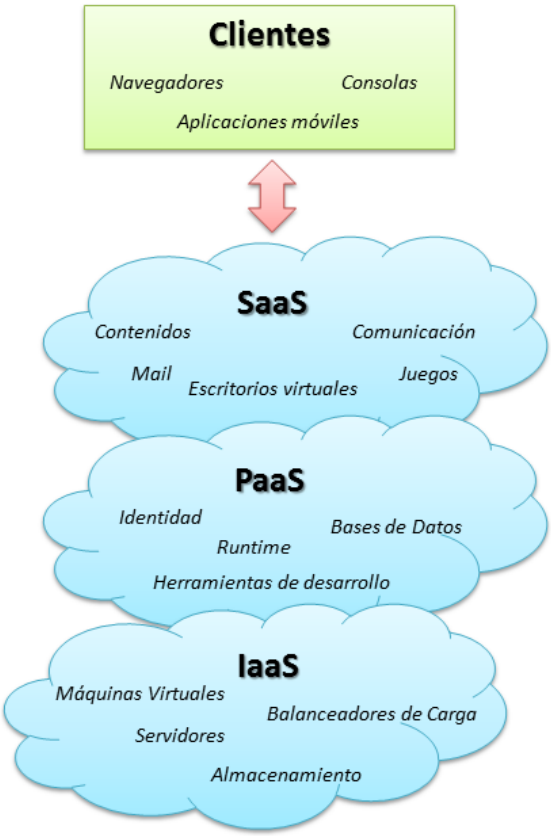

Figura 2.2: Esquema de la nube y modelos de servicio

Infraestructura como servicio (Infrastructure as a Service (IaaS))

En este modelo de servicio, se provee acceso a recursos computacionales en un ambiente virtualizado. Se ofrece espacio virtual, conexiones a la red, ancho de banda, direcciones IP y balanceadores de carga. Físicamente, los recursos de hardware son varios servidores generalmente distribuidos en varios data centers mantenidos por el proveedor del servicio. Este modelo ofrece varias ventajas para el usuario [4][8]:

- No es necesario invertir en hardware ya que es mantenido por el proveedor del servicio, ahorrando dinero y tiempo al cliente.

- El cliente no queda atado a una cantidad de recursos, debido a que si los requerimientos de hardware crecen, se puede contratar más capacidad al proveedor. Esto permite al cliente crecer fácilmente cuando lo necesita y pagar únicamente por los recursos que utiliza.

- Como se utilizan servidores virtuales albergados en una red de servidores físicos, el cliente se ve beneficiado por la redundancia.
- El servicio puede ser accedido desde cualquier lugar.
- No hay SPOF, si un servidor falla, el servicio no se ve afectado gracias al hardware restante y la redundancia.

Plataforma como servicio (Platform as a Service (PaaS))

En el modelo de Platform as a Service (plataforma como servicio o PaaS por sus siglas en inglés), el proveedor ofrece una plataforma de servicios que generalmente incluye sistema operativo, ambiente para la ejecución de un lenguaje de programación, base de datos y servidor web. Los desarrolladores pueden crear y ejecutar sus soluciones en una plataforma web sin la necesidad de preocuparse por manejar las capas de hardware y software. Algunas ofertas de PaaS, cómo Microsoft Azure, Google App Engine y Heroku, ofrecen recursos de cómputo y almacenamiento que escalan automáticamente para ajustarse a la demanda de la aplicación.

Algunos beneficios de este tipo de servicio consisten en [6]:

- El cliente no tiene que comprar el hardware ni contratar expertos que lo manejen, de forma que se puede centrar en el desarrollo de sus aplicaciones. Añadiendo que el cliente sólo renta los recursos que necesita.
- El cliente puede tener control sobre las herramientas instaladas en su plataforma y ajustarla a sus requerimientos específicos.
- Cómo los servicios de PaaS sólo necesitan una conexión a internet y un web browser, desarrolladores en distintos lugares pueden trabajar juntos en la misma aplicación.
- Se incluyen servicios de seguridad de datos, backup y recuperación.

Software como servicio (Software as a Service (SaaS))

En el modelo de software como servicio (o SaaS por sus siglas en inglés) el proveedor instala y opera el software en la nube y los usuarios acceden al software a través de clientes de la nube (por ejemplo web browsers). Ejemplos de este modelo de servicio son Gmail, Twitter y Facebook.

Los clientes no manejan ni la infraestructura ni la plataforma donde se ejecutan las aplicaciones. De esta forma, los usuarios no deben instalar las aplicaciones en sus computadoras, necesitando únicamente una conexión a internet.

Este modelo presenta varios beneficios para los usuarios [16] [51]:

- No es necesario cambiar el hardware para ejecutar el software, el poder de procesamiento que requiere la aplicación es suministrado por el cloud provider.
- No requiere instalación.
- Sólo se paga por lo que se usa, si se necesita una aplicación por un cierto período de tiempo, el usuario sólo paga por ese tiempo.
- Es escalable, si el usuario precisa más servicios o espacio de almacenamiento, puede acceder a estos recursos sin necesidad de instalar nuevo hardware o software.
- Cuando el proveedor realiza una actualización del software, el cliente automáticamente accede a la nueva versión, sin necesidad de instalar nada.
- No depende del dispositivo con el que se acceda. Al ser servicios que se acceden a través de internet, pueden ser utilizados desde cualquier dispositivo con acceso.
- El servicio puede ser accedido desde cualquier lugar con conexión a Internet.

2.3. MapReduce

MapReduce es un modelo de programación para procesar y generar grandes cantidades de datos, propuesto por J. Dean y S. Ghemawat (Google) en el 2004 [12]. Este modelo de programación consta principalmente de dos conceptos tomados de programación funcional:

- Map: En esta etapa, se aplica a todos los datos una función Map definida por el usuario, que retorna un conjunto intermedio de pares clave/valor.
- Reduce: A partir de una función definida por el usuario, se combinan todos los resultados intermedios asociados a la misma clave produciendo el resultado final.

En principio este modelo no parecería tener ningún beneficio en relación a otros modelos ya que, en general, una implementación de MapReduce al ejecutase en un solo hilo de ejecución sería más lenta que una implementación tradicional que resolviese el mismo problema. Los grandes beneficios de MapReduce son su paralelización y tolerancia a fallos.

La función Map es aplicada interactivamente sobre todos los datos, por lo que estos datos pueden dividirse para que se les aplique la función Map en paralelo. Lo mismo sucede con la función Reduce, que puede aplicarse a los pares clave/valor independientemente sobre los recursos disponibles.

Por otro lado, como el problema original es dividido en múltiples partes, si alguna falla no es necesario comenzar el proceso de cero, basta con ejecutar nuevamente sobre el conjunto de datos que falló.

Un ejemplo clásico de MapReduce es el WordCount que cuenta la cantidad de ocurrencias de cada palabra en un texto. Sí el WordCount fuese a ser ejecutado en un único nodo, una implementación podría consistir en un bucle que recorra todas las palabras y un HashList<k,v>, donde la clave sería la palabra y el valor un entero que se iría incrementando cada vez que la palabra es encontrada.

En un ambiente distribuido, el procedimiento es similar: primero cada proceso recorre una parte del archivo generando resultados parciales, luego se suman los valores de cada clave. En este caso, el Map recibirá un conjunto de palabras y retornará un conjunto de pares <w, #palabras>. Luego, el Reduce recibiría este conjunto de pares y retornaría un conjunto de <palabra, N> donde N es la cantidad de repeticiones de la palabra. En la Figura 2.3 se aprecia el flujo para este ejemplo.

Utilizando MapReduce, se pueden procesar grandes cantidades de datos de forma paralela, reduciendo significativamente el tiempo de procesamiento. Esto se logra al poder dividir los datos y analizarlos en distintos procesos que ejecutan de forma paralela. Otra ventaja es que si un proceso falla, no es necesario reejecutar todos los procesos, basta con reejecutar el proceso que falló, aumentando así la eficiencia del sistema.

2.4. Hadoop

Apache Hadoop [17] es un framework de código abierto para el procesamiento de grandes cantidades de datos utilizando el paradigma de computación distribuida. Está desarrollado en el lenguaje de programación Java y es mantenido por la fundación Apache. Hadoop utiliza Ma-

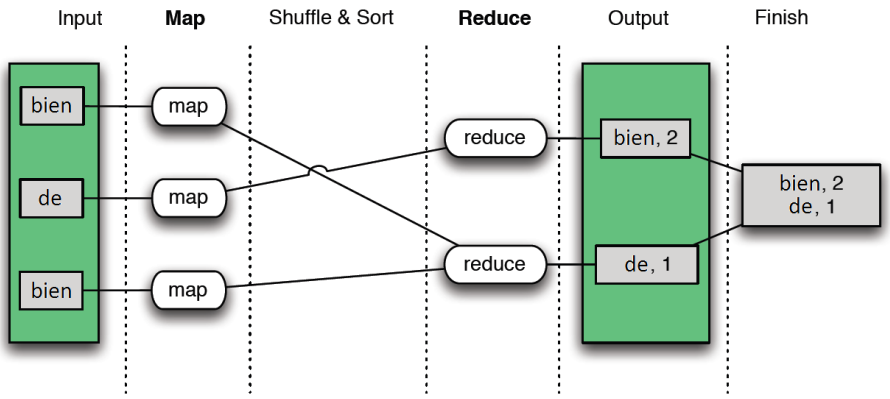

Figura 2.3: Etapas de MapReduce [44]

pReduce para procesar los datos divididos entre los nodos disponibles, permitiendo atacar problemas que utilizan grandes cantidades de datos a un bajo costo y con alto desempeño. Por esta razón compañías como Yahoo, Facebook, Twitter y LinkedIn utilizan Hadoop para procesar datos [15].

Hace ya varios años Hadoop ha alcanzado gran popularidad debido al sencillo modelo de programación que utiliza: una vez configurado y definidas las funciones de Map y Reduce, se puede comenzar a utilizarlo de forma eficiente. Esta sencillez permite que usuarios sin grandes conocimientos de programación puedan utilizar Hadoop para resolver sus problemas.

La simplicidad de Hadoop, además de la necesidad de las organizaciones de recurrir a grandes volúmenes de datos para la toma de decisiones y la baja de los costos de cloud computing, han contribuído al crecimiento de Hadoop como plataforma de procesamiento en el cloud.

Existen otras alternativas a Hadoop, por ejemplo utilizar un sistema basado en una biblioteca para el diseño y programación de aplicaciones paralelas y distribuidas como Message Passing Interface (MPI) [2], pero requieren que la sincronización y coordinación entre los nodos sea programada por el usuario. Hadoop, por otro lado, se encarga de dividir los datos, programar tareas, monitorearlas y reejecutar las que fallan, brindando una intensa ventaja en comparación con las demás alternativas para esta clase de tareas.

Hadoop se compone principalmente de dos sistemas con arquitectura maestro-esclavo (master-slave en inglés) (ver figura 2.4): el framework de MapReduce (MRFW) y el Hadoop Distributed File System (HDFS).

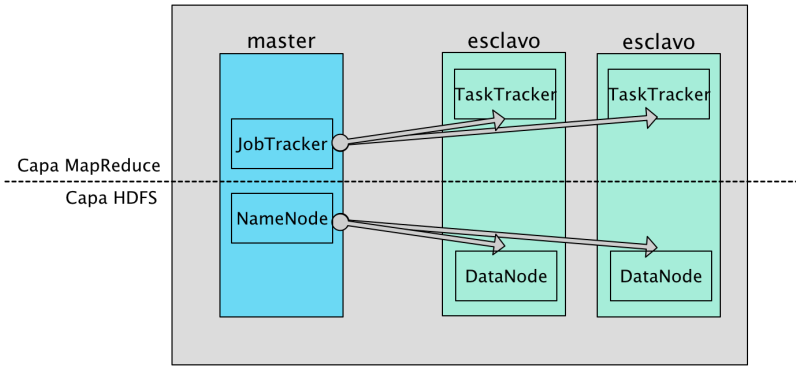

Figura 2.4: Capas principales de Hadoop

El primero es el encargado de procesar los datos, mientras que el HDFS fue diseñado para almacenar datos en grandes cantidades de hardware distribuido de bajo costo. En las siguientes secciones se presenta una introducción a ambos frameworks que componen Hadoop.

2.4.1. Hadoop Distributed File System

El Hadoop Distributed File System (HDFS) es un sistema de archivos que maneja bloques: los archivos individuales son divididos en bloques de tamaño fijo. Cada bloque es almacenado en uno o más nodos llamados DataNodes [56]. A diferencia de un sistema de archivos corriente, los bloques están dotados de un tamaño por defecto mínimo de 64MB. Dependiendo del tamaño de los datos guardados puede ser recomendable aumentar este valor. Este método de manejo de datos permite almacenar archivos que ocupen más espacio que el disponible en un único nodo, guardando los distintos bloques de un mismo archivo entre varios nodos.

Para la prevención de pérdida de bloques se utiliza un mecanismo de replicación: los bloques son replicados en más de un nodo, de forma que si uno falla, los bloques que almacena no se pierden. Más adelante en la sección 4.1.1 se detalla sobre este mecanismo.

Para mantener la sincronización del sistema de archivos, el nodo maestro, llamado NameNode, almacena la metadata (datos sobre los datos) del sistema de archivos. El NameNode mantiene los nombres de los archivos, los permisos y la ubicación de cada bloque de cada archivo. Cuando un cliente quiere leer un archivo, el NameNode le envía una lista con los DataNodes que almacenan los bloques del archivo y el cliente consume

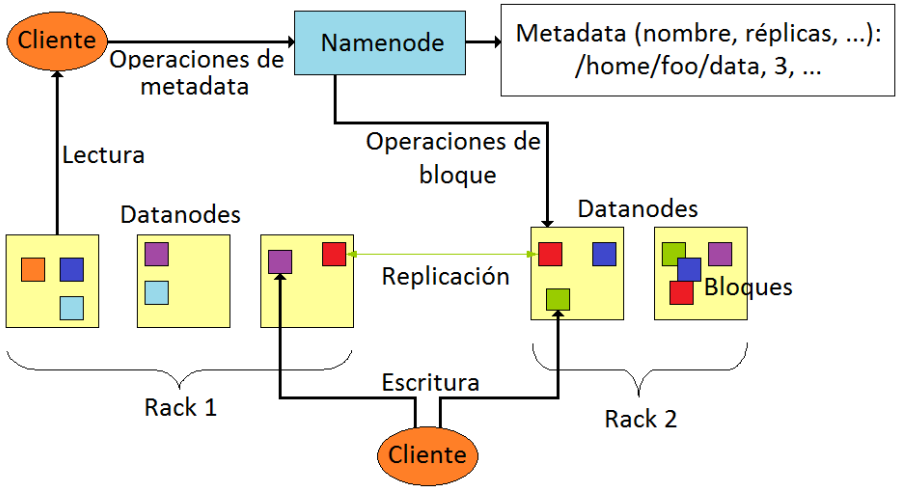

Figura 2.5: Arquitectura del HDFS (adaptado de [3])

los datos que necesita directamente de los DataNodes. De esta forma, el NameNode no está involucrado directamente en la transferencia de datos, distribuyendo la carga.

´PUno de los mayores problemas de esta arquitectura es la escalabilidad del NameNode [15]. Como el servicio NameNode se encuentra en solamente un nodo, al tener la metadata del sistema almacenada en memoria el tamaño del sistema de archivos se encuentra acotado por la capacidad del nodo donde se ejecuta.

2.4.2. MapReduce Framework

Este sistema, al igual que el HDFS, contiene una arquitectura maestro-esclavo. Cuando un cliente inicia un trabajo (o Job en inglés) en Hadoop, el pedido es manejado por el JobTracker, que se encarga de la coordinación y programación de las tareas de Map y Reduce en los diferentes nodos esclavos llamados TaskTrackers (ver figura 2.6). Cada TaskTracker ejecuta las tareas que le fueron asignadas y notifica al JobTracker cuando termina [45].

En todo momento el JobTracker conoce el estado de las tareas ya que cada cierto tiempo cada TaskTracker le notifica el estado de las tareas que está realizando y si tiene disponibilidad para realizar más tareas (este mecanismo será detallado posteriormente).

Como se mencionó anteriormente, una de las ventajas de usar el paradigma MapReduce es que cada tarea de Map y Reduce se puede ejecutar de manera aislada y, si existe alguna falla, puede ser reejecutada sin ne-

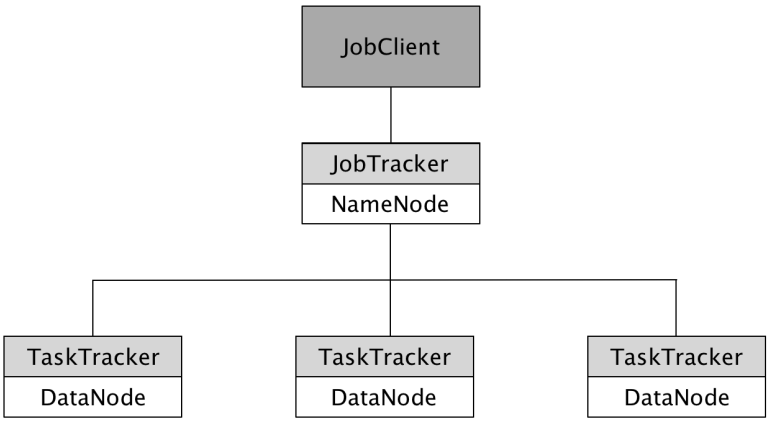

Figura 2.6: Estructura del MRFW

cesidad de comenzar de cero con todo el proceso. Este es el trabajo del JobTracker: cuando detecta que alguna tarea falló o un TaskTracker deja de responder, el JobTracker sabe exactamente qué tareas estaba realizando el TaskTracker y las reasigna a otros TaskTrackers, de esta forma no hay que iniciar de cero todas las tareas. El JobTracker, incluso puede lanzar tareas en más de un nodo especulativamente cuando sospecha que alguno puede fallar y posee los recursos suficientes.

2.4.3. Versiones

Apache Hadoop tiene (en agosto de 2014) tres diferentes ramas de versiones de las cuales dos tienen versiones estables.

Versión 1.X
La versión estable de Hadoop con la arquitectura original.

Versión 2.X
Actualmente cuenta con la versión 2.2.X estable desde Octubre 2013 y la versión 2.4 actualmente no estable en estado beta.

Versión 0.2X
Similar a la rama 2.X manteniendo con el versionado original y no apto para producción.

2.5. El Proyecto PERMARE

PERMARE (PERvasive MApREduce) es un proyecto de investigación en el marco del programa STIC-AmSud de cooperación internacional entre Francia y Latinoamérica. El proyecto es llevado a cabo en conjunto entre las universidades Universidade Federal de Santa Maria (Brasil), Université Paris 1 – Panthéon Sorbonne (Francia), Université de Reims Champagne-Ardenne (Francia) y la Universidad de la República. El principal objetivo del proyecto PERMARE es adaptar el modelo de MapReduce a ambientes *pervasivos* y grids voluntarios (desktop grids).

Una pervasive grid es un tipo de infraestructura de gran escala con características específicas de volatilidad, confianza, conectividad, seguridad, etc. Un desktop grid consiste un tipo particular de pervasive grids, dónde los recursos pertenecen a una organización o entidades independientes (como personas individuales con sus computadoras personales) y estos ceden los recursos que no usan (ciclos de procesador, almacenamiento) para cálculos del grid.

Uno de los problemas que presenta Hadoop (la implementación más conocida de MapReduce), es que requiere un ambiente altamente estructurado: el administrador debe definir manualmente las características (como número de procesadores de cada máquina) de los recursos donde se ejecutará Hadoop. Por otro lado, la instalación requiere máquinas estables y privilegios de administrador. Todo esto hace prácticamente imposible para un usuario ejecutar MapReduce sobre recursos poco utilizados que tenga disponibles, por ejemplo en una empresa o una red voluntaria donde los nodos pueden unirse o abandonar la red dinámicamente.

Se ha intentado desarrollar frameworks P2P para resolver el problema de ejecución de Hadoop sobre infraestructuras no dedicadas [32][49][50], pero los frameworks desarrollados tienen sus propias APIs que no son compatibles con aplicaciones escritas para Hadoop. Por este motivo, una de los principales ambiciones del proyecto PERMARE consiste justamente en desarrollar una API que sea compatible con Hadoop.

Existen algunos trabajos [48] en los que se ha intentado abordar los inconvenientes de esta clase de ambientes, problemas como la heterogeneidad, la volatilidad de los recursos y la tolerancia a fallos con un gran impacto en el desempeño de la red, pero el proyecto PERMARE innova principalmente en lo siguientes dos aspectos:

- Intenta usar una implementación de MapReduce conocida y ampliamente utilizada como Hadoop, e incluir elementos dependientes

del contexto de ejecución (context-aware) que faciliten la utilización en pervasive y desktop grids.

- Desarrollo un ambiente P2P distribuido compatible con la API de Hadoop.

El primer punto busca abarcar los problemas relacionados con el context-awareness y la planificación (scheduling) de tareas, mientras que el segundo punto trata con los problemas de almacenamiento de datos y compatibilidad de código. En conjunto, estos dos puntos permiten proveer una visión más amplia del problema, permitiendo llegar a soluciones más eficientes.

Un punto importante es el objetivo de mantener la compatibilidad con la API de Hadoop, permitiendo así usar aplicaciones existentes, y a su vez evaluar el desempeño y problemas de tolerancia a fallos relacionados a una pervasive grid.

2.5.1. Usos de PERMARE

En los últimos tiempos las infraestructuras en la nube han inaugurado nuevas posibilidades. Aún así las empresas continúan prefiriendo confiar en sus recursos internos sin hacer uso de la nube (utilizando un modelo de desktop grid) para desarrollar sus aplicaciones. Un aspecto importante de este tipo de soluciones en la nube es la dudosa seguridad de la integridad y confidencialidad de los datos, ya que estos son almacenados en una infraestructura que la empresa no puede controlar y esta infraestructura es manejada por desconocidos. A partir de las revelaciones de Edward Snowden en el 2013, numerosas organizaciones se orientaron hacia la búsqueda de alternativas al cloud computing tradicional [9] [26].

La solución planteada por el proyecto PERMARE propone brindar la posibilidad a las empresas de hacer uso de MapReduce en ambientes pervasivos, otra funcionalidad que la implementación actual de Hadoop no permite. Asimismo, también se podría trabajar sobre ambientes híbridos que usen una desktop grid y recursos de la nube.

Por otro lado, la solución desarrollada puede ser utilizada con fines de investigación. Sería posible invertir los recursos ociosos en una universidad, para atacar problemas que requieran grandes cantidades de procesamiento. El paradigma de trabajo propuesto permitiría que centros de investigación con limitaciones respecto a recursos económicos puedan llevar a adelante esta clase de investigaciones.

Capítulo 3

Trabajos relacionados

Como se describirá en la sección 4.1, Hadoop posee varios mecanismos para la tolerancia a fallos, que usualmente resultan suficientes para los sistemas computacionales dedicados donde ejecuta Hadoop [12]. Por este motivo, los trabajos relacionados con los mecanismos de tolerancia a fallas de Hadoop son escasos, y tratan, generalmente, sobre mejorar el desempeño o recuperarse de errores de software usando recursos computacionales adicionales.

Una de las primeras investigaciones sobre la tolerancia a fallos de MapReduce en sistemas en la nube fue realizada por Zheng [57], que propuso un mecanismo basado en réplicas pasivas de las tareas por encima del esquema estándar de reejecución brindado por Amazon Elastic MapReduce. Amazon Elastic MapReduce es un servicio de Amazon que simplifica el uso de Hadoop, facilitando el proceso de crear nodos dinámicamente. Para cada tarea, se almacenan k backups además de la tarea que se ejecuta; si la tarea falla, una de las réplicas es ejecutada. Zheng propone heurísticas para agendar backups, mover instancias de backups y seleccionar el backup a ejecutar en caso de falla para reponerse rápidamente. El artículo reporta un análisis experimental ejecutando aplicaciones MapReduce de mediano porte, en el que se obtuvieron mejoras significativas en las métricas de tolerancia a fallos utilizadas. El trabajo realizado por este investigador puede resultar interesante desde el punto de vista del proyecto PERMARE, debido a que la idea del proyecto es justamente ejecutar Hadoop en entornos dinámicos como la nube, por lo que sería provechosa la viabilidad de incorporar estas ideas en el proyecto.

Un entorno de tolerancia a fallas bizantinas fue propuesto por Bessani et al. [7] para soportar fallas en el sistema que corrompen los resultados obtenidos por las tareas. El mecanismo propuesto se basa en ejecutar múltiples réplicas de las tareas, seleccionando como resultado final, aquel

que se reitere más veces en el total de las réplicas. En el esquema básico de tolerancia a fallas se ejecuta repetidamente cada tarea hasta que $f + 1$ ejecuciones obtengan el mismo resultado, donde f consiste en un límite superior para la cantidad de TaskTrackers defectuosos a tolerar. Se proponen varias formas para mejorar la performance en un ambiente tolerante a fallas bizantinas:

- Reducción a $f + 1$ réplicas: En lugar de seguir el esquema básico de réplica que consiste en realizar $2f + 1$ réplicas y almacenar los resultados en el HDFS, se propone reducir a $f+1$ réplicas. EN caso que todas las ejecuciones obtuvieron el mismo resultado, se toma como válido; en caso contrario se ejecutan más réplicas (f como máximo), hasta obtener $f + 1$ resultados iguales.

- Ejecución tentativa: Para no esperar a obtener los $f + 1$ resultados iguales de la fase de Map antes de iniciar el Reduce, se propone ejecutar tareas de Reduce especulativas apenas se obtengan los primeros resultados del Map. En caso de obtener progresivamente más resultados de la etapa de Map se concluye que el resultado válido es otro, entonces se reinicia la tarea de Reduce con los datos correctos.

- Respuestas reducidas: Como los resultados de las tareas pueden ser grandes, se propone almacenar solamente el resultado de la primera tarea (para cada resultado obtenido) y sólo se guarda lo mínimo de los demás resultados. De esta forma se logra como resultado reducir el tráfico en la red.

- Reducir el overhead del almacenamiento: Disminuir el factor de réplicas del HDFS a 1 en lugar de 3 (valor por defecto). Se considera que los resultados de las tareas van a estar replicados, debido a que cada tarea es ejecutada varias veces y sus resultados son guardados.

En su plataforma distribuida de Hadoop (Hadoop on the Grid – HOG), He et al. [21] introdujeron un tercer nivel de tolerancia a fallos en Hadoop: el nivel de fallas de sitio, extendiendo los dos niveles provistos por la implementación estándar (nivel de nodo y nivel de rack). La arquitectura de HOG está compuesta por tres componentes: presentación y ejecución en el grid, el HDFS y el framework de MapReduce. En el sistema de presentación y ejecución del grid los pedidos de los worker nodes de Hadoop son enviados al grid y se administra su ejecución. La presentación y ejecución de trabajos en el grid es manejada por Condor y GlideinWMS, que son frameworks genéricos diseñados para la

asignación y administración de recursos. Condor [19] es usado para administrar la presentación y ejecución de los worker nodes de Hadoop y GlideinWMS [42] es utilizado para asignar nodos en los sitios remotos. En la implementación clásica del HDFS se hace uso del conocimiento sobre los racks para ubicar los nodos, pero este método requiere conocimiento sobre la disposición física del cluster. Dependencia del rack (rack awareness) sería impracticable en HOG, donde es poco probable que los usuarios tengan conocimiento de esta disposición. Por esto se utiliza dependencia del sitio (site awareness) para la ubicación de datos y de las tareas. También se aumentó el número de réplicas a 10 (este número fue determinado de forma experimental). Para la comunicación entre el JobTracker y los TaskTrackers se utilizó HTTP, que tuvo un impacto negativo en los tiempos de inicialización y el comienzo de la transmisión de datos. En la evaluación experimental se comparó un sistema ejecutando HOG sobre la Open Science Grid (OSG) contra una implementación tradicional de Hadoop ejecutando en un cluster. Los resultados obtenidos demuestran que el sistema HOG terminó las tareas de MapReduce antes que la implementación tradicional de Hadoop, aun cuando se perdieron nodos. HOG mostró una buena escalabilidad, pero el tiempo de respuesta a las cargas de trabajo no siempre disminuyó al aumentar el número de nodos en el sistema.

Salbaroli [41] propone una versión redundante de Hadoop agregando múltiples réplicas del JobTracker. Dos alternativas son propuestas:

- Múltiples Jobtrackers con un modelo de réplica maestro-esclavo: Existen dos tipos de JobTracker: el JobTracker maestro y un número variable de JobTrackers esclavos que monitorean el estado del primero y eligen un sucesor, en caso de que el JobTracker maestro falle. El autor menciona tres formas de implementar la sincronización de este modelo: i) cada operación que afecta el estado dispara un evento de cambio de estado que es propagado a los esclavos; ii) una cantidad de cambios de estado configurable por el usuario dispara el evento de cambio de estado; y iii) nunca disparar un evento de cambio de estado, por lo que en caso de falla el esclavo tiene que empezar desde cero.

- Múltiples JobTrackers con un modelo de réplica multi-maestro: Existen múltiples JobTrackers que trabajan simultáneamente. Este modelo es más complejo y costoso en tiempo de procesamiento, debido a que existen varias entidades procesando concurrentemente los pedidos y modificando el estado. Se mencionan dos alternativas para llevar a cabo este modelo: los JobTrackers ejecutan las

mismas operaciones paralelamente, o los JobTrackers realizan diferentes operaciones. La primera opción requiere de un manejador o manager que reciba las operaciones y la envíe a los JobTrackers, pero este manager sería un SPOF, por lo que el autor descarta esta opción. La segunda opción depende fuertemente de la sincronización entre los JobTrackers, pero permite balancear la carga entre ellos. Este modelo también es rechazado por el autor porque el workload del JobTracker no es tan grande como para necesitar un balanceador.

En el presente trabajo se analizaron estas alternativas para eliminar el punto único de fallos (en inglés es Singlo Point Of Failure, por lo que se utilizará la sigla SPOF en el resto del documento) que es el JobTracker dentro de la propuesta del proyecto PERMARE. Ambas opciones propuestas fueron rechazadas, la primera porque al tener JobTrackers esclavos, cuya única tarea es monitorear al JobTracker para detectar fallas, se están desperdiciando recursos que podrían ser utilizados mientras el JobTracker está funcionando correctamente. La segunda opción fue también descartada por dos razones: i) por la complejidad de la sincronización de múltiples JobTrackers ejecutando simultáneamente, llevar a cabo esta sincronización requiere cambios mayores en Hadoop que escapan al alcance del proyecto y requieren mucho tiempo de pruebas para comprobar el correcto funcionamiento; ii) la otra razón para rechazar esta idea es que en general con un nodo dedicado para el JobTracker es más que suficiente poder de cálculo para que el JobTracker realice sus tareas, por lo que replicar el JobTracker estaría destinando recursos de cómputo a una tarea que no los necesita cuando se podrían utilizar para otras tareas con mayores requerimientos de recursos.

Kuromatsu et al. [28] propuso un sistema de auto-recuperación del JobTracker en caso de una falla sin la utilización de hardware adicional, basándose en Hadoop 0.20.2. El sistema implementado se basa en tres mecanismos:

- Checkpoint del JobTracker: Para reducir el tamaño del snapshot[1] sólo se guarda la mínima información necesaria, que incluye el status de las tareas, el status de los TaskTrackers, la asignación de las tareas y la ubicación de los datos en el HDFS. El resto de la información puede ser reconstruida a partir del entorno y de la configuración estática. Para disminuir el overhead de los checkpoints se

[1]Snapshot se refiere a un respaldo que contiene toda la información necesaria para recrear el JobTracker.

utilizaron checkpoints a partir de eventos y no checkpoints periódicos: el JobTracker genera un snapshot cada N tareas completadas, donde el valor de N se puede fijar en el archivo de configuración de Hadoop. El snapshot es almacenado en el HDFS porque de esta manera puede ser accedido desde cualquier nodo y además el HDFS logra tolerancia a fallos por réplica, por lo que se tiene alta disponibilidad y confiabilidad.

- Detección automática de fallas: Para detectar una falla en el JobTracker se utiliza el mecanismo de heartbeat (mensajes de monitoreo que serán descritos en la sección 4.1.2) de Hadoop entre el JobTracker y los TaskTrackers. Cuando el JobTracker no responde a M mensajes de heartbeat consecutivos, el TaskTracker comienza a enviarle heartbeats al nuevo TaskTracker. Este método presenta el inconveniente de que se puede tener detecciones falsas del JobTracker caído, debido por ejemplo a sobrecarga en la red. Cuando esto sucede, podría ocurrir que se cree el nuevo JobTracker aunque el original no haya fallado, lo que haría que se dupliquen las tareas y se degrade el desempeño. Para solucionar este problema, cuando el nuevo JobTracker se crea, destruye al JobTracker original.

- Recuperación de la falla: Cuando el sistema detecta una falla en el JobTracker, un TaskTracker se finaliza e inicializa al proceso del JobTracker, que recupera el estado del viejo JobTracker del snapshot en el HDFS. La elección del nuevo JobTracker se hace a partir de una lista de candidatos mantenida por el JobTracker y almacenada en el HDFS. Cuando un TaskTracker detecta que el JobTracker está caído, consulta la lista de candidatos. El TaskTracker que está ejecutando en el nodo que está primero en la lista de candidatos se convierte en el nuevo JobTracker. Los demás TaskTrackers esperan por un tiempo dado (fijado en un segundo) e intentan conectarse con el primer candidato de la lista. Si después de una cierta cantidad de intentos no logran comunicarse, comienzan a intentarlo con el segundo candidato de la lista. De esta forma todos los TaskTrackers se conectan con el nuevo JobTracker sin que sea necesaria ninguna sincronización.

Se llevó a cabo un análisis experimental para comparar el rendimiento de esta versión tolerante a fallos en el JobTracker con una versión sin modificar de Hadoop y el overhead de los checkpoints fue menor al 4.3 %.

Este trabajo fue publicado luego de que ya se había decidido cómo atacar el problema de SPOF del JobTracker dentro del proyecto PERMA-

Autor	Año	Aporte
Salbaroli	2008	Análisis de alternativas para replicar el JobTracker
Zheng	2010	Mejora de la tolerancia a fallas en la nube mediante redundancia
Bessani	2010	Tolerancia a fallas bizantinas mediante ejecución de múltiples réplicas de tareas
He et al	2012	Introducción de site awareness
Kuromatsu	2013	Snapshots del JobTracker, detección de fallas a partir del heartbeat y conversión de TaskTracker en JobTracker

Tabla 3.1: Resumen de trabajos relacionados.

RE. Muchas de las ideas presentadas en el paper de Kuromatsu resultan similares a las que se habían pensado implementar en el presente trabajo, incluyendo la realización de snapshots, la detección de fallas mediante el heartbeat y la conversión de un TaskTracker en el nuevo JobTracker al detectarse una falla en el JobTracker original. Para no repetir el trabajo presentado en la publicación de Kuromatsu, se decidió realizar una nueva implementación de Hadoop a partir de sus ideas presentadas, pero atacando los problemas que encontramos en su solución. Uno de estos problemas es la detección errónea de falla en el JobTracker que sucede cuando un grupo de TaskTrackers detecta una falla en el JobTracker mientras en realidad no la hay; esto provoca que haya dos JobTrackers en el sistema. Para solucionar este problema, Kuromatsu hace que el nuevo JobTracker mate al viejo. Esta solución tiene como consecuencia que cada vez que un solo TaskTracker detecte una falla en el JobTracker, se realice el procedimiento de recuperación, por más que el JobTracker original no haya tenido ninguna falla en realidad.

Otro problema de la solución de Kuromatsu que se ataca en este trabajo, consiste en que el TaskTracker se convierte en el nuevo JobTracker al detectarse una falla, caso en el que todas las tareas que estaba ejecutando el TaskTracker se pierden. En la solución presentada en este trabajo, el TaskTracker no se convierte en JobTracker, sino que inicia un nuevo proceso dentro del nodo que será el nuevo JobTracker, mientras que el TaskTracker sigue ejecutando sus tareas hasta terminarlas.

Hasta el momento, la mayor parte de los trabajos realizados sobre tolerancia a fallos en Hadoop desarrollan respecto a las principales fuentes de fallo. Existen varios artículos que describen los problemas de SPOF

del JobTracker. Por otro lado, existen también algunos artículos que proponen soluciones y muy pocos las han implementado. Finalmente, hay varios trabajos que tratan la tolerancia a fallos de Hadoop en términos más generales (como la tolerancia a fallas bizantinas) y no tan específicos al JobTracker. En esto último se enfocó éste trabajo.

En este proyecto se pretende implementar una versión de Hadoop que solucione el problema de SPOF del JobTracker sin utilizar recursos adicionales y con el mínimo overhead posible, de esta forma se aportará al proyecto PERMARE una versión de Hadoop sin SPOF en el JobTracker que podrá ser adaptada para ser utilizada sobre ambientes pervasivos. Los detalles de la solución propuesta se presentan en el siguiente capítulo.

Capítulo 4

Una implementación tolerante a fallos del JobTracker de Hadoop

En este capítulo se presentan los detalles de los distintos mecanismos de tolerancia a fallas, las posibles soluciones al problema del JobTracker como punto de falla del sistema, y se describen en detalle los distintos aspectos de la solución alcanzada.

4.1. Tolerancia a Fallos

Como se mencionó en la sección 2.4, Hadoop consta básicamente de dos sistemas maestro-esclavo: el framework de MapReduce (MRFW) y el Hadoop Distributed File System (HDFS). Dado que Hadoop está diseñado para trabajar en miles de nodos, que pueden y suelen ser de bajo costo, la probabilidad de que alguno falle es alta. Por este motivo, estos dos sistemas que componen Hadoop tienen sus mecanismos de tolerancia a fallos.

4.1.1. Tolerancia a fallos en el HDFS

A continuación se detallan los mecanismos de tolerancia a fallos actualmente implementados en el HDFS.

Réplica

La tolerancia a fallos en el HDFS se logra mediante la réplica de datos. Por defecto la cantidad de réplicas en el HDFS es 3 (significando

que cada bloque de datos está replicado tres veces en el HDFS). Este número puede ser aumentado por el usuario para archivos específicos que deben ser accedidos frecuentemente o contienen información crítica.

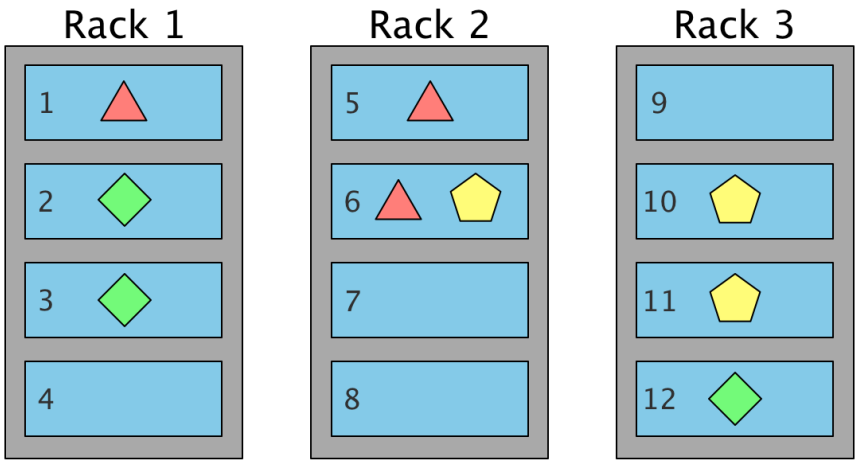

Figura 4.1: Política de almacenaje de HDFS en un cluster con distintos racks. Las figuras geométricas representan los bloques de datos manejados por el HDFS, donde las figuras de igual forma son réplicas del mismo bloque.

Un punto importante a tener en cuenta por el NameNode consiste en dónde almacenar las réplicas. Esta decisión puede tener un alto impacto en la performance y confiabilidad del HDFS. Por ejemplo, si las réplicas se ubican en distintos racks, se obtendría un sistema robusto con respecto a tolerancia a fallas. La desventaja consiste en que la comunicacion entre racks es más lenta, provocando un overhead alto en la escritura. Por otro lado, brindaría la ventaja de reducir los tiempos de lectura, al haber más probabilidad de que el cliente y los datos se encuentren en el mismo rack.

En el caso de que todas las réplicas se almacenan en el mismo rack, el overhead de escritura será minimizado, pero el tiempo de lectura aumentará a causa de que disminuye la probabilidad de que el cliente y los datos estén en el mismo rack. También disminuiría considerablemente la robustez del HDFS debido a que si el rack completo falla o pierde comunicación, se perderían todos los datos no replicados en otro rack.

Es necesario lograr un equilibrio al replicar en racks independientes o solamente en uno. Asumiendo que se trabaja con tres réplicas, el HDFS guarda dos de las réplicas en nodos distintos del mismo rack y la tercer

réplica se guarda en un rack distinto [15]. En la Figura 4.1 se representa esta política de almacenaje.

Fallas en los DataNodes

El NameNode es el encargado de administrar el HDFS. De cada archivo conoce los nodos donde se encuentran todos los bloques y las réplicas. También conoce qué nodos contienen espacio disponible para almacenar nuevos datos. Cuando un cliente quiere escribir en el HDFS, primero se contacta con el NameNode y designa a los distintos DataNodes para almacenar las réplicas, dependiendo de la configuración usada. Luego el cliente escribe los datos en estos DataNodes. Si un cliente quiere leer un archivo, le envía el pedido al NameNode, entonces este le responde al cliente una lista con los DataNodes que tienen las réplicas. Luego el cliente lee los datos directamente de los DataNodes.

Consecuentemente, es fundamental que el NameNode conozca el estado de los DataNodes en todo momento, debido a que podría comunicarle a los clientes que guarden los datos en DataNodes que no están respondiendo (lo que disminuiría la robustez del sistema), o podría decirle a los clientes que lean datos de DataNodes que tampoco están respondiendo (lo que aumentaría el overhead en comunicaciones debido a que el cliente debería consultar a varios DataNodes hasta encontrar alguno que responda). Una razón más destacable por la que el NameNode mantenga el estado actualizado de los DataNodes, consiste en que cuando un DataNode falla, el NameNode debe respaldar los bloques de datos que estaban almacenados en ese DataNode para mantener la cantidad de réplicas de los bloques. Si no lo hace se podrían llegar a perder datos, en caso de que fallen todos los DataNodes que almacenan las réplicas de un mismo bloque.

Para mantener al NameNode actualizado se utilizan dos mecanismos de comunicación entre el NameNode y los DataNodes [15]:

- Cada DataNode mantiene un reporte de los bloques del su nodo. Cada una hora, el DataNode envía este reporte al NameNode, de esta forma el NameNode siempre tiene una visión actualizada de las réplicas en el cluster.

- Cada DataNode envía una notificación al NameNode cada diez minutos, de esta forma el NameNode sabe si los DataNodes están operando correctamente. Si después de diez minutos el NameNode no recibe una notificación de un DataNode, el NameNode asume

que el DataNode falló y comienza a replicar los bloques que tenía el DataNode en otros DataNodes.

Falla en el NameNode

El NameNode es único en el HDFS. Todo el sistema depende de él y en caso de falla, colapsa todo el sistema, lo que lo convierte en un SPOF. Para minimizar los efectos de un fallo en el NameNode, Hadoop utiliza el BackupNode. El BackupNode realiza checkpoints periódicos del HDFS y mantiene en memoria una copia actualizada del sistema de archivos, la que es sincronizada con el NameNode [10]. En el caso de que el NameNode falle, el BackupNode contiene una copia del sistema de archivos, por lo que puede utilizarse para reiniciar el NameNode.

Como se acaba de mencionar, el BackupNode tiene una copia actualizada del namespace. Gracias a esto, puede crear los checkpoints sin descargar el checkpoint y los archivos journal del NameNode. Esto provoca que el proceso de checkpoint sea más eficiente, debido a que el procesamiento se realiza de forma local en el BackupNode [10].

4.1.2. Tolerancia a fallos en el MRFW

Como fue desarrollado anteriormente, Hadoop fue diseñado para ejecutarse en clusters o plataformas de grid computing. En estos entornos las fallas no poseen tanta importancia como en un entorno pervasivo, porque los nodos operan sin fallas por largos períodos de tiempo [45]. De todos modos, la falla de un nodo continua siendo relevante. De no ser tomado en cuenta, podría significar recomenzar un trabajo desde cero, perdiendo horas o incluso días invertidos.

Para limitar el peligro de fallas en el MRFW, Hadoop utiliza dos métodos: un mecanismo de heartbeat y la ejecución especulativa. Para estudiar estos métodos fue necesario inspeccionar el código fuente, debido a que la información disponible respecto a la tolerancia a fallos en Hadoop es escasa y enfocada principalmente en el HDFS.

Heartbeat

Uno de los componentes principales de Hadoop es el JobTracker, el organizador y el principal coordinador de tareas. Además, el JobTracker está a cargo de la distribución de las tareas de MapReduce entre los nodos disponibles. Cada vez que el JobTracker recibe un nuevo trabajo para ejecutar, contacta a un grupo de procesos TaskTracker, que son demonios

que ejecutan en los working nodes (existe un TaskTracker por cada worker en la infraestructura). Las tareas de MapReduce son asignadas a estos working nodes para los que sus demonios TaskTracker reportan que tienen espacio disponible para realizar tareas (varias tareas asignadas para el mismo worker son manejadas por un único demonio TaskTracker) [38].

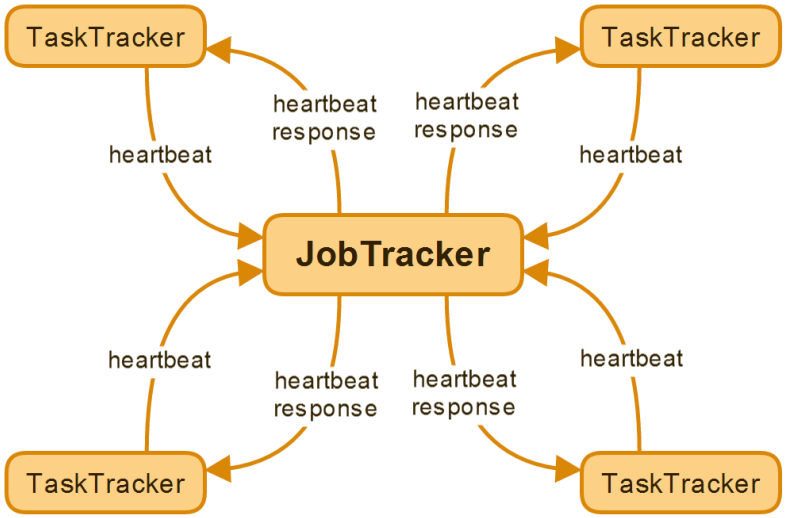

Figura 4.2: Comunicación entre el TaskTracker y el JobTracker en Hadoop [27]

El JobTracker debe conocer en todo momento el estado de los TaskTrackers, por esto los monitorea constantemente usando mensajes de control llamados señales de heartbeat. Estas señales de heartbeat son enviadas desde los TaskTrackers al JobTracker que, después de recibirla, le envía una respuesta al TaskTracker que incluye algunos comandos (como iniciar o finalizar una tarea, o si el TaskTracker debe ser reiniciado). Esta respuesta es llamada HeartbeatResponse. El esquema del procedimiento de heartbeat puede verse en la figura 4.2.

En caso de que el JobTracker no reciba ningún heartbeat del TaskTracker durante un período de tiempo (por default, es fijado en 10 minutos), el JobTracker entiende que el worker asociado a ese TaskTracker ha fallado. Cuando esto sucede, el JobTracker necesita reagendar todas las tareas pendientes y en progreso a otro TaskTracker. También deben ser reagendadas todas las tareas de Map si pertenecen a trabajos incompletos, ya que los resultados intermedios que estaban en el TaskTracker fallado pueden no estar disponibles para las tareas de Reduce.

Algunos de los datos más relevantes en el heartbeat que el TaskTracker le envía al JobTracker son:

- *TaskTrackerStatus*: Contiene información sobre el worker manejado por el TaskTracker, como la memoria virtual disponible e información de la CPU.
- *Restarted*: Indica si el TaskTracker fue reiniciado.
- *Si es el primer heartbeat*: Indica si es el primer heartbeat recibido por el TaskTracker.
- *Si el nodo requiere ejecutar más tareas*

El JobTracker mantiene la blacklist con los TaskTrackers fallidos y también el último heartbeat recibido de cada TaskTracker. De esta forma, cuando el JobTracker recibe un nuevo heartbeat que informa que es el primer heartbeat o que el TaskTracker fue reiniciado, el JobTracker, usando esta información, puede decidir si reiniciar el TaskTracker o removerlo de la blacklist (ver sección 4.1.2).

Luego de recibido el heartbeat, el status del TaskTracker es actualizado en el JobTracker y el HeartbeatResponse es creado. Este HeartbeatResponse contiene las próximas acciones a ser llevadas a cabo por el TaskTracker. Si hay tareas para ejecutar, el TaskTracker requiere nuevas tareas (este es un parámetro del heartbeat) y no está en la lista negra, entonces se crean tareas de limpieza y configuración inicial (conocidas como tareas de setup). Si no hay tareas de limpieza o setup para ejecutar, el JobTracker obtiene nuevas tareas. El JobTracker también busca:

- Tareas para ser eliminadas
- Trabajos para ser eliminados/limpiados
- Tareas cuya salida todavía no haya sido guardada

Todas estas acciones, si aplican, son agregadas a la lista de acciones que serán enviadas en el HeartbeatResponse. Los mecanismos de tolerancia a fallos implementados en Hadoop se limitan a reasignar tareas cuando la ejecución falla. Hay dos escenarios soportados:

- Cuando una tarea asignada a un TaskTracker falla, la comunicación por el heartbeat es usada para notificar al JobTracker, quien reasigna la tarea a otro nodo en caso de ser posible.

- Cuando un TaskTracker falla, el JobTracker notará la falla porque no va a recibir heartbeats del TaskTracker. Entonces, el JobTracker va a reasignar las tareas que tenía el TaskTracker a otro TaskTracker.

Blacklist de TaskTrackers

Para intentar disminuir la cantidad de tareas reasignadas, el JobTracker mantiene una lista negra con los TaskTrackers que considera que pueden fallar con alta probabilidad. Un TaskTracker que está en la lista negra permanece en comunicación con el JobTracker, pero no son asignadas tareas al worker correspondiente. Cuando un número específico de tareas (por defecto 4) pertenecientes a un job específico manejadas por un TaskTracker fallan,el sistema considera que ha ocurrido una falla. Si el TaskTracker tiene más de un número específico de fallas (por defecto 4), el TaskTracker es pasado a la lista negra.

Para que un TaskTracker sea quitado de la lista negra hay dos posibilidades: si una o varias fallas expiran (por defecto una falla expira luego de 24 horas), el TaskTracker es automáticamente quitado de la blacklist, sin necesidad de ninguna comunicación; la otra posibilidad es que el TaskTracker se reinicie, cuando esto sucede el JobTracker es notificado en el heartbeat enviado por el TaskTracker y el TaskTracker es quitado de la lista negra.

Ejecución Especulativa

Un problema con el sistema Hadoop es que al dividir las tareas entre muchos nodos, es posible que algunos nodos lentos limiten la velocidad de todo el sistema. Por ejemplo, si un nodo tiene un controlador de discos lento, entonces estaría leyendo los datos de entrada más lento que los otros nodos. Por lo tanto, cuando la tareas de Map hayan terminado en los otros nodos, el sistema todavía debería esperar a que la última tarea de map finalice, lo que podría demandar mucho más tiempo de procesamiento [55]. En el peor de los casos, este nodo lento podría fallar al final y la tarea que todo el sistema estaba esperando que terminara tendría que ser comenzada nuevamente en otro nodo, lo que podría llegar a tener un gran impacto en la performance del sistema.

Para solucionar este problema, cuando la mayoría de las tareas en un trabajo están finalizando, Hadoop envía copias redundantes de las tareas que todavía no han terminado a los nodos que no tienen otras tareas que ejecutar. Cuando una tarea finaliza, el TaskTracker que la ejecutó se lo

comunica al JobTracker (a través del heartbeat), quien le ordena a los demás TaskTrackers que estaban ejecutando la misma tarea que abandonen esta tarea y descarten sus resultados [55]. Mediante este mecanismo de ejecución especulativa, Hadoop minimiza el efecto de nodos lentos mejorando la performance del sistema.

4.2. Solución al problema SPOF del JobTracker

Se evaluaron varias soluciones para resolver el problema del SPOF del JobTracker en Hadoop:

- Múltiples JobTracker
- JobTracker secundario
- Recuperación dinámica del JobTracker usando snapshots

Estas alternativas se describen en las siguientes secciones.

4.2.1. Múltiples JobTracker

La primera opción evaluada fue tener más de un JobTracker ejecutando en un cluster de Hadoop. Estos JobTrackers trabajaran en conjunto recibiendo trabajos de los JobClients, y distribuyendo las tareas entre los TaskTrackers.

Para poder realizar esto se necesita diseñar un protocolo de comunicación entre JobTrackers, permitiéndoles la sincronización del trabajo pendiente/finalizado, la planificación de las tareas en conjunto y poder conocer el estado de cada uno. Este protocolo de comunicación sería análogo al utilizado por el heartbeat entre el JobTracker y los TaskTrackers (como se detalla en la sección 4.1.2), pero operaría exclusivamente entre JobTrackers manteniendo una especie de backup mutuos. Un punto importante en el diseño de este protocolo de comunicación es como evitar conflictos de acciones entre ellos. No se debe de tener que depender de una única entidad, ya que se tendría teniendo un SPOF.

Algunas de las alternativas para resolver la dependendencia de una entidad para evitar conflictos son:

- Tener una jerarquía interna de líder de los JobTrackers para poder realizar una planificación única sin conflictos. Este líder podría

ser rápidamente reemplazado, ya que el resto de los JobTrackers tendría acceso a la misma información que el líder original.

- Realizar una partición lógica de Hadoop, para que cada parte sea administrada por uno de los JobTrackers, y en caso de que un JobTracker tenga tareas para un TaskTracker en control de otro JobTracker, le tendría que enviar las tareas a través del JobTracker que administra la partición de Hadoop a la que pertenece el TaskTracker.

En cualquiera de los dos alternativas anteriores, en caso de que uno de los JobTrackers falle, los otros JobTrackers se percatarían gracias a los heartbeats. En tal caso, existen dos opciones para la recuperación de la falla: la primera es levantar un nuevo JobTracker en otro nodo, compartiendo la información que disponible del resto de los JobTrackers, mientras que la segunda opción es no tomar ninguna acción y simplemente notificar a los JobClients y TaskTrackers para que se conecten a alguno de los otros JobTrackers activos.

Por un lado, tener múltiples JobTrackers permite alcanzar mayores niveles de carga del JobTracker, aunque se debe realizar un análisis con mayor profundidad sobre los aspectos relacionados al overhead que produce la comunicación entre JobTrackers. Por el otro lado, una de las desventajas de esta solución es que al tener más de un JobTracker, se esta perdiendo capacidad potencialmente valiosa del cluster que podría ser destinada para nodos trabajadores.

4.2.2. JobTracker secundario

Otra alternativa es tener un nodo secundario que funcione como JobTracker pasivo. El JobTracker primario envía toda la información que recibe y produce al JobTracker pasivo, y el JobTracker secundario recibe la información actualizando, su estado pero sin realizar acciones sobre el sistema.

En el caso de que el JobTracker primario falle, el JobTracker secundario toma el lugar del primario, mientras que se queda a la espera de que el JobTracker primario resuma su operación. La recuperación del JobTracker original puede darse poco después (segundos) si el problema fue a nivel de software, o mucho tiempo después (horas o incluso días) si el problema se encuentra a nivel de hardware y se necesite reemplazar algún componente de la infraestructura.

Una vez resumido el JobTracker primario, se puede convertir nuevamente en el activo pidiendo toda la información al secundario, o se puede

convertir en el secundario, recibiendo del nuevo JobTracker primario la información y esperando pasivamente en caso de que falle.

Para esta alternativa se debería de crear un protocolo análogo al del heartbeat con el propósito de transferir información del JobTracker primario al JobTracker secundario y para detectar fallas del JobTracker primario. Dicho protocolo sería más simple que el heartbeat, ya que los datos se mueven en un único sentido, del JobTracker primario al secundario en la operación normal.

A diferencia de la solución anterior en la que se plantea el uso de múltiples JobTrackers, esta permite un diseño más simple sin tener que diseñar un protocolo potencialmente complejo. La desventaja es que esta manera de guardar la información tiene el potencial de triplicar el tráfico de salida del JobTracker, ya que debe enviar al JobTracker secundario la información que se le envía a los TaskTrackers, y todos los cambios de estado que recibe de éstos. Es posible implementar un protocolo que envié los cambios de a varios al mismo tiempo minimizando la transferencia de datos sin tener que reenviar todo lo que envía y recibe.

Como desventaja sucede que, al igual que en la solución anterior, se tiene el problema de que se utiliza un nodo dedicado ocupando recursos sólo para el caso en que falle el JobTracker.

4.2.3. Recuperación dinámica del JobTracker usando snapshots

Tratando de evitar la desventaja más importante de las alternativas anteriores, relacionada a la subutilización de un recurso potencialmente valioso, surge la idea de poder recuperar el JobTracker en cualquiera de los nodos usando un snapshot que refleje el estado de este, previo al fallo.

Para implementar esta estrategia, es necesario instrumentar la creación de un snapshot cada vez que el JobTracker cambia de estado. El cambio de estado puede ser por una tarea planificada en un nodo, una tarea completada, un nuevo job, y muchas otras razones que modifican el estado interno del JobTracker durante su ejecución. En algunos cambios de estado, se necesita que la generación de snapshots se realice de forma inmediata, de manera de que no se pueda continuar con la ejecución sin persistir el estado de JobTracker con el nuevo estado. Esta realización de snapshots de forma automática puede ser problemática, ya que persistir esta información de forma constante es potencialmente un problema importante al desempeño del sistema.

Teniendo un snapshot, en el caso de que el JobTracker falle se debe

de poder elegir de forma distribuida un nodo esclavo que reemplace al JobTracker, recuperar el estado original a partir del snapshot y notificar al cluster del nuevo JobTracker para luego seguir ejecutando los trabajos como lo estaba haciendo el JobTracker original.

Para poder realizar las acciones mencionadas en los párrafos anteriores, es necesario tener algún método de comunicación entre los nodos esclavos. Usando dicho método de debe de poder ejecutar algún procedimiento de selección de reemplazo de JobTracker, además de poder notificar a todos los TaskTrackers quién es el nuevo JobTracker.

La ventaja que ofrece la utilización de snapshots es una manera de recuperar el JobTracker sin desperdiciar uno o más nodos, permitiendo maximizar la capacidad del cluster y perdiendo el SPOF en el JobTracker.

La utilización de snapshots es la solución con la que se decidió avanzar. En la siguiente sección se describe cómo se pudieron resolver las dificultades mencionadas en los párrafos anteriores y se detalla el diseño final de la solución.

4.3. Diseño de la solución al SPOF del JobTracker

Como se mencionó en la sección anterior, se decidió trabajar con la alternativa de realizar snapshots del JobTracker, y en caso de que el JobTracker falle recuperarlo en otro nodo del cluster a partir de estos snapshots.

Uno de los problemas que se plantearon anteriormente en este documento, es el potencial impacto en el desempeño del JobTracker debido a los snapshots. La necesidad de actualizar el snapshot de forma constante puede ser un problema en la eficiencia del sistema.

En nuestro enfoque, lo que se propone para disminuír la sobrecarga en el sistema debida a la actualización del snapshot es la creación de un snapshot incremental. En lugar de crear un snapshot completo en cada etapa, se propone la creación de un snapshot completo inicial. Una vez creado este snapshot inicial, a medida que se modifica el estado del JobTracker se actualizan únicamente las secciones del snapshot que se modifican, minimizando el trabajo de generación del snapshot. Para poder realizar esta actualización, se debe definir cómo particionar el estado del JobTracker para actualizar sólo las partes que se modifican.

Se decidió particionar el snapshot actualizando los atributos de la clase JobTracker de forma independiente. La ventaja de esta opción es que

permite un control manual de cuándo y qué agregar en la actualización del snapshot.

4.3.1. Atributos del JobTracker

Se realizó un relevamiento de los atributos de la clase JobTracker y cómo son utilizados en la ejecución del JobTracker. A continuación se listan los atributos que forman parte del snapshot con una breve descripción de su uso:

- *int nextJobId*. Es utilizado para la creación secuencial de los identificadores de los jobs. Cuando el JobClient se conecta con el JobTracker para iniciar un job, le solicita este valor para asignarlo al nuevo job y, de esta forma, el nuevo job tiene un nombre único.
- *Map<String, Set<TaskTracker>> hostnameToTaskTracker*, mantiene un registro de los TaskTrackers en cada hostname.
- *Map<String, Node> hostnameToNodeMap*, es un mapeo entre los hostname y el Node del cluster. Este valor sirve para crear la topología del cluster.
- *Set<Node> nodesAtMaxLevel*, con el mismo propósito que el atributo anterior, mantiene un set de nodos de máximo nivel. En la estructura de datos de Hadoop los nodos tienen una jerarquía; se tienen nodos por data center, por rack y por nodo final. Esta lista mantiene un listado de los nodos raíces que abarcan todo el cluster.
- *Map<JobID, JobInProgress> jobs*, mantiene un listado de todos los jobs que se ejecutaron o están ejecutándose.
- *Map<TaskAttemptID, TaskInProgress> taskidToTIPMap*, mapa de las tareas en proceso, usando como clave el id de las tareas se obtiene el objeto TaskInProgress.
- *TreeMap<TaskAttemptID, String> taskidToTrackerMap*, mapeo del id de las tareas al nombre del TaskTracker que la está ejecutando.
- *HashMap<String, TaskTracker> taskTrackers*, mapa de TaskTrackers usando el nombre de ellos como clave.
- *TreeSet<TaskTrackerStatus> trackerExpiryQueue*, mantiene un listado de los estados de los taskTrackers que están por expirar para eventualmente borrarlos.

- *Map<String, HeartbeatResponse> trackerToHeartbeatResponseMap*, por cada uno de los TaskTrackers se guarda la respuesta del último heartbeat. Esta respuesta es utilizada para saber si un TaskTracker ya se contactó y descartar respuestas duplicadas o fuera de orden.
- *Map<String, Set<JobID>> trackerToJobsToCleanup*, mantiene una mapa de los jobs para los cuales hay que realizar el cleanup en cada TaskTracker.
- *TreeMap<String, Set<TaskAttemptID>> trackerToMarkedTasksMap*, guarda un set de los identificadores de las tareas finalizadas en cada uno de los TaskTrackers.
- *TreeMap<String, Set<TaskAttemptID>> trackerToTaskMap*, por cada uno de los TaskTrackers guarda un set de los identificadores de las tareas que se están ejecutando en este momento.
- *Map<String, Set<TaskAttemptID>> trackerToTasksToCleanup*, al igual que para los jobs, por cada TaskTracker se guarda un listado de las tareas para las cuales hay que hacer cleanup.
- *TreeMap<String, ArrayList<JobInProgress>> userToJobsMap*, mantiene una lista de los jobs ejecutados por un mismo usuario.
- *FaultyTrackersInfo faultyTrackers*, maneja la lista de TaskTrackers en la blacklist.
- *int totalMaps*, registro de la cantidad de tareas de tipo Map que se realizaron en el cluster.
- *int totalReduces*, registro de la cantidad de tareas de tipo Reduce que se hicieron.
- *int occupiedMapSlots*, suma de la cantidad de espacios de tareas de tipo Map de todos los TaskTrackers que se encuentran ocupados.
- *int occupiedReduceSlots*, suma de la cantidad de espacios para tareas de tipo Reduce que se encuentran ocupados.
- *int reservedMapSlots*, suma de la cantidad de espacios de tareas de tipo Map de todos los TaskTrackers que se encuentran reservados.
- *int reservedReduceSlots*, suma de la cantidad de espacios para tareas de tipo Reduce que se encuentran reservados.

- *int totalMapTaskCapacity* y *int totalReduceTaskCapacity*, el número total de espacios para realizar tareas en el cluster, incluyendo ocupados, reservados y libres.
- *int totalSubmissions*, número total de jobs enviados a ejecutar por el JobTracker.

4.3.2. Persistencia del Snapshot

Una vez identificados los atributos necesarios para el snapshots, se procede a definir dónde guardar el snapshot a generar. El almacenamiento debe cumplir con las siguientes propiedades:

- *Ser tolerante a fallas.* No puede perderse información si falla el JobTracker o cualquier otro elemento del cluster.
- *Permitir escrituras parciales.* Se necesita poder escribir los atributos que se definieron en la parte anterior de forma independiente.
- *Ser eficiente.* La operación de guardado del snapshot no debe de ser muy costosa en tiempo de acceso para no afectar el desempeño del JobTracker.

La primera alternativa que se evaluó fue guardar el snapshot directamente en el HDFS. Esta alternativa cumple las primeras propiedades: por un lado permite realizar escrituras parciales, los atributos se podrían tratar como archivos y escribirlos de forma independiente; por el otro lado, al ser un sistema de archivos distribuido, dependiendo de la configuración de replicación de datos que se use, se puede tener una alta tolerancia a fallos. Sin embargo, surge un problema con el tercer requerimiento, ya que el HDFS está optimizado para escribir una vez y escribir muchas veces, escribiendo desde un stream de datos con archivos muy grandes (tiene un block size de 64MB), por lo que claramente no cumple con la última propiedad buscada.

Buscando alternativas se encontró ZooKeeper, otro proyecto de Apache que se detalla a continuación.

4.3.3. ZooKeeper

Apache ZooKeeper [18] [24] es un servicio para la centralización de configuración y coordinación de aplicaciones distribuidas, con el fin de simplificar tareas innatas a la ejecución de sistemas distribuidos como pueden ser:

- Condiciones de carrera, que suceden cuando varios recursos acceden y modifican un recurso compartido, de manera de poder dejar el sistema en un estado inconsistente.
- Bloqueo mutuo o deadlocks.
- Detección de fallos, un problema importante y comúnmente encontrado en los sistemas distribuidos es poder detectar, con un alto grado de seguridad, que un sistema falló y poder distinguirlo de otros problemas como un problema de conectividad temporal, o degradación de la red.
- Denominación o naming, esto no representa un problema en sistemas lineales, pero se convierte en un problema complejo en sistemas distribuidos.

ZooKeeper trabaja como un sistema de archivos compartido. En lugar de tener archivos tiene lo que se llaman znodes, o nodos de ZooKeeper. Estos znodes pueden ser tanto registros de datos como padres de otros znodes. La estructura se maneja como un sistema de archivos: se tiene el znode raíz "/", y los nodos hijos de este se pueden acceder de la forma "/app1", creando de esta manera namespaces.

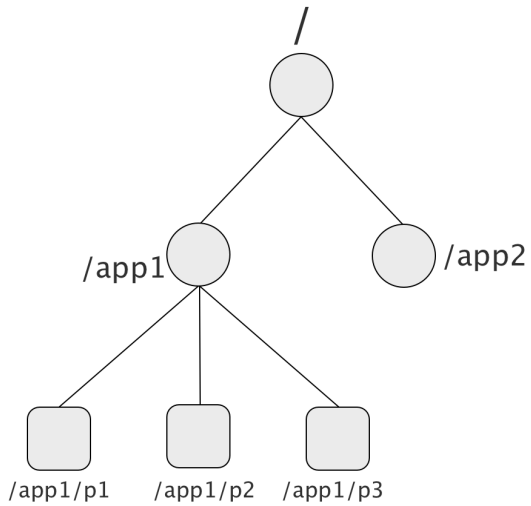

Figura 4.3: Estructura interna de nodos de ZooKeeper

ZooKeeper no fue diseñado como una base de datos o para guardar objetos extensos. Tiene un límite de 1MB de datos por znode, y se recomienda que en promedio se guarde mucho menos datos que este límite, ya

que trabajar con grandes cantidades de datos puede afectar la operación del sistema.

Para cada znode también se mantiene un número de versión de cambios, cambios en la lista de control de accesos y registro del tiempo de modificaciones. Cada vez que se va a escribir datos en un znode, se debe especificar el número de versión que se devolvió al realizar la lectura; en caso de que las versiones sean distintas, la actualización de datos falla.

Existen cuatro tipos distintos de znodes:

- Znodes normales: estos se crean especificando el nombre y el path donde se guarda, y se deben de borrar manualmente.
- Znodes de secuencia: estos znodes permiten la posibilidad de ser creados de forma ordenada y con nombre único. Cuando se crea un znode se especifica el path y el prefijo y ZooKeeper le agrega un número único al final del nombre para el znode padre. Por ejemplo, si se agregan varios znodes de secuencia al znode */app1* con prefijo *cliente_app*, ZooKeeper crea los znodes */app1/cliente_app0000000001*, */app1/cliente_app0000000002*, etc. De esta forma se asegura el orden y la unicidad de los nombres.
- Znodes efímeros o como se llaman en inglés, *ephemeral*: estos znodes existen mientras se mantenga la sesión del cliente con ZooKeeper activa y se borran automáticamente al terminar la sesión. Estos znodes no pueden tener hijos ya que, como lo dice el nombre, no se espera que existan por mucho tiempo.
- Znodes efímeros de secuencia: Combinando los znodes efímeros con los de secuencia, ZooKeeper permite crear znodes con nombres únicos y ordenados que se borran al momento de perder la sesión con el cliente que los creó.

Al igual que Hadoop, ZooKeeper tiene una arquitectura maestro esclavo. Cada nodo ejecuta el mismo servicio, y se elige a uno de estos como su líder. Este líder es el encargado de recibir las consultas que cambian el estado de ZooKeeper.

Para replicar el árbol de datos en todos los nodos, ZooKeeper funciona en modo quórum. En vez de esperar a que cada servidor guarde un dato antes de continuar, se crea un quórum, es decir, una mínima cantidad de nodos que hayan guardado los datos antes de decirle al cliente que se guardó exitosamente.

ZooKeeper provee las siguientes funciones que actúan como la interfaz del sistema:

- **create**, crea un znode en determinada posición.
- **delete**, borra un znode.
- **exists**, chequea si existe o no un determinado znode.
- **get data**, devuelve la información guardada en un znode.
- **set data**, escribe información en un znode.
- **get childen**, devuelve la lista de znodes hijos de un znode.
- **sync**, espera a que los datos se propaguen correctamente.

ZooKeeper cumple con las propiedades necesarias para guardar el snapshot del JobTracker mencionadas en la sección 4.3.2. Es tolerante a fallas, muchos proyectos lo utilizan por su confiabilidad al momento de integrarse con aplicaciones. Y, además de su alta disponibilidad, ZooKeeper tiene un excelente desempeño [39].

Se decidió la creación de un znode a nivel de raíz llamado *jobTracker*, de manera de tener toda la información del snapshot y de coordinación dentro de un mismo znode. Este nombre puede cambiarse modificando la propiedad *mapred.zookeeper.jobTrackerNode* en el archivo de configuración *mapred-site.xml*. Dentro de este znode maestro, se crea cada atributo del JobTracker como un znode separado del resto. Por ejemplo, el objeto *hostnameToTaskTracker* se persiste en */jobTracker/hostnameToTaskTracker*.

4.3.4. Método y formato de persistencia

Conociendo donde guardar la información, es necesario poder transferir el objeto Java en memoria a ZooKeeper en la generación/actualización del snapshot y en dirección opuesta al momento de recuperación. Para esto se evaluaron tres alternativas para realizar la persistencia:

- Serialización de Java, usando la interfaz `Serializable`.
- Serialización utilizada por Hadoop con la interfaz `Writable`. Hadoop tiene un mecanismo de serialización optimizado para seleccionar solo los datos a transferirse minimizando la transferencia en la red. Para utilizar esta interfaz se deben implementar los métodos `write` y `readFields` que respectivamente escriben y leen un `DataOuput`/`DataInput`. Esta interfaz es utilizada por Hadoop tanto para el mecanismo de heartbeat como para los jobs MapReduce.

- Persistencia en un formato alternativo como puede ser JSON o XML.

La primera alternativa, realizar la persistencia usando la serialización nativa de Java, es la más simple. Java automáticamente permite la serialización si la clase implementa la clase `java.io.Serializable`. Como este método serializa el objeto entero, se necesitaría especificar ciertos atributos dentro de las clases como transient[1] para evitar referencias circulares o persistir datos innecesarios.

Para utilizar la interfaz de serialización `Writable` de Hadoop es necesario implementar los métodos `write` y `readFields`. Dentro de estos métodos se escriben/leen los distintos atributos del objeto que se quieren utilizar, pudiéndose elegir qué persistir y en qué orden. A diferencia de la serialización usando la serialización nativa de Java, al especificar manualmente los atributos y el orden, no se debe de guardar la metadata de los atributos (nombre y tipo del atributo, nombre de la clase), permitiendo un resultado final más compacto. Por otro lado, la implementación de los métodos `write` y `readFields` en cada clase y atributos de forma recursiva puede ser muy costosa en tiempo. Otro problema de la serialización de Hadoop es que si en algún caso se quiere persistir una colección o clase de la librería estándar de Java, se debe usar `Serializable` o crear nuevas clases que extiendan la colección o clase y que implementen `Writable` de forma correcta.

Por último, la tercera opción analizada, realizar la persistencia en un formato alternativo, no ofrece grandes ventajas más allá de tener un formato legible. La implementación sería similar en complejidad y magnitud a la necesaria para utilizar la serialización de Hadoop, perdiendo las ventajas de tamaño que se pueden alcanzar utilizando la serialización con la interfaz `Writable`.

Se investigó en profundidad la serialización de Hadoop, tratando de realizar un prototipo rápido con pocos atributos del JobTracker, únicamente persistiendo y no recuperando. Rápidamente fue claro que el costo de implementación era muy alto y que al modificar tantas clases y atributos se tendría un alto riesgo de generar errores en el código. Se decidió finalmente utilizar el modo de serialización estándar de Java usando la interfaz `Serializable`.

[1] Transient es una palabra reservada de Java que indica que un atributo no debe ser serializado.

4.3.5. Momento de realización de la persistencia

Teniendo el método de persistencia y el lugar donde realizarla, se debe determinar en qué momento de la ejecución del JobTracker realizar esta persistencia. Se tienen dos alternativas: 1) se almacena el snapshot de forma sincrónica, bloqueando el funcionamiento del JobTracker hasta que el snapshot se encuentre guardado correctamente, o 2) se almacena el snapshot de forma asincrónica, de manera que el JobTracker no se bloquea.

La ventaja de la persistencia sincrónica es que se asegura que el JobTracker no siga trabajando si el snapshot tiene una versión no final. De esta forma, de fallar el JobTracker se tiene la seguridad de que el snapshot tiene la última versión antes de la falla. La desventaja de este método de persistencia es que puede ser un cuello de botella en el desempeño del JobTracker, en caso que la persistencia se enlentezca por alguna razón, el JobTracker también se enlentece.

En la modalidad de persistencia asincrónica, las ventajas y desventajas son las opuestas a las de persistencia sincrónica: en caso de fallar el JobTracker no existe seguridad de que el snapshot tenga el estado real que poseía al momento de fallar y por este motivo se puede perder información de forma permanente. Por el otro lado, la ventaja de esta modalidad de persistencia es que no bloquea la ejecución del JobTracker, y por esta razón no degrada el desempeño de este.

En busca de un compromiso entre las ventajas y desventajas de la persistencia sincrónica y la persistencia asincrónica, se propuso implementar una solución híbrida entre ambas. Se consideran prioritarios ciertos atributos del JobTracker y su persistencia al snapshot será de manera bloqueante con respecto a la ejecución del JobTracker. Por el otro lado, ciertos atributos se consideran secundarios ya que su pérdida no inhabilita o retrasa la ejecución o finalización de los jobs; estos atributos serán persistidos en el snapshot de forma asincrónica.

Para esta solución híbrida entre sincronización bloqueante y no bloqueante se crea un clase interna `SnapshotService`, que implementa `Runnable` y que se ejecuta de forma paralela al resto de los procesos del JobTracker. Este thread procesa, cada cierto tiempo (especificado en 5 segundos), una lista de atributos a actualizar y los guarda en el snapshot uno por uno, leyendo el atributo en cuestión al momento del guardado. Para agregar atributos a esta lista, se provee un método `addEvent` que recibe un `String` con el nombre del atributo. El método lo encapsula en un objeto que guarda el tiempo y lo agrega a la lista de atributos a persistir. Para procesar los atributos, primero se pasa la lista a un `HashSet` usando los

nombre de los atributos como clave, evitando de esta forma respaldar elementos repetidos. Finalmente se vacía la lista. Se puede ver un fragmento del código en el listado 4.1.

Para los casos de actualización sincrónica, el servicio `SnapshotService` provee el método *update* usado por la actualización asincrónica, que de manera bloqueante busca el valor del objeto actual, lo serializa y lo guarda en ZooKeeper.

```
Set<String> attributesToUpdate = new HashSet<String>();

synchronized(modificationsQueue) {
  for (ModificationEvent event : modificationsQueue) {
    attributesToUpdate.add(event.getAttribute());
  }

  // Remove all elements from modifications queue.
  modificationsQueue.clear();
}

// With the set with attributes we update them in Zookeeper.
for (String attribute : attributesToUpdate) {
  update(attribute);
}
```

Listado 4.1: Actualización de atributos.

En la tabla 4.1 se listan todos los elementos que se guardan en el snapshot, indicando si son guardados de forma sincrónica o asincrónica.

4.3.6. Detección de fallas

Para poder detectar fallas de forma fiable sin tener falsos positivos, es necesario algún tipo de comunicación entre todos los elementos del sistema. Sin esta comunicación puede suceder que un nodo tenga problemas de red y no se pueda comunicar con el JobTracker y asumir de forma incorrecta que el JobTracker falló. Para esto se utiliza ZooKeeper, que guarda el estado real del JobTracker. Se utiliza un znode ephemeral creado por el JobTracker, por defecto en la ruta */jobTracker/alive*. En caso de que el JobTracker pierda la conexión con ZooKeeper, se puede asumir que el JobTracker se encuentra caído. Este nodo también cumple de puntero al JobTracker, guardando el hostname del JobTracker como valor del znode.

En caso de una caída del JobTracker, tanto los TaskTrackers como

Atributo	Tipo de sincronización
nextJobId	sincrónica
jobs	sincrónica
taskidToTIPMap	sincrónica
taskidToTrackerMap	sincrónica
taskTrackers	sincrónica
trackerToHeartbeatResponseMap	sincrónica
trackerToTaskMap	sincrónica
hostnameToTaskTracker	asincrónica
hostnameToNodeMap	asincrónica
nodesAtMaxLevel	asincrónica
trackerExpiryQueue	asincrónica
trackerToJobsToCleanup	asincrónica
trackerToMarkedTasksMap	asincrónica
trackerToTasksToCleanup	asincrónica
totalMaps	asincrónica
totalReduces	asincrónica
occupiedMapSlots	asincrónica
occupiedReduceSlots	asincrónica
reservedMapSlots	asincrónica
reservedReduceSlots	asincrónica
totalMapTaskCapacity	asincrónica
totalReduceTaskCapacity	asincrónica
totalSubmissions	asincrónica

Tabla 4.1: Elementos guardados en el snapshot.

los JobClients se ven afectados perdiendo la conexión con el JobTracker. Estos mantienen una conexión con el JobTracker por la cual envían y reciben la información relacionada a los heartbeats y los JobStatus respectivamente. Al perderse la conexión, el mecanismo propio de Hadoop de la conexión, realiza inicialmente 10 reintentos de forma automática. En caso de no reconectarse con el JobTracker se lanza una `SocketException`.

Cuando se captura la excepción `SocketException` lanzada por la pérdida de conexión, se chequea ZooKeeper para confirmar que el fallo haya sucedido realmente. De esta manera, en caso de tener un TaskTracker que perdió toda conectividad con el resto del sistema, puede verificar si es un problema de JobTracker o si es un problema propio. Esto se logra verificando el valor */jobTracker/alive* del *znode*: en caso de existir y tener el valor del JobTracker original, se vuelve a intentar la conexión

normalmente; en caso contrario, se espera a que el znode exista y tenga un valor distinto del JobTracker original.

4.3.7. Selección del nuevo JobTracker

Una vez detectada la falla del JobTracker y la confirmación con ZooKeeper, se debe elegir un nodo para ser el sucesor y convertirse en el nuevo JobTracker. Haciendo uso de los znodes ephemerales secuenciales de ZooKeeper, al iniciarse un TaskTracker crea un znode ephemeral secuencial en el znode */jobTracker/taskTrackers*. Como se mencionó en la sección 4.3.3, la creación de znodes secuenciales garantiza que cada nodo creado tenga un sufijo numérico incremental sin repeticiones, mientras que la propiedad ephemeral tiene como resultado que los nodos más estables se encuentren al inicio y los nodos que han sido reiniciados o han tenido problemas de conexiones se encuentren al final de la lista. Es por esta simple propiedad que se decidió tomar como próximo JobTracker al primer elemento de la lista de hijos secuenciales del nodo */jobTracker/taskTrackers*.

Confirmada la falla del JobTracker con la no existencia del nodo */jobTracker/alive*, los TaskTrackers inician un proceso para obtener el primer nodo hijo del nodo */jobTracker/taskTrackers* y obtener el valor del nodo */jobTracker/alive*, si es que existe.

Cuando un TaskTracker obtiene el valor del primer nodo hijo, lo compara consigo mismo para verificar si se debe de transformar en el JobTracker o no. De ser el nuevo JobTracker, inicia por línea de comandos al JobTracker pasando como parámetros la opción `-recoverFromSnapshot` y los datos de conexión del ZooKeeper, y queda esperando en un bucle a que se cree correctamente el nodo *alive*.

En caso de que el TaskTracker no sea el próximo JobTracker, busca el valor del nodo *alive* directamente. De no existir este valor, vuelve a ejecutarse el bucle, verificando nuevamente los nodos hijos por si hubieron cambios. Cuando el TaskTracker detecta que existe el nodo *alive*, es poruqe el nuevo TaskTracker ya ha sido creado.

Una situación que puede suceder es que el JobTracker vuelva al sistema, ya sea porque su caída estuvo relacionada a un problema de red temporal o porque se haya reiniciado. En este caso, el JobTracker va a ser el que haya creado y configurado primero el nodo *alive*. Mientras el JobTracker está caído, el JobClient lo único que hace es chequear la creación del nodo *alive* hasta que la creación sea exitosa.

4.3.8. Recuperación del JobTracker

La recuperación del JobTracker se inicia cuando el nodo con el TaskTracker seleccionado para iniciar el nuevo JobTracker ejecuta el siguiente comando:

```
bash -c $HADOOP_PATH/hadoop-daemon.sh start jobtracker
-recoverFromSnapshot $ZOOKEEPER_NODES
```

$HADOOP_PATH es la ruta absoluta del directorio *bin* de Hadoop y $ZOOKEEPER_NODES es la línea de conexión para ZooKeeper, conteniendo una línea con los hostnames y puertos separados por comas que tienen ZooKeeper en ejecución (por ejemplo, `hadoop1:2181,hadoop2:2181,hadoop3:218`
Es necesario pasarle esta información al nuevo JobTracker de esta manera porque el nuevo JobTracker no debería usar los archivos de configuración del TaskTracker, ya que estos pueden ser distintos a los del JobTracker original.

Lo primero que hace el JobTracker es crear la conexión con ZooKeeper, utilizando los datos de conexión recibidos por línea de comandos. Luego comienza el proceso de inicialización y recuperación de los atributos del nuevo JobTracker. El constructor se compone de dos partes: la inicialización normal de ciertos atributos y la recuperación de atributos guardados en ZooKeeper. Para la primera acción, se copó el código usado en el constructor original del JobTracker, este código crea objetos como el `queueManager` y el `clock`, entre muchos otros, e inicia los threads de los distintos servicios. Para la segunda parte se realizan dos tareas: la recuperación y adaptación de los valores guardados en Zookeeper, y la reparación/regeneración inmediata de algunos de estos valores.

El problema que se tiene al realizar la serialización de los objetos es que muchos de éstos tienen referencias circulares. Las referencias circulares o cíclicas suceden cuando dos clases se referencian directa o indirectamente entre sí. Por ejemplo, la clase JobInProgress tiene como atributo al JobTracker, mientras que JobTracker tiene como atributo *jobs* una lista de JobInProgress. Al serializar un JobInProgress, se va a intentar serializar también el JobTracker, y éste va a intentar serializar el atributo *jobs* que contiene la colección de JobInProgress. Para solucionar las referencias circulares, se debe configurar el atributo jobTracker del JobInProgress como *transient*, para que no se incluya en la serialización del JobInProgress.

Al momento de recuperar estas referencias circulares, se debe de volver a configurar correctamente los atributos marcados como *transient*. En el caso de la recuperación de los `jobs`, una vez recuperado del ZooKeeper, para cada uno de los objetos JobInProgress se llama al método

`setRecoveredProperties`. Este método asigna el JobTracker, el JobConf y otros atributos que dependen de éstos, y luego deserializa cada uno de los objetos TaskInProgress que guarda internamente.

Otro problema que surge de la serialización y recuperación es la inconsistencia de objetos en distintos atributos serializados. Por ejemplo, por un lado se persiste el atributo `hostnameToNodeMap` que contiene los objetos `Node` asociados a cada hostname, mientras que, por otro lado, dentro de los JobInProgress se tienen los atributos `nonRunningMapCache` y `runningMapCache` que son maps y usan los `Node` como clave del mapa. Los `Node` se recuperan correctamente de forma independiente, el problema surge cuando se quiere buscar en los mapas con un `Node` del atributo `hostnameToNodeMap`. Como el `hostnameToNodeMap`, el `nonRunningMapCache` y el `runningMapCache` se recuperan independientemente, se generan objetos con hashCode distinto, lo que hace que la búsqueda por clave siempre devuelve vacío. Para solucionar este problema, se debe buscar por cada una de las claves el nuevo objeto y regenerar los mapas.

Además de la solución de modificar lo ya recuperado, en ciertos casos en que se tienen los mismos objetos en dos colecciones diferentes, se persiste el objeto real en una colección y se mantiene, persiste y recupera una referencia al objeto real en otros casos. Por ejemplo, el mapa `userToJobsMap` mantiene un listado de los JobsInProgress de cada usuario. En vez de persistir y recuperar el listado de los JobsInProgress, se crea y se mantiene una colección en paralelo con únicamente los `JobID` de los JobInProgress, la cual se persiste en cada modificación. Al momento de la recuperación, se regenera el mapa original usando el listado de `JobInProgress jobs` y los `JobID` que se guardaron.

Finalmente luego de la correcta recuperación de los valores, se crea el nodo ephemeral */jobTracker/alive* para notificarle a todos los dependientes del nuevo JobTracker que el mismo ya ha sido recuperado y está operando nuevamente.

4.3.9. Sincronización con los TaskTrackers y JobClient

Una vez que los TaskTrackers tienen la referencia del nuevo JobTracker, le envían el nuevo Heartbeat con los estados de las últimas tareas realizadas, incluida la información que inicialmente se intentó enviar cuando se detectó la falla. El JobTracker recibe esta información y, al tener los datos de los últimos heartbeat recibidos por cada taskTracker que se guardaron de manera sincrónica, responde de forma natural y como si

nunca hubiera sucedido un fallo.

De parte del JobClient, al volver a tener la conexión con el JobTracker, le pide normalmente un JobStatus y, obteniendo la información del JobInProgress, el JobTracker le devuelve el valor actualizado al JobClient.

4.4. Descripción de la implementación

En esta sección se comenta brevemente la clase ZooKeeperManager que se encarga de mantener y manejar la conexión con ZooKeeper.

4.4.1. ZooKeeperManager

La clase ZooKeeperManager es la encargada de toda la comunicación con el ZooKeeper, facilitando y simplificando la interacción utilizando distintos métodos. ZooKeeperManager es la única clase, no interna, creada en la implementación realizada en el proyecto y es usada por el JobTracker, los TaskTrackers y los JobClients.

Esta clase a su vez usa la API de Zookeeper `org.apache.zookeeper`, la cual se descarga automáticamente en el build al ser agrega como dependencia en el archivo de configuración de Apache Ivy, el manejador de dependencias utilizado por Apache Hadoop. La clase ZooKeeperManager mantiene la conexión con ZooKeeper, interpreta los errores de la API oficial y maneja los reintentos cuando es necesario antes de lanzar excepciones.

Los métodos utilizados son:

- `void create(String path, byte[] data)`:
  Crea un nodo en la ruta `path` con el valor `data`.

- `void write(String path, byte[] data)`:
  Escribe un nodo en la ruta `path` con el valor `data`. Si el nodo en esa ruta no existe entonces falla.

- `byte[] read(String path)`:
  Lee el valor en el nodo de ruta `path`.

- `byte[] getFirstOrderedChildren(String path, String prefix)`:
  Devuelve el valor del primer nodo hijo donde los hijos son nodos secuenciales usando el prefijo `prefix`.

- `String createEphemeral(String path, byte[] data)`:
  Crea un nodo ephemeral como hijo del nodo en la ruta `path`, conteniendo los datos `data`. Se utiliza, por ejemplo, para crear el nodo */jobTracker/alive*.

- `String createSeqEph(String path, byte[] data)`:
  Crea un nodo secuencial ephemeral como hijo del nodo en la ruta `path` con el dato `data`. Se utiliza para la creación de los nodos ephemerales de los TaskTrackers en la ruta */jobTracker/taskTrackers* que eventualmente se utiliza para la selección el próximo JobTracker.

Un ejemplo de cómo utilizar estos métodos puede verse en el siguiente extracto de código de la inicialización del JobTracker. Se instancia el ZooKeeperManager usando los datos de configuración del sistema. Se crea el directorio raíz utilizado por Hadoop (por defecto */jobTracker*), y se crea el nodo ephemeral */jobTracker/alive*. De esta manera se simplifica enormemente los problemas de conexión, reintentos que dependen del tipo de error de ZooKeeper, cantidad de reintentos posibles, interpretación de errores, y logueo correspondiente.

```
// start the ZooKeeper Manager
zooKeeperManager = new
    ZooKeeperManager(conf.get("mapred.zookeeper.nodes",
    "127.0.0.1:2181"));

// Creating main dir if not created.
zooKeeperManager.write(zkTrackerNode, "".getBytes());

// Created alive ephemeral node.
zooKeeperManager.createEphemeral(zkTrackerNode + "/alive",
    this.getRealHostname().getBytes());
```

Listado 4.2: Ejemplo uso de ZooKeeperManager

Capítulo 5

Análisis Experimental

En este capítulo se comienza describiendo los problemas que hubo que afrontar a la hora de realizar pruebas, tanto durante el desarrollo como durante las pruebas finales, y las herramientas utilizadas y desarrolladas para afrontarlos. Luego se detallan las pruebas realizadas para validar la implementación y se realiza un análisis de las mismas.

5.1. Introducción

El desarrollo de aplicaciones distribuidas es una tarea difícil. Se requiere mucho esfuerzo mental para entender cómo se comporta y cómo puede fallar cada componente de la aplicación de forma independiente, lo que hace que este tipo de aplicaciones sean más difíciles para diseñar y desarrollar que otras. La dificultad se incrementa por el hecho de que este tipo de aplicaciones requiere de mucha preparación y configuración para ser probadas y testeadas en un ambiente similar al de producción final, y no permiten un testeo funcional rápido e iterativo.

En el caso del desarrollo de la tolerancia a fallos del JobTracker, no solo se debe probar que todo funcione correctamente, sino también se tiene que poder "hacer caer" al JobTracker de forma manual, y comprobar el funcionamiento del resto de los nodos en el escenario de falla.

Para solucionar estos problemas, se decidió recurrir a la virtualización. Al utilizar máquinas virtuales, las características de cada nodo pueden ser abstraídas de manera de replicar su funcionamiento y simplificar la configuración de aplicaciones distribuidas. Adicionalmente, algunas funcionalidades como la capacidad de pausar/resumir y la creación de checkpoints pueden resultar sumamente valiosas. Naturalmente, las aplicaciones ejecutando en máquinas virtuales no siempre alcanzan el desempeño de las

mismas ejecutando directamente en dispositivos físicos. Se han hecho importantes avances en este sentido, y dependiendo de la estrategia de virtualización (hipervisores (hypervisors) vs. contenedores (containers)), el tipo de CPU, y del resto del hardware se han alcanzado niveles de desempeño muy cercanos al de las máquinas físicas [54].

Para poder probar Hadoop de manera de que permitiera un prototipado rápido y poder simular distintos escenarios de tolerancia a fallos, se evaluaron distintas estrategias de virtualización, que se describen a continuación.

5.2. Estrategias de virtualización

Las técnicas de virtualización son uno de los principales componentes de *cloud computing*. Aunque son utilizadas desde hace ya varias décadas, varias soluciones como XEN [5], VMWare [52] o KVM [20] y soporte de hardware (Intel-VT and AMD-V) han convertido a la virtualización en un elemento esencial en las infraestructuras cloud.

La virtualización en base a hypervisor consiste en un sistema operativo cliente ejecutando sobre el sistema operativo anfitrión, de manera de proveer una abstracción total del sistema operativo (véase la figura 5.1) donde cada sistema operativo se ejecuta de forma independiente del resto. Los beneficios de esta arquitectura incluye la independencia sobre el hardware utilizado, la aislación creada, y la posibilidad de ejecutar distintos sistema operativos en un host.

A pesar de estas ventajas, la virtualización en base a hypervisores se considera de peso pesado, ya que ese nivel de abstracción genera un overhead importante, especialmente si las aplicaciones son intensivas desde el punto de vista de I/O. A su vez, la cantidad de instancias que es posible ejecutar está limitada por la cantidad de recursos (disco, memoria, etc) del host de base, ya que se debe proveer suficiente memoria tanto a los sistemas operativos guest como al host, los cuales tienden a ocupar en el orden de cientos de MB o varios GB.

Por causa de estas desventajas se han realizado distintos esfuerzos para la implementación de la virtualización en base a contenedores. Este enfoque depende de que el sistema operativo facilite la partición de los recursos de manera de crear distintas instancias aisladas sobre kernel del host.

Estas instancias trabajan a nivel del sistema operativo, todas ellas compartiendo el kernel del sistema operativo. Obviamente, esta arquitectura proporciona una aislación más débil en comparación con la vir-

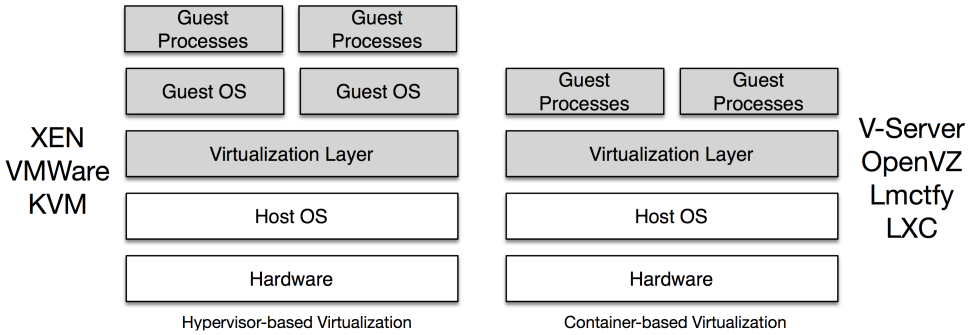

Figura 5.1: Comparación de virtualización basada en hypervisor (izquierda) y basada en contenedores (derecha)

tualización en base a hypervisors, pero mucho trabajo se ha hecho y se sigue realizando para mejorar los niveles de aislación [23] [43].

El soporte de contenedores depende en los servicios del sistema operativo host (BSD jails, Solaris Zones, Linux namespaces/cgroups [34]), aunque todos ellos dependen de dos características: (*i*) un mecanismo de administración del espacio de nombres que permite que dos procesos tengan distinta visibilidad del sistema y (*ii*) un mecanismo de manejo de recursos que permita la restricción de uso de CPU, memoria, etc. a los distintos contenedores.

Linux presenta varios sistemas de contenedores como por ejemplo Linux-VServer [13], OpenVZ [37], Google Lmctfy [31] o LXC [22]. Todos estos sistemas proveen manejo de bajo nivel de virtualización de contenedores y son principalmente responsables para el manejo de los namespaces y la aislación.

Linux-VServer es uno de los sistemas más antiguos de contenedores de Linux. Depende de herramientas tradicionales de Linux como son *chroot* y *rlimit* para proveer manejo de aislación básico. También provee aislación de procesos, CPU y red a través de un patch al kernel.

Más recientemente, OpenVZ y LXC dependen de nuevas funcionales del kernel de Linux como son *namespaces* y *cgroups*. Ambos sistemas usan namespaces para la aislación de los contenedores y para asegurarse de que solo puedan acceder al subconjunto de recursos asignados. Como cada namespace tiene su propio espacio de PID, a los contenedores se les pueden realizar checkpoints, pausar y resumir, e incluso migrar. Una de las diferencias entre estos sistemas es en la estrategia del manejo de los recursos: OpenVZ desarrolló su propio manejo de recursos que controla la administración del CPU y la cuota del disco mientras que LXC depende en la funcionalidad *cgroups* para configurar y limitar el uso de CPU y los

demás recursos.

Tanto OpenVZ como LCX son manejadores de bajo nivel que proveen únicamente las funcionalidades básicas, y en ambos casos han sido extendidos para proveer un manejo con funcionalidades más complejos para proveer el despliegue, configuración y migración, entre otras. OpenVZ es la base de la solución comercial de virtualización Parallels Cloud Server [1]. LXC es la base de Docker [25], un manejador de contenedores LXC open-source creado originalmente por el proveedor de PaaS (Platform as a Service) dotCloud como un proyecto interno.

5.3. Docker-Hadoop

Hadoop se puede ejecutar de dos modos: single node o multi-node. En modo single node, se ejecuta el JobTracker, NameNode, TaskTracker y DataNode en un mismo servidor. Este modo sirve para probar rápidamente el comportamiento de una aplicación, pero es insuficiente para probar fallos en el JobTracker, ya que parte de lo desarrollado involucra, entre otras características, que los nodos tengan distintas direcciones IP.

Entonces, para probar la falla del JobTracker se necesita poder ejecutar en modo multi-node. Para ejecutar en modo multi-node o se tienen varias computadoras disponibles o se ejecuta utilizando máquinas virtuales.

En el contexto del proyecto, se propuso la utilización de Docker para el prototipado rápido e iterativo.

5.3.1. Docker

Docker [25], como se mencionó anteriormente, es un software open-source que sirve para automatizar la ejecución de aplicaciones usando LXC.

Docker provee una simple API para poder crear y configurar un contenedor que fácilmente se pueda ejecutar en varias instancias y por distintos servidores, creando un ambiente simple de replicar.

5.3.2. Uso de Docker para ejecución multi-node local

Se propone usar Docker para la creación de una imagen lista para instanciarse tanto como una máquina ejecutando JobTracker, o cualquiera de los servicios de Hadoop.

En la instanciación de la imagen se especificará el hostname, la IP y qué servicios debe ejecutar e iniciar automáticamente. De esta forma, se pueden ejecutar localmente varias instancias con un overhead mínimo.

Cada instancia deberá además, usar un directorio compartido para acceder al último build de Hadoop, sin tener que recrear la imagen nuevamente para cada modificación del código de Hadoop.

5.3.3. Docker-hadoop

Utilizando Docker se creó un *Dockerfile*, archivo que encapsula la creación y configuración del ambiente necesario para ejecutar Hadoop y ZooKeeper. Este archivo permite replicar la imagen final que se utiliza para una distribucion más fácil.

Dentro de este archivo se realizan las siguientes configuraciones:

- Instalación de Java 6 y otras herramientas necesarias (ssh, curl, etc)
- Creación de usuario e instalación de claves pública/privada para poder conectarse sin problemas a los contenedores a través de ssh.
- Configuración de carpetas compartidas para que todas las instancias accedan a la misma versión (modificada) de Hadoop.

Al ejecutarse el constructor del Dockerfile, éste inicia el demonio de ssh, dejando habilitado el inicio de los procesos a través de este medio.

Para facilitar el manejo de instanciación y manipulación de las instancias, se creó una librería en Python que abstrae al usuario final de los detalles de inicio de servicios y configuraciones de red entre otros detalles.

Usando la información que provee la API de Docker, y guardando otros datos específicos en una base de datos local, la biblioteca en Python creada provee las siguientes primitivas:

- list_cluster
- create_cluster
- container_detail
- start_container
- start_service
- stop_container

Esta biblioteca es usada por una aplicación independiente que pública una página web para la visualización e interacción con el cluster. Transformando las primitivas de la biblioteca en una API que maneja mensajes en formato JSON, fue posible desarrollar una visualización dinámica del cluster a través de una página. Desde esta página es posible:

- Configurar los hostname e IP del cluster a crear (Figura 5.2)
- Determinar la cantidad de nodos y que servicio debe ejecutar cada uno (Figura 5.2)
- Iniciar el cluster y monitorear el status de cada nodo (Figura 5.3)
- Visualizar en tiempo real los logs de cada servicio de Hadoop (Figura 5.4)
- Detener nodos de forma arbitraria (Figura 5.3)

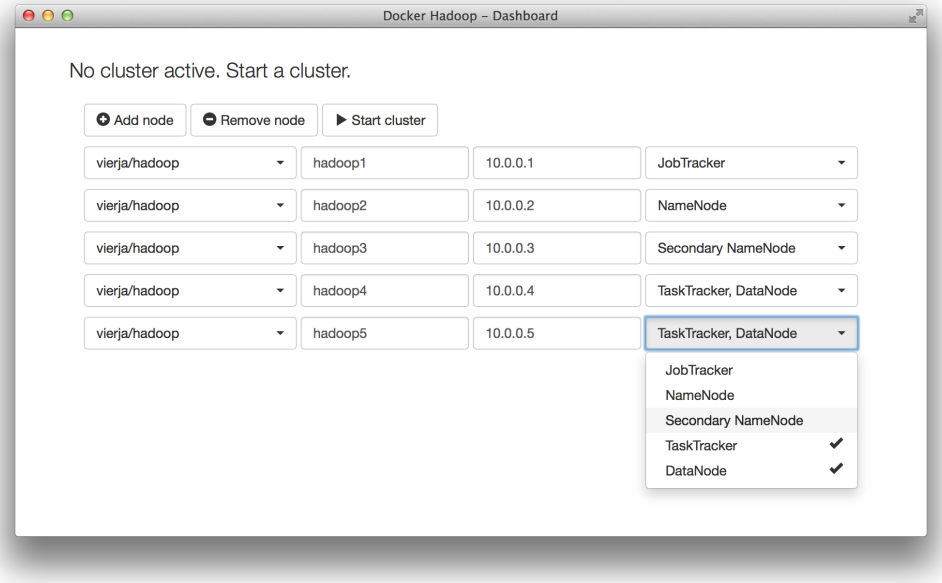

Figura 5.2: Vista inicial de configuración del cluster

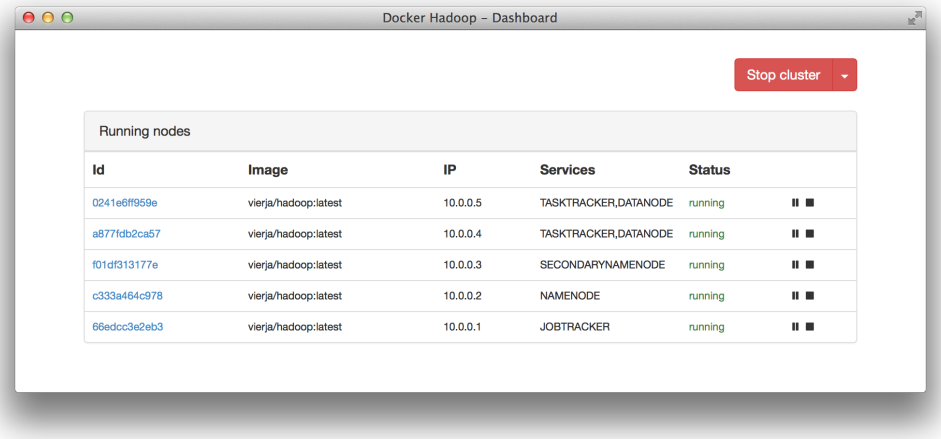

Figura 5.3: Monitoreo del cluster en ejecución

Figura 5.4: Log de un nodo.

La alternativa a ver los logs en la página, es entrar por *ssh* a la instancia de la imagen de forma manual. Al configurar las IPs de los Docker, automáticamente se crea una interfaz en la máquina ejecutando la aplicación con una IP en la misma subred.

Para la simulación de un nodo caído se puede hacer un stop al contenedor correspondiente, y este es inmediatamente detenido y eliminado. Este funcionamiento es gracias a que Docker permite detener el contenedor virtual como si fuera un proceso ejecutando al cual se le aplica una señal de finalización.

Una de las desventajas frente a la virtualización de máquinas es que no se puede suspender la ejecución para volver a ejecutarla a partir del mismo estado más adelante.

5.4. Pruebas realizadas

Dada la complejidad de realizar las pruebas en un cluster real con distintos nodos físicos, y escasez de recursos disponibles, se decidió realizar las pruebas utilizando Docker-Hadoop en servidores virtuales.

Se decidió utilizar cinco nodos de ZooKeeper ejecutando en los primeros cinco nodos creados por Docker-Hadoop.

Se decidió utilizar 5 nodos de ZooKeeper corriendo en los primeros 5 nodos.

5.4.1. Ambiente de pruebas

Para la realización de las pruebas se decidió utilizar DigitalOcean. DigitalOcean es un proveedor de Virtual Private Servers (VPS) que permite rápidamente la creación y destrucción de servidores virtuales. Con una interfaz muy simple y fácil de usar, DigitalOcean ofrece un sistema de clonado de servidores virtuales, lo cual permitió la creación de una imagen pre-configurada con Docker y Hadoop, facilitando la creación de nuevas instancias para testar. Este entorno integrado permitió probar la implementación de Docker-Hadoop en varios VPS en paralelo de forma ágil.

DigitalOcean cuenta con varias configuraciones de servidores virtuales que varían en cantidad de CPUs, memoria RAM y espacio de almacenamiento. Todas las instancias usan virtualización por KVM. Se decidió utilizar la configuración más potente disponible, cuyos detalles se presentan en la Tabla 5.1

Características	Valor
Número de CPUs	20
Memoria RAM	64 GB
Almacenamiento	640 GB SSD
Sistema operativo	Ubuntu 13.10

Tabla 5.1: Características del ambiente de pruebas.

El cluster de Hadoop se configuró con un valor de replicación (*dfs.replication* en el archivo *conf/hdfs-site.xml*) de 3. Como tamaño de bloque se utiliza un valor de 64 MB, el valor por defecto.

Para las pruebas se utilizó el *WordCount*, un simple programa de Hadoop que se utiliza como ejemplo en muchos tutoriales de Hadoop y que fue mencionado en la sección 2.3. El programa *WordCount* recibe un texto guardado en el HDFS y retorna la cantidad de veces que aparece cada palabra en el texto. El código está implementado en Java y cuenta con dos funciones además del *main*, que son la función de *Map* y la función de *Reduce*. La función *Map* separa el texto recibido en palabras y va guardando el par <palabra, 1> en *output*, donde el 1 indica que la palabra se encontró una vez. La función Reduce recorre los pares <palabra, 1> obtenidos en el Map y va sumando la cantidad de repeticiones de cada palabra. En el *main* se asignan en la configuración de Hadoop los parámetros necesarios para que pueda ejecutar el WordCount, como cuáles son las clases de Map y Reduce y la ubicación de los archivos de entrada y salida. El código de *WordCount* se presenta en el Listado 5.1.

```
public class WordCount implements Serializable {
    public static class Map extends MapReduceBase implements
        Mapper<LongWritable, Text, Text, IntWritable> {
      private final static IntWritable one = new IntWritable(1);
      private Text word = new Text();

      public void map(LongWritable key, Text value,
          OutputCollector<Text, IntWritable> output, Reporter
          reporter) throws IOException {
        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line);
        while (tokenizer.hasMoreTokens()) {
          word.set(tokenizer.nextToken());
          output.collect(word, one);
        }
```

```
    }
  }

  public static class Reduce extends MapReduceBase implements
      Reducer<Text, IntWritable, Text, IntWritable> {
    public void reduce(Text key, Iterator<IntWritable>
        values, OutputCollector<Text, IntWritable> output,
        Reporter reporter) throws IOException {
      int sum = 0;
      while (values.hasNext()) {
        sum += values.next().get();
      }
      output.collect(key, new IntWritable(sum));
    }
  }

  public static void main(String[] args) throws Exception {
    JobConf conf = new JobConf(WordCount.class);
    conf.setJobName("wordcount");

    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(IntWritable.class);

    conf.setMapperClass(Map.class);
    conf.setCombinerClass(Reduce.class);
    conf.setReducerClass(Reduce.class);

    conf.setInputFormat(TextInputFormat.class);
    conf.setOutputFormat(TextOutputFormat.class);

    FileInputFormat.setInputPaths(conf, new Path(args[0]));
    FileOutputFormat.setOutputPath(conf, new Path(args[1]));

    JobClient.runJob(conf);
  }
}
```

Listado 5.1: Código de la clase WordCount.

5.4.2. Pruebas de reconocimiento del ambiente

Se decidió comenzar realizando pruebas aumentando gradualmente la cantidad de nodos TaskTrackers, manteniendo constante la cantidad de

datos de entrada en 10 GB, a modo de determinar los límites prácticos del ambiente de pruebas utilizado. La máxima cantidad de TaskTracker utilizados fue veinte, ya que a partir de los siete TaskTracker no se observó mejora en el tiempo total de ejecución. En la tabla 5.2 se pueden ver los resultados obtenidos.

Cantidad de TaskTrackers	Tiempo de ejecución (s)
1	2750
2	1513
3	1141
5	1021
7	922
10	1100
15	1053
20	1194

Tabla 5.2: Tiempo de ejecución diferentes números de TaskTrackers.

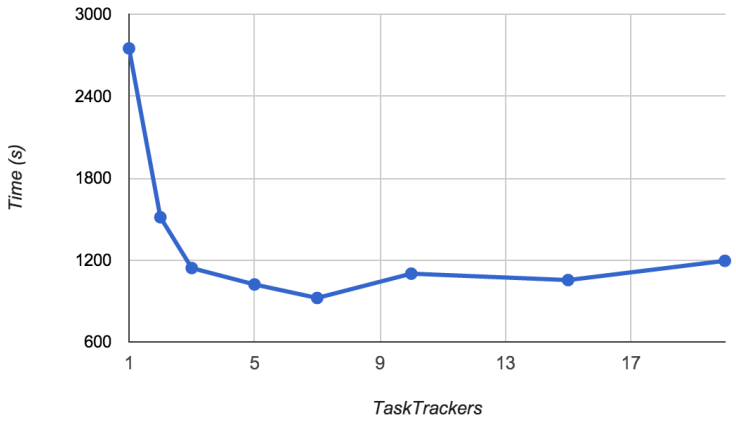

Figura 5.5: Tiempo de ejecución al variar la cantidad de TaskTrackers.

A partir de los datos obtenidos se calculó el speedup (algorítmico)[1] al variar la cantidad de TaskTrackers. El speedup es utilizado para medir la mejora de rendimiento de una aplicación al aumentar la cantidad de procesadores (o en este caso nodos), comparándola con el rendimiento de la misma aplicación usando un único procesador (o nodo). Cómo se

[1]Relación entre el tiempo de ejecución al utilizar un solo TaskTracker y el tiempo de ejecución al utilizar varios TaskTrackers.

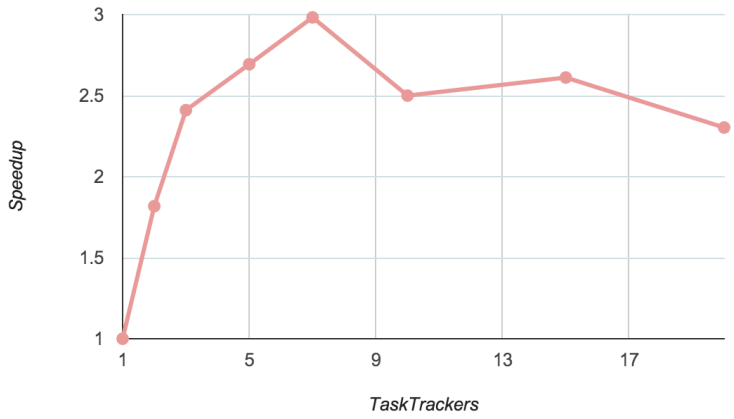

Figura 5.6: Speedup al aumentar la cantidad de TaskTrackers.

puede ver en las Figuras 5.5 y 5.6, se consigue una mejora hasta llegar a los 7 TaskTrackers, después de eso el speedup se mantiene constante.

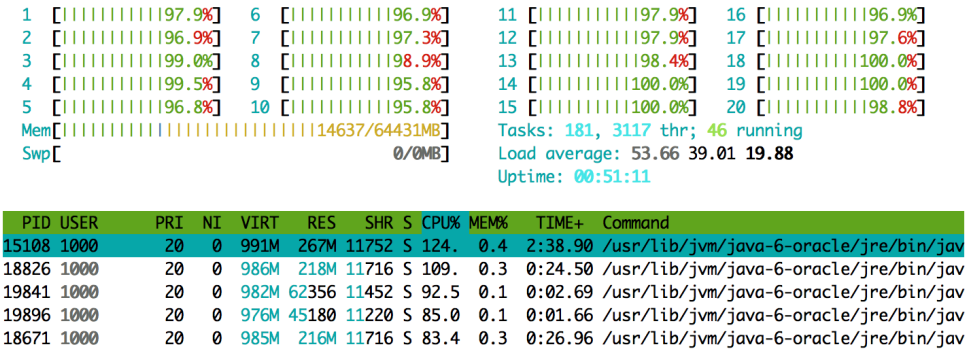

Figura 5.7: Comando *htop* durante la ejecución de un job.

La disminución en el incremento del speedup se puede deber a distintos cuellos de botella del sistema. Con 7 nodos TaskTracker/NameNodes más el JobTracker, NameNode, SecondaryNameNode, JobClient y 5 procesos ZooKeeper se tiene un total de 23 máquinas virtuales Java en ejecución. Utilizando el comando *htop* [36] durante la ejecución se pudo observar que se alcanzaba una carga (instantánea, promedio de 5 minutos y promedio de 15 minutos) varias veces mayor a 20 al utilizar más de 7 nodos TaskTrackers. La carga del sistema es una unidad para medir la cantidad de trabajo de CPU que se realiza. Un nodo desocupado tiene una carga de 0, mientras que cada proceso usando o esperando el uso del CPU aumenta el valor de la carga en 1. Dado que la cantidad

de procesadores que del sistema utilizado para estas pruebas fue de 20 CPUs y la carga medida durante las pruebas fue mayor de 20, está claro que el cuello de botella fue el CPU del sistema.

Por esta razón, se decidió realizar el resto de las pruebas utilizando entre 1 y 7 TaskTrackers.

5.4.3. Comparación con Hadoop sin modificar

Se decidió realizar pruebas para comparar el tiempo de ejecución de la versión de Hadoop 1.0.4 sin modificar con la versión modificada de Hadoop incluyendo tolerancia a fallos, con el fin de identificar un posible overhead en la nueva versión.

En las pruebas se utilizó un archivo de texto de 10 GB y se fue variando la cantidad de TaskTrackers. En la tabla 5.3 se pueden ver los resultados.

TaskTrackers	**Hadoop 1.0.4**	**Hadoop PERMARE**	**Overhead (%)**
2	1469	1534	4.42
3	1109	1176	6.04
5	1137	1110	-2.37
7	888	879	-1.01

Tabla 5.3: Comparación del tiempo de ejecución sin fallas en segundos de la versión de Hadoop 1.0.4 sin modificar y la versión modificada.

En la tabla 5.3 se puede observar que al utilizar 5 y 7 TaskTrackers el tiempo de ejecución disminuyó, esta disminución en el tiempo de ejecución puede deberse a que el tiempo de ejecución de Hadoop varía en algunos segundos al realizar varias veces la misma prueba debido a factores que no podemos controlar, como por ejemplo la distribución de los datos del HDFS. Si se aumentase la cantidad de ejecuciones de la muestra, se debería obtener un promedio de tiempo de ejecución mayor para la versión modificada de Hadoop que para la versión sin modificar, ya que la versión modificada debe realizar más trabajo computacional para guardar los snapshots que la versión original de Hadoop.

A partir de los datos obtenidos se calculó el overhead para cada cantidad de TaskTrackers (ver tabla 5.3). El promedio obtenido fue de 1.77 % lo que es significativamente bajo.

5.4.4. Recuperación de fallas

Una vez validado que el overhead de la versión modificada de Hadoop es mínimo, se decidió controlar cuánto tiempo demora el JobTracker en recuperarse de una falla, este tiempo se denomina downtime. Con este objetivo, se realizaron pruebas variando el momento de la falla y manteniendo constante la cantidad de datos de entrada (10 GB) y la cantidad de TaskTrackers (5). Los resultados se pueden observar en la Tabla 5.4.

% Map	% Reduce	Tiempo hasta falla	JobTracker downtime	Tiempo total de ejecución
5	0	51	104	1135
6	0	69	66	967
10	1	87	73	1023
10	1	91	79	1018
18	4	143	137	1060
20	4	156	86	1073
30	8	208	84	1073
41	12	313	80	1104
50	14	369	101	1126
73	22	438	79	911
91	29	525	86	978

Tabla 5.4: Tiempos de recuperación del JobTracker con 10 GB de datos de entrada y 5 TaskTrackers.

Para los datos obtenidos se calculó el porcentaje de tiempo que el JobTracker está caído, obteniéndose en promedio un 8.48 % de downtime del JobTracker, respecto al tiempo total de ejecución. Cabe señalar que en estas pruebas el tiempo de ejecución fue muy corto en comparación a los tiempos de ejecución de Hadoop resolviendo un problema real, donde la ejecución puede durar horas o días. Por lo tanto, el peso del JobTracker downtime en el tiempo total de ejecución debería disminuir al utilizar Hadoop en la práctica. Por ejemplo, si se utilizase Hadoop para ejecutar un trabajo que demora dos días en ejecutarse, el tiempo total de ejecución sin fallas en el JobTracker en segundos sería `2*24*60*60=172800 seg`, utilizando un tiempo de downtime del JobTracker por falla de 88.64 segundos (promedio de downtime del JobTracker obtenido a partir de los datos de la Tabla 5.4), una falla significa un 0,05 % de la ejecución total. En este ejemplo, deberían producirse 20 fallas en el transcurso de la ejecución para que el downtime del JobTracker representase el 1 % del total de la ejecución y 217 fallas para llegar a un downtime del JobTracker del

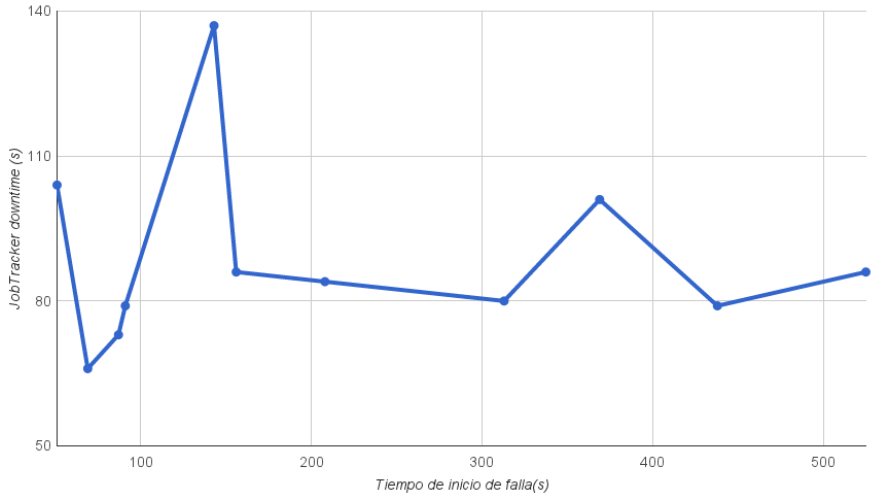

Figura 5.8: Downtime del JobTracker en función del momento en que se produce la falla.

10 %. Teniendo en cuenta que 217 fallas en el JobTracker en dos días de ejecución representan más de una falla cada 15 minutos, en un ambiente con tantas fallas en el JobTracker (que por el mecanismo de elección visto en la sección 4.3.7 es uno de los más estables del sistema) habría que comenzar a analizar si ese es el ambiente correcto para ejecutar Hadoop antes de preocuparse por el downtime del JobTracker.

La gráfica 5.8 muestra cómo varía el downtime del JobTracker en función del momento en el que se produce la falla. En esta gráfica no se aprecia una relación entre ambas variables, incluso se puede ver que el downtime del JobTracker se mantiene relativamente constante, teniendo un promedio de 88.64 segundos. Por lo tanto se puede concluir que no existe una relación entre el momento de la falla y el tiempo que el JobTracker está caído. Esto es de suma importancia, ya que si el downtime del JobTracker aumentase cuanto más tarde se produjese la falla, al producirse una falla cerca del final de la ejecución el tiempo de recuperación podría ser muy grande, lo que degrada el desempeño del sistema.

El promedio de 88.64 segundos de downtime del JobTracker fue obtenido utilizando 5 TaskTrackers. Debido a que la cantidad de datos a cargar de cada TaskTracker es siempre la misma, al utilizar más TaskTrackers este tiempo promedio aumentará de forma lineal. Lo mismo ocurre con la cantidad de tareas que se lleva a cabo en los TaskTrackers cuando el JobTracker falla, los datos que se guardan en el snapshot son

los mismos sin importar el número de tareas. Al aumentar la cantidad de tareas, el downtime promedio del JobTracker aumenta linealmente.

5.4.5. Múltiples errores

Uno de los objetivos fue que el sistema se pudiera recuperar de múltiples fallas consecutivas en el JobTracker. Para verificar este objetivo se probó hacer fallar el JobTracker tres veces durante la misma ejecución, obteniéndose los resultados que se pueden ver en la tabla 5.5.

# Falla	% Map	% Reduce	Tiempo hasta falla	JobTracker downtime
1	8	0	71	105
2	62	18	466	62
3	83	26	623	87

Tabla 5.5: Tiempos de recuperación del JobTracker con 10 GB de datos de entrada, 7 TaskTrackers en ejecución y múltiples fallas.

El resultado de estas pruebas fue exitoso, ya que no sólo el JobTracker se recuperó de forma correcta luego de las tres fallas, sino que además no se observa una relación entre el tiempo de recuperación del JobTracker y la cantidad de fallas previas. Este comportamiento significa que el downtime del JobTracker no depende de la historia previa de fallas.

5.5. Resumen del análisis experimental

A partir del análisis experimental realizado fue posible validar la versión modificada de Hadoop con soporte para tolerancia a fallos.

- Se confirmó la eficiencia del diseño, obteniéndose un overhead del 1.77 % en comparación con la versión sin modificar de Hadoop.
- Utilizando 5 TasktTrackers y 10 GB de datos de entrada, se calculó el tiempo que el JobTracker demora en recuperarse luego de una falla. Se obtuvo un tiempo de recuperación promedio de 88.64 segundos, lo que representa un 8.48 % del tiempo de ejecución total. Este tiempo no varía al cambiar el momento en que se produce la falla. Estos resultados validan que el tiempo de recuperación del JobTracker no tiene un gran impacto en el tiempo total de ejecución. Incluso el peso del downtime del JobTracker en el tiempo total de ejecución debería disminuir al atacar problemas qué requieran mayor tiempo de ejecución.

- Se validó que el sistema puede recuperarse de múltiples fallos en el JobTracker durante la misma ejecución.
- En las últimas pruebas se verificó que el tiempo de recuperación del JobTracker no aumenta al aumentar la cantidad de fallas, lo que en caso contrario causa que el desempeño del sistema disminuya cuando el JobTracker falla repetidas veces.

En resumen, todas las pruebas realizadas validan la solución alcanzada, indicando también que la pérdida de rendimiento debido a los mecanismos implementados para la generación del snapshot es mínimo.

Capítulo 6

Conclusiones y trabajo futuro

Esta sección presenta las conclusiones de este proyecto, sus aportes en diferentes áreas y los trabajos a futuro que se desprenden del proyecto.

6.1. Conclusiones

Este trabajo ha presentado los esfuerzos realizados para lograr una versión de Hadoop tolerante a fallos en el JobTracker, cumpliendo con las expectativas que el proyecto PERMARE puso sobre este proyecto de grado. La versión de Hadoop realizada en este proyecto cumplió exitosamente con todas las pruebas realizadas para comprobar su tolerancia a fallos en el JobTracker, recuperándose incluso de más de un fallo durante la misma ejecución. Esta tolerancia a fallos se logró con un overhead de 1.77 % respecto de la versión sin modificar de Hadoop, este valor es muy bueno, sobre todo si se considera que si en la versión original de Hadoop falla el JobTracker, hay que comenzar la ejecución de los trabajos desde cero. Por otro lado, este proyecto de grado aporta una herramienta para la simulación y el manejo de clusters, que automatiza la instalación y la puesta en marcha de Hadoop y también permite detener cualquier nodo del sistema, lo que es fundamental para testear la tolerancia a fallos en Hadoop.

El meanismo de tolerancia a fallos implementado se basa en que el manejador de trabajos (JobTracker) realice snapshots de su estado periódicamente y también en algunos eventos especiales. De esta forma, cuando el JobTracker falla, uno de los manejadores de tareas (TaskTrackers) es designado como el nuevo JobTracker, quién utiliza el último snapshot generado para recuperar el estado del JobTracker original.

A partir del análisis experimental, se determinó que la solución ob-

tenida presenta un mínimo overhead en comparación con la versión sin modificar de Hadoop, del orden del 1.77 %, como se dijo anteriormente. Esto hace que esta nueva versión de Hadoop pueda ser utilizada tanto en ambientes altamente propensos a fallas como en ambientes donde las fallas son escasas, ya que el overhead es mínimo y se tiene la ventaja de estar utilizando una versión tolerante a fallos en el JobTracker.

El análisis experimental mostró que las tareas continúan su ejecución aún después de fallar el JobTracker en repetidas oportunidades durante la misma ejecución. El tiempo de recuperación promedio del JobTracker fue de alrededor de un minuto y medio. Este tiempo no depende del momento en el que se produce la falla, ya que el tiempo de recuperación luego de cada falla varió muy poco en el transcurso de las pruebas. Es importante mencionar que, aunque durante este tiempo de recuperación no hay un JobTracker manejando los trabajos, las tareas ya asignadas se siguen ejecutando y el cluster sigue en funcionamiento.

Por otro lado, se desarrolló de forma exitosa Docker-Hadoop, una herramienta que permite simular y manejar un cluster en una única máquina en un ambiente de pruebas, o incluso desplegar cientos de instancias de Hadoop en un cluster o en la nube. El uso de Docker-Hadoop simplifica ampliamente la realización de pruebas, ya que la instalación y puesta en marcha de Hadoop se realiza prácticamente de forma automática, necesitando solamente un par de clicks para elegir la cantidad de TaskTrackers a utilizar e iniciar el cluster. Docker-Hadoop también permite detener cualquier nodo, lo que permitió detener al JobTracker para simular las fallas en diferentes escenarios.

Este proyecto de grado cumplió con los objetivos asignados dentro del proyecto PERMARE, desarrollando una versión de Hadoop tolerante a fallos que hará posible el desarrollo de una versión de Hadoop que pueda ser ejecutada en entornos pervasivos al ser integrada a los otros componentes del proyecto PERMARE. También se realizó un aporte extra al proyecto PERMARE que fue el desarrollo de Docker-Hadoop, una herramienta que puede ayudar a los demás investigadores relacionados al proyecto a testear y validar sus implementaciones, disminuyendo considerablemente el tiempo de desarrollo y experimentación.

También se realiza un gran aporte a la comunidad de Hadoop en general, ya que la mayoría de los trabajos relacionados con la tolerancia a fallos en Hadoop se centran en el HDFS, existiendo poca bibliografía relacionada a la tolerancia a fallos en el JobTracker que es un SPOF de Hadoop. Si bien se han propuesto versiones tolerantes a fallos en el JobTracker, la versión más estable es de código cerrado y las demás to-

davía están en etapa de investigación y testeo o tienen cosas que mejorar. La versión desarrollada en este proyecto introduce un overhead mínimo en relación a la versión original de Hadoop y además soporta múltiples fallas en el JobTracker, teniendo un downtime del JobTracker bastante pequeño en las pruebas realizadas.

6.2. Trabajo futuro

A partir del trabajo realizado surgen dos líneas principales para expandir la investigación en el futuro: la relacionada a las modificaciones de Hadoop dentro del contexto del proyecto PERMARE y la vinculada con la plataforma de testeo en base a contenedores.

Con respecto a las modificaciones de Hadoop, debe realizarse una mejor integración con ZooKeeper al momento de hacer un despligue en un cluster. En especial debe permitirse iniciar ZooKeeper de forma integrada y automática al iniciar los distintos nodos. Hasta el momento esto se realizó de forma manual o con Docker-Hadoop, pero se debe facilitar el procedimiento de inicio para permitir realizar un despliegue simple en los distintos entornos que permite Hadoop.

Otra de las líneas de trabajo relacionadas a Hadoop consiste en actualizar la nueva versión 2 de Hadoop con los cambios realizados en este proyecto. Una vez validada la idea con Hadoop 1, se podrían aplicar los conceptos de snapshotting y recuperación en base de ZooKeeper para el ResourceManager y el NodeManager, los reemplazos del JobTracker en la versión 2.

Por otra parte, en Docker-Hadoop existen varios puntos para abordar como trabajo futuro. El más importante es permitir la creación y modificación de condiciones de red de forma artificial, ya sea agregando condiciones de atraso, pérdida y repetición de paquetes, o hasta la creación de particiones de red, aislando cierto nodo o conjuntos de nodos del resto para evaluar el comportamiento.

También relacionado a Docker-Hadoop puede ser interesante dar un paso más y permitir ejecutar cualquier tipo de imágenes de forma independiente. La aplicación hoy en día se encuentra muy ligada con Hadoop, pero sería posible abstraerse de Hadoop y permitir, de forma simple, la ejecución de cualquier sistema distribuido. Una versión como ésta sería un gran aporte ya que cualquier investigador podría usarla en sus poryectos de comuptación distribuída.

6.3. Trabajos publicados

En el marco del proyecto PERMARE, se han publicado los siguientes artículos científicos que incluyen desarrollos y resultados obtenidos en este proyecto:

- J. Rey, M. Cogorno, S. Nesmachnow, and L. A. Steffenel. Efficient prototyping of fault tolerant map-reduce applications with docker-hadoop. 2014.
- M. Cogorno, J. Rey and S. Nesmachnow. Fault tolerance in hadoop mapreduce implementation. May work package 1 intermediate report. 2013. http://hal.archives-ouvertes.fr/docs/00/86/31/76/PDF/D1.1-FaultToleranceHadoop.pdf, consultada Setiembre 2014.

También se escribió un paper que todavía no fue publicado pero ya fue enviado para su aprobación:

- J. Rey, M. Cogorno, S. Nesmachnow and L. A. Steffenel. Efficient Prototyping of Fault Tolerant Map-Reduce Applications with Docker-Hadoop. 10th Dependable and Adaptive Distributed Systems, 30th ACM Symposium on Applied Computing, Salamanca, España, 2015.

Bibliografía

[1] Parallels Cloud Server. http://sp.parallels.com/products/pcs/, consultada Setiembre 2014.

[2] The Message Passing Interface (MPI) standard. http://www.mcs.anl.gov/research/projects/mpi/, consultada Agosto 2014.

[3] Apache. The Hadoop distributed file system: architecture and design, 2008. http://hadoop.apache.org/docs/r0.18.3/hdfs_design.html, consultada Julio 2014.

[4] M. Dias De Assunção, A. Di Costanzo, and R. Buyya. Evaluating the cost-benefit of using cloud computing to extend the capacity of clusters. In *Proceedings of the 18th ACM international symposium on high performance distributed computing*, pages 141–150, Delft, Netherlands, 2009. ACM.

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, 2003.

[6] D. Beimborn, T. Miletzki, and S. Wenzel. Platform as a service (PaaS). *Business & Information Systems Engineering*, 3(6):381–384, 2011.

[7] A. N. Bessani, V. V. Cogo, M. Correia, P. Costa, M. Pasin, F. Silva, L. Arantes, O. Marin, P. Sens, and J. Sopena. Making Hadoop MapReduce byzantine fault-tolerant. *DSN, Fast abstract (2010)*. http://www.gsd.inesc-id.pt/ mpc/pubs/bft-mapreduce-fadsn10.pdf, consultada Setiembre 2014.

[8] S. Bhardwaj, L. Jain, and S. Jain. Cloud computing: A study of infrastructure as a service (IaaS). *International Journal of engineering and information Technology*, 2(1):60–63, 2010.

[9] D. Castro. How much will PRISM cost the U.S. cloud computing industry? 2013. http://www2.itif.org/2013-cloud-computing-costs.pdf, consultada Setiembre 2014.

[10] R. Chansler, H. Kuang, S. Radia, K. Shvachko, and S. Srinivas. The Hadoop distributed file system. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 6–7, Incline Village, NV, USA, 2010.

[11] M. Cogorno, J. Rey, and S. Nesmachnow. Fault tolerance in Hadoop MapReduce implementation. May Work Package 1 Intermediate Report. 2013. http://hal.archives-ouvertes.fr/docs/00/86/31/76/PDF/D1.1-FaultToleranceHadoop.pdf, consultada Setiembre 2014.

[12] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[13] B. des Ligneris. Virtualization of Linux based computers: the Linux-VServer project. In *19th International Symposium on High Performance Computing Systems and Applications*, pages 340–346, Paris, Francia, 2005.

[14] M. D. Dikaiakos, D. Katsaros, P. Mehra, G. Pallis, and A. Vakali. Cloud computing: distributed Internet computing for IT and scientific research. *Internet Computing, IEEE*, 13(5):10–13, 2009.

[15] J. Evans. Fault tolerance in Hadoop for work migration. Technical report, Technical Report CSCI, 2011. http://salsahpc.indiana.edu/b534projects/sites/default/files/public/0_Fault%20Tolerance%20in%20Hadoop%20for%20Work%20Migration_Evans,%20Jared%20Matthew.pdf, consultada Setiembre 2014.

[16] C. Finch. The benefits of the software-as-a-service model. *Computerworld Management*, 2, 2006. http://www.appware.com/best_practices/pdf/saas_02_article.pdf, consultada Setiembre 2014.

[17] The Apache Software Foundation. Apache Hadoop. http://hadoop.apache.org/, consultada Mayo 2014.

[18] The Apache Software Foundation. Apache ZooKeeper. http://zookeeper.apache.org/, consultada Mayo 2014.

[19] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke. Condor-G: A computation management agent for multi-institutional grids. *Cluster Computing*, 5(3):237–246, 2002.

[20] I. Habib. Virtualization with kvm. *Linux Journal*, 2008(166):8, 2008.

[21] C. He, D. Weitzel, D. Swanson, and Y. Lu. HOG: Distributed Hadoop MapReduce on the grid. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 1276–1283, Salt Lake City, UT, USA, 2012. IEEE.

[22] M. Helsley. LXC: Linux container tools, 2009. http://www.ibm.com/developerworks/linux/library/l-lxc-containers, consultada Julio 2014.

[23] T. Heo. Cgroup unified hierarchy, 2014. https://www.kernel.org/doc/Documentation/cgroups/unified-hierarchy.txt, consultada Julio 2014.

[24] P. Hunt, M. Konar, F. Paiva, and B. Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *USENIX Annual Technical Conference*, volume 8, page 9, Boston, USA, 2010.

[25] S. Hykes and Docker Inc. Docker. http://www.docker.com/, consultada Mayo 2014.

[26] I.Rubinstein and V J. Hoboken. Privacy and security in the cloud: Some realism about technical solutions to transnational surveillance in the post-Snowden era. 2014. http://papers.ssrn.com/sol3/papers.cfm?abstract_id=2443604, consultada Setiembre 2014.

[27] S. Kadirvel and J. A. B. Fortes. Towards self-caring MapReduce: a study of performance penalties under faults. *Concurrency and Computation: Practice and Experience*, 2013. http://onlinelibrary.wiley.com/doi/10.1002/cpe.3044/abstract, consultada Setiembre 2014.

[28] N. Kuromatsu, M. Okita, and K. Hagihara. Evolving fault-tolerance in Hadoop with robust auto-recovering JobTracker. *Bulletin of Networking, Computing, Systems, and Software*, 2(1):p–4, 2013.

[29] B. Lampson, M. Paul, and H. J. Siegert, editors. *Distributed systems - Architecture and implementation, an advanced course*, London, UK, 1981. Springer-Verlag.

[30] G. Le Lann. Distributed systems-towards a formal approach. In *IFIP Congress*, volume 7, pages 155–160, Toronto, Canada, 1977.

[31] V. Marmol and R. Jnagal. Let me contain that for you. http://es.slideshare.net/vmarmol/containers-google, consultada Julio 2014.

[32] F. Marozzo, D. Talia, and P. Trunfio. A peer-to-peer framework for supporting mapreduce applications in dynamic cloud environments. In *Cloud Computing*, pages 113–125. Springer, 2010.

[33] P. Mell and T. Grance. The NIST definition of cloud computing. 2011.

[34] P. Menage. Linux Cgroups. https://www.kernel.org/doc/Documentation /cgroups/cgroups.txt, consultada Julio 2014.

[35] A. K. Mok. Fundamental design problems of distributed systems for the hard-real-time environmen. Technical report, Cambridge, MA, USA, 1983.

[36] H. Muhammad. HTOP. http://hisham.hm/htop, consultada Julio 2014.

[37] OpenVZ. OpenVZ Linux Containers. http://openvz.org, consultada Julio 2014.

[38] V. Patil and P. Soni. Hadoop skeleton & fault tolerance in Hadoop clusters. *International Journal of Application or Innovation in Engineering & Management*, 2(2):247–250, 2013.

[39] C. M. Pham, V. Dogaru, R. Wagle, C. Venkatramani, Z Kalbarczyk, and R. Iyer. An evaluation of zookeeper for high availability in system S. In *Proceedings of the 5th ACM/SPEC international conference on Performance engineering*, pages 209–217. ACM, 2014.

[40] J. Rey, M. Cogorno, S. Nesmachnow, and L. A. Steffenel. Efficient prototyping of fault tolerant Map-Reduce applications with Docker-Hadoop. 2014.

[41] F. Salbaroli. Proposal for a fault tolerant Hadoop Job Tracker, 2008. https://889d8a5e-a-62cb3a1a-s-sites.googlegroups.com/site/hadoopthesis/Home/FaultTolerantHadoop.pdf, consultada Setiembre 2014.

[42] I. Sfiligoi. GlideinWMS—A generic pilot-based workload management system. In *Journal of Physics: Conference Series*, volume 119, page 062044. IOP Publishing, 2008.

[43] S. Soltesz, H. Pötzl, M. Fiuczynski, A. Bavier, and L. Peterson. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. *SIGOPS Oper. Syst. Rev.*, 41(3):275–287, March 2007.

[44] L. A. Steffenel. STIC-AmSud PER-MARE Project Training, 2013.

[45] L. A. Steffenel, O. Flauzac, A. Schwertner, P. Pitthan, B. Stein, G. Cassales, S. Nesmachnow, J. Rey, M. Cogorno, M. Kirsch-Pinheiro, and C. Souveyet. MapReduce challenges on pervasive grids. *Journal of Computer Science*, 10(11):2193–2209, 2014.

[46] L. A. Steffenel, O. Flauzac, A. Schwertner, P. Pitthan, B. Stein, S. Nesmachnow, M. Kirsch, and D. Diaz. PER-MARE: Adaptive deployment of MapReduce over pervasive grids. In *P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2013 Eighth International Conference on*, pages 17–24. IEEE, 2013.

[47] S. Subashini and V. Kavitha. A survey on security issues in service delivery models of cloud computing. *Journal of Network and Computer Applications*, 34(1):1–11, 2011.

[48] Bing Tang, Mircea Moca, Stephane Chevalier, Haiwu He, and Gilles Fedak. Towards mapreduce for desktop grid computing. In *P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2010 International Conference on*, pages 193–200. IEEE, 2010.

[49] V. Tuulos. Disco Project. http://discoproject.org/, consultada Mayo 2014.

[50] S. S. Vazhkudai, X. Ma, V. W. Freeh, J. W. Strickland, N. Tammineedi, and S. L. Scott. Freeloader: Scavenging desktop storage resources for scientific data. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 56, Seattle, WA, USA, 2005. IEEE Computer Society.

[51] T. Velte, A. Velte, and R. Elsenpeter. *Cloud computing, a practical approach*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 2010.

[52] B. Walters. VMware virtual platform. *Linux journal*, 1999(63es):6, 1999.

[53] T. White. *Hadoop: the definitive guide*. O'Reilly Media, Inc., 2009.

[54] M. Xavier, M. Neves, F. Rossi, T. Ferreto, T. Lange, and C. De Rose. Performance evaluation of container-based virtualization for high performance computing environments. In *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, pages 233–240, Belfast, North Ireland, 2013. IEEE.

[55] Yahoo. Yahoo's Hadoop tutorial: Module 2: The Hadoop distributed file system. https://developer.yahoo.com/hadoop/tutorial/module2.html, consultada en Mayo 2014.

[56] Yahoo. Yahoo's Hadoop tutorial: Module 4: MapReduce. https://developer.yahoo.com/hadoop/tutorial/module4.html, consultada en Mayo 2014.

[57] Q. Zheng. Improving MapReduce fault tolerance in the cloud. In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–6. IEEE, 2010.

[58] D. Zissis and D. Lekkas. Addressing cloud computing security issues. *Future Generation Computer Systems*, 28(3):583–592, 2012.