FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

TÍTULO: DOMEX: Ambiente distribuido Web para la ejecución de jobs MapReduce

AUTORES: Scoffield David, Giorgetti Valentín

DIRECTOR/A: Waldo Hasperué

CODIRECTOR/A:

ASESOR/A PROFESIONAL:

CARRERA: Licenciatura en Sistemas

Resumen

Esta tesina aborda el desarrollo de DOMEX, un sistema web para procesamiento distribuido de jobs MapReduce, que permite a los estudiantes experimentar y aprender sobre el paradigma de manera práctica y accesible, utilizando sus propios dispositivos como nodos de un clúster. Esta herramienta busca superar las barreras asociadas a los costos y la complejidad de las infraestructuras tradicionales o servicios en la nube, brindando una alternativa viable y eficiente para el aprendizaje y la implementación de soluciones MapReduce en entornos educativos.

Palabras Clave

Big Data, MapReduce, Hadoop, HDFS, WebSockets, WebRTC, Python, clúster, procesamiento distribuido, comunicación en tiempo real, algoritmos, computación en la nube, STUN, TURN, NAT, HTTPS

Conclusiones

El desarrollo de DOMEX ha sido un proceso enriquecedor que permitió integrar diversas tecnologías y aplicar los conocimientos adquiridos durante la carrera. Este proyecto, además de reforzar conocimientos adquiridos, buscó superar barreras de infraestructura para ofrecer una solución accesible.
Se espera que DOMEX facilite el aprendizaje del paradigma MapReduce en el ámbito académico, proporcionando una herramienta práctica y efectiva para estudiantes que deseen explorar este enfoque de procesamiento distribuido.

Trabajos Realizados

Desarrollo de DOMEX, una herramienta educativa para la ejecución distribuida de jobs MapReduce, que será utilizada en la cátedra "Conceptos y Aplicaciones en Big Data". El código fuente está disponible en un repositorio público en GitHub, y se han realizado pruebas para validar su correcto funcionamiento. Además, se ha documentado detalladamente todo el proceso, incluyendo los desafíos encontrados y las soluciones implementadas.

Trabajos Futuros

Las mejoras propuestas incluyen la implementación de las etapas shuffle y sort, la posibilidad de ejecutar soluciones iterativas y la asignación de diferentes funciones map para conjuntos de datos específicos. También se sugiere incorporar el manejo de variables globales, un sistema de autocompletado en el editor de código, y extender la operatividad del sistema por fuera de una red local. Además, se recomienda buscar formas para optimizar el uso de los recursos computacionales en los nodos y mejorar el algoritmo de distribución de claves para aumentar la eficiencia y escalabilidad del sistema.

Fecha de la presentación: **SEPTIEMBRE 2024**

ÍNDICE

Capítulo 1: Introducción 5
1.1 Motivación 5
1.2 Objetivos 7
1.3 Resultados esperados 7
1.4 Estructura de la tesina 8
Capítulo 2: Big Data y Paradigmas de Programación 10
2.1 Explicación de Big Data 10
2.2 Tecnologías que posibilitan trabajar con Big Data 12
2.2.1 Hadoop Ecosystem 12
2.2.2 Apache Spark 13
2.2.3 Apache Flume y Apache Kafka 13
2.2.4 NoSQL Databases 14
2.2.5 Data Warehousing and Analytics 14
2.2.6 Herramientas de Machine Learning 14
Capítulo 3: Ecosistema Hadoop y MapReduce 16
3.1 El Paradigma MapReduce 16
3.1.1 Historia y Evolución 16
3.2 Componentes del ecosistema Apache Hadoop 17
3.2.1 HDFS (Hadoop Distributed File System) 18
3.2.2 Job Tracker 19
3.2.3 Task Tracker 19
3.2.4 YARN (Yet Another Resource Negotiator) 19
3.2.5 Apache Hive 20
3.2.6 Apache Pig 20
3.2.7 Apache HBase 20
3.3 Etapas del Paradigma MapReduce 21
3.3.1 Map 21
3.3.2 Shuffle 22
3.3.3 Sort 22
3.3.4 Reduce 22
3.3.5 Combine 23
3.4 Ejemplo de Conteo de Palabras con MapReduce 23
3.5 Conclusión 26
Capítulo 4: Sistemas Distribuidos y Comunicación en Tiempo Real 27
4.1 Explicación de Sistemas Distribuidos 27
4.1.1 Clasificación 28
4.2 Mecanismos de Comunicación en Tiempo Real para la Web 29
4.2.1 WebSockets 29
4.2.2 WebRTC 30
4.2.3 Integración de WebSockets y WebRTC en la Aplicación Propuesta 30
Capítulo 5: Investigación previa y Soluciones Actuales 32
5.1 Soluciones Actuales para la Ejecución de Jobs MapReduce 32

5.1.1 Herramienta Utilizada en el Dictado de la Asignatura 33
5.1.2 Otras Soluciones Existentes 34
5.1.2.1 Implementación de Hadoop con un Clúster de Computadoras 34
5.1.2.2 Soluciones Basadas en la Nube 35
5.1.2.3 Soluciones Híbridas y Alternativas 37
5.2 Ejecutar Python en el Navegador 38
5.2.1 Pyodide 39
5.2.2 Brython 39
5.2.3 Skulpt 40
5.2.4 Transcrypt 40
5.3 Análisis y Aplicación de Código Base para Funciones Map, Combine y Reduce 41
5.3.1 Necesidad de un Código Base 41
5.3.2 Implementación del Código Base 42
5.3.3 Integración con React 45
Capítulo 6: Implementación de la Aplicación 46
6.1 Arquitectura de la Aplicación 46
6.1.1 Frontend (Next.js) 47
6.1.2 Backend (Node.js y Express.js) 48
6.1.3 Servidor NGINX 48
6.1.4 Cert-Generator 49
6.1.5 Docker y Redes 49
6.2 Flujo de Comunicación de la Aplicación 50
6.2.1 Frontend y Backend 50
6.2.2 Comunicación entre peers (frontend) 53
6.3 Explicación del Funcionamiento de la Aplicación 55
6.3.1 Dos Tipos de Nodos: Master y Slaves 56
6.3.2 Funciones del Master 57
6.3.2.1 Definición de funciones 57
6.3.2.2 Monitoreo de Estado y Gestión de Conexiones 58
6.3.2.3 Asignación y Distribución de Tareas 59
6.3.2.4 Manejo de Resultados y Finalización de Tareas 61
6.3.3 Funciones del Slave 61
6.3.3.1 Ejecución de Tareas Asignadas 62
6.3.3.2 Monitoreo y Reporte de Estado 62
6.3.3.3 Manejo de Errores 62
6.3.3.4 Visualización de estadísticas 62
6.3.4 Subida de Archivos 62
6.3.5 Circuito de Mensajes 64
6.3.5.1 Inicio de Conexión y Señalización 64
6.3.5.2 Distribución de Tareas y Sincronización 64
6.3.5.3 Ejecución y Reporte de Resultados 65
6.3.5.4 Finalización de Tareas y Desconexión 65
6.3.5.5 Sincronización para la Ejecución del Job MapReduce 65
6.3.6 Gestión de Sesiones 67

6.3.6.1 Gestión de Sesiones en el Backend 67
6.3.6.2 Gestión de Sesiones en el Frontend 68
6.3.6.3 Coordinación Integral 68
6.3.6.4 Persistencia y Reconexión 68
6.3.6.5 Desconexiones Involuntarias y Manejo de Estados 69
6.3.7 Estadísticas 69
6.3.7.1 Tiempo de ejecución 69
6.3.7.2 Uso de recursos 70
6.3.7.3 Resultados de ejecución 70
6.4 Limitaciones actuales de la App 71
6.4.1 Restricción en el Tamaño de Archivos 71
6.4.2 Ausencia de Restricciones en la Cantidad de Archivos 71
6.4.3 Limitaciones de Conectividad y Redes 72
6.4.4 Consideraciones sobre la Escalabilidad 72
6.4.5 Conclusiones y Recomendaciones 73
Capítulo 7: Pruebas funcionales de la aplicación 74
7.1 Problema de Proyección 75
7.2 Problema de Contador 78
7.3 Problema de Agregación 80
7.4 Problema de Join 82
7.5 Problema de Clustering (k-means) 85
7.6 Conclusiones 87
Capítulo 8: Problemas y Soluciones en el Desarrollo 88
8.1 NAT Simétrico y Asimétrico 88
8.2 Servidores STUN y TURN 90
8.3 Implementación de HTTPS en Red Local 92
8.4 Particionado de Mensajes por la Red WebRTC en Chunks 94
8.4.1 Implementación de la Transmisión de Chunks 94
8.4.2 Desafíos y Soluciones 95
Capítulo 9: Conclusiones y Trabajo Futuro 96
9.1 Resumen de los Hallazgos 96
9.2 Posibles Mejoras y Futuras Investigaciones 98
9.2.1 Implementación de las Etapas Shuffle y Sort 98
9.2.2 Soluciones Iterativas 98
9.2.3 Configuración de Funciones Map para Diferentes Conjuntos de Datos 99
9.2.4 Administración de Parámetros y Variables Globales 99
9.2.5 Integración de IntelliSense en el Editor de Código 99
9.2.6 Implementación de la Aplicación para Funcionamiento Global 100
9.2.7 Mejora del Poder Computacional de los Nodos 100
9.2.8 Optimizar el algoritmo de distribución de claves 100
9.2.9 Conclusión 101
Referencias bibliográficas 102
Anexos 105
Anexo 1: Instalación y ejecución local de la aplicación 105

Link del repositorio: 105
1. Pre-requisitos 105
2. Guía de Instalación 106
3. Configuración 107
PRIVATE_IP 108
FRONT_INTERNAL_PORT y BACK_INTERNAL_PORT 108
NEXT_PUBLIC_ICESERVER 108
NEXT_PUBLIC_SERVER_URL 109
CLUSTER_IDS_LENGTH 109
Anexo 2: Dockerización de la aplicación 110
1. Configuración General de Docker Compose 110
2. Descripción de Servicios 111
a. Servicio “cert-generator” 111
b. Servicio “next-app” 113
c. Servicio “backend” 117
d. Servicio “nginx” 120
Anexo 3: Configuración del Servidor NGINX 123
1. Bloque 'events { }' 123
2. Bloque 'http { }' 123
3. Bloque 'server { }' para Redirección de HTTP a HTTPS 123
4. Bloque 'server { }' para Manejo de Solicitudes HTTPS 124
Anexo 4: Código del módulo de Python MapReduceJob 126

Capítulo 1: Introducción

1.1 Motivación

En la actualidad, la enorme cantidad de datos que se genera a una velocidad vertiginosa y en diversos formatos desafía las capacidades de las herramientas tradicionales de procesamiento de información. Este fenómeno, denominado Big Data, ha revolucionado la forma en que las organizaciones toman decisiones, investigan e innovan (White, 2015).

El crecimiento desmedido de datos provenientes de múltiples fuentes plantea grandes retos en cuanto a la manipulación y análisis eficiente de la información. Este escenario demanda el desarrollo de soluciones tecnológicas avanzadas que no solo manejen grandes volúmenes de datos, sino que también aprovechen al máximo su potencial informativo.

Vivimos en la era de los datos. Según una estimación de IDC, el tamaño del "universo digital" en 2013 era de 4.4 zettabytes, con una proyección de crecimiento a 44 zettabytes para 2020 (White, 2015). Este aumento masivo en la cantidad de datos proviene de diversas fuentes, como la Bolsa de Nueva York que genera alrededor de 4-5 terabytes de datos por día, y Facebook que aloja más de 240 mil millones de fotos, creciendo a razón de 7 petabytes por mes (White, 2015). Además, como nos plantea Petroc (2023) en un artículo de Statista[1], la cantidad de datos está creciendo a un ritmo increíble y se prevé que alcance los 180 zettabytes en 2025.

En el transcurso de nuestros estudios, cursamos la asignatura "Conceptos y Aplicaciones de Big Data", donde exploramos estos temas y generamos soluciones empleando el paradigma MapReduce: "*un paradigma de procesamiento de datos caracterizado por dividirse en dos fases o pasos diferenciados: Map y Reduce. Estos subprocesos asociados a la tarea se ejecutan de manera distribuida, en diferentes nodos de procesamiento o esclavos. Para controlar y gestionar su ejecución, existe un proceso Master o Job Tracker. También es el encargado de aceptar los nuevos trabajos enviados al sistema por los clientes*" (Fernandez, 2022).

Para implementar este paradigma, utilizamos una herramienta proporcionada por la cátedra, conocida como MRE. Este módulo, implementado en Python, permite la declaración y emulación de la ejecución de jobs MapReduce de manera secuencial.

---

[1] Artículo de la web de Statista que enseña el crecimiento de los datos/información en el tiempo y una previsión a futuro

https://www.statista.com/statistics/871513/worldwide-data-created/?trk=article-ssr-frontend-pulse_little-text-block

Sin embargo, una de las limitaciones de esta herramienta es la necesidad de que los estudiantes instalen Python en sus sistemas. Además, la ejecución se realiza de manera secuencial en la computadora del estudiante, en lugar de distribuirse como lo hace el framework MapReduce.

Otra opción para los estudiantes es configurar un clúster[2] para ejecutar los jobs MapReduce, lo cual no es una tarea sencilla. Para la conformación de un clúster se requieren varias computadoras dedicadas, y la instalación del framework MapReduce es compleja y técnica, superando los temas en los que se tiene el enfoque en la cátedra. Simular un entorno distribuido propio no es viable debido a las configuraciones necesarias y las limitaciones de infraestructura (IBM, 2022). La tercera opción es la de alquilar servicios en la nube, lo cual resulta costoso y poco práctico para los estudiantes.

Para entender más a fondo lo que se quiere llevar a cabo es necesario entender lo que conlleva implementar un sistema distribuido: *“Los sistemas distribuidos pueden concebirse como aquellos cuya funcionalidad se encuentra fraccionada en componentes que al trabajar sincronizada y coordinadamente otorgan la visión de un sistema único, siendo la distribución, transparente para quien hace uso del sistema”* (Patricia Bazán, Doctora en Ciencias Informáticas graduada en la Facultad de Informática de la UNLP, 2017).

En respuesta a estas limitaciones, nuestra propuesta consiste en la creación de una herramienta para la cátedra "Conceptos y Aplicaciones de Big Data": una aplicación web que facilite a los estudiantes la creación de clústers, la definición de Jobs MapReduce (incluyendo las funciones de mapeo, combinación y reducción de datos), y que permita una ejecución distribuida a través de la red utilizando las computadoras de los estudiantes como nodos esclavos del clúster. Este enfoque busca proporcionar a la cátedra y a sus estudiantes una herramienta que brinde una experiencia más cercana a la operativa con Big Data en entornos distribuidos, superando las limitaciones previamente mencionadas.

Es crucial aclarar que esta herramienta no busca reemplazar el framework MapReduce ni trabajar con grandes volúmenes de datos. En lugar de eso, pretende ofrecer a los estudiantes una forma amigable de analizar el paradigma, estudiar la distribución de tareas, la cantidad de información que maneja cada nodo y los datos que se transmiten en el clúster.

[2] Grupos de servidores que se administran de manera coordinada y que participan en la gestión de carga de trabajo. Un clúster puede contener nodos o servidores de aplicaciones individuales.

1.2 Objetivos

Este trabajo de grado tiene como objetivo la propuesta e implementación de DOMEX (Distributed Online MapReduce EXperience), un sistema web que permita establecer un entorno distribuido para la ejecución de jobs MapReduce. Este entorno será administrado por la aplicación, utilizando los diferentes dispositivos conectados como nodos de un clúster, distribuyendo la carga de procesamiento entre ellos. La ejecución de los jobs se realizará íntegramente en los navegadores de los usuarios. Además, permitirá subir archivos (de tamaño limitado) que serán procesados, emulando un sistema de archivos distribuido similar a HDFS (Hadoop Distributed File System).

Para alcanzar este objetivo, se llevarán a cabo las siguientes actividades:

1. Evaluar la posibilidad de ejecutar código Python en el navegador.
2. Investigar en profundidad mecanismos de comunicación en tiempo real para la web tales como WebSockets (entre cliente y servidor) y WebRTC (entre clientes) provistos por la plataforma web.
3. Diseñar y desarrollar un sistema web que permita a los diferentes usuarios que ingresen, crear clústers o unirse a existentes, subir archivos sobre los que se trabajarán y ejecutar jobs MapReduce.
4. Proporcionar métricas sobre la ejecución de los jobs MapReduce, incluyendo tiempo total de procesamiento y cantidad de datos transmitidos por cada nodo.
5. Realizar pruebas para verificar el funcionamiento y alcance de la aplicación.

1.3 Resultados esperados

1. **DOMEX**: Una nueva herramienta que será utilizada por la cátedra “Conceptos y Aplicaciones de Big Data” para la ejecución de jobs MapReduce de manera distribuida a través de nodos conectados a una misma red.
2. **Código fuente**: Disponible en un repositorio público en Github para su libre acceso.
3. **Visualización de resultados**: Los resultados del job MapReduce estarán disponibles para su análisis posterior.
4. **Pruebas de funcionamiento**: Validación del correcto funcionamiento de la aplicación mediante diferentes casos de prueba.

5. **Documentación**: Un documento detallando los temas abordados, desafíos encontrados y soluciones implementadas.

1.4 Estructura de la tesina

La estructura de esta tesina se organiza de manera que cada capítulo se desarrolle de forma coherente y fluida, abarcando tanto los aspectos teóricos como prácticos del proyecto.

Para comenzar, el **Capítulo 1: Introducción** presenta la motivación del trabajo, los objetivos que se pretenden alcanzar, los resultados esperados y la estructura del documento. Este capítulo establece el contexto y la importancia del tema de estudio, destacando los desafíos y las soluciones propuestas, lo cual permite entender la relevancia del proyecto desde sus primeras líneas.

A continuación, en el **Capítulo 2: Big Data y Paradigmas de Programación**, se proporciona una visión general del concepto de Big Data y de los diferentes paradigmas de programación utilizados para su procesamiento. Este capítulo incluye una revisión de la literatura sobre Big Data y una comparación de los distintos enfoques de programación, ofreciendo una base teórica sólida para entender el contexto en el que se desarrolla la investigación. La discusión aquí es fundamental para preparar al lector sobre los fundamentos teóricos necesarios para comprender los siguientes capítulos.

Luego, el **Capítulo 3: Ecosistema Hadoop y MapReduce** describe el paradigma MapReduce y los componentes del ecosistema Hadoop que lo soportan. Se exploran los conceptos básicos, la historia y evolución del paradigma, así como los componentes específicos de Hadoop. Esta explicación proporciona un marco de referencia claro y detallado para la implementación técnica del proyecto.

Posteriormente, el **Capítulo 4: Sistemas Distribuidos y Comunicación en Tiempo Real** explica los conceptos fundamentales de los sistemas distribuidos y los mecanismos de comunicación en tiempo real utilizados en la web. En este capítulo se abordan tecnologías como WebSockets y WebRTC, y su aplicación en sistemas distribuidos. La discusión de estas tecnologías es crucial para entender cómo se logrará la comunicación efectiva entre los nodos del clúster en la aplicación propuesta.

Más adelante, en el **Capítulo 5: Investigación Previa y Soluciones Actuales**, se analizan las herramientas y métodos existentes para la ejecución de Python en el navegador y otras soluciones actuales para la ejecución de jobs MapReduce. Este capítulo incluye un análisis de las soluciones actuales y una evaluación de su aplicabilidad, identificando las mejores prácticas y las limitaciones de las soluciones

existentes. Este análisis comparativo es esencial para justificar la elección de la solución propuesta en esta tesina.

En el **Capítulo 6: Implementación de la Aplicación**, se detalla la arquitectura de la aplicación, el flujo de comunicación y el funcionamiento de los diferentes componentes. Aquí se describe la implementación técnica y los desafíos encontrados durante el desarrollo. Este capítulo proporciona una guía detallada del proceso de creación de la aplicación DOMEX, lo cual es fundamental para los lectores interesados en los aspectos técnicos del proyecto.

Luego, el **Capítulo 7: Pruebas Funcionales de la Aplicación** describe las pruebas realizadas para validar el correcto funcionamiento del sistema. En este capítulo se evalúa el rendimiento de la aplicación y su capacidad para manejar diferentes escenarios de uso, asegurando que cumpla con los requisitos establecidos. Estas pruebas son cruciales para demostrar la viabilidad y eficacia del sistema desarrollado.

En el **Capítulo 8: Problemas y Soluciones en el Desarrollo**, se discuten las complicaciones encontradas durante el desarrollo de la aplicación y las soluciones implementadas. Este capítulo aborda problemas como la configuración de servidores STUN y TURN, el problema de los NAT simétrico y asimétrico, y la implementación de HTTPS en una red local, entre otros. La discusión de estos problemas y soluciones proporciona valiosas lecciones aprendidas para futuros desarrollos en este campo.

Finalmente, el **Capítulo 9: Conclusiones y Trabajo Futuro** resume los hallazgos del proyecto y propone posibles mejoras y futuras líneas de investigación. Se discuten las implicaciones de los resultados obtenidos y se sugieren áreas para futuras investigaciones. Este capítulo destaca el impacto y las contribuciones del trabajo realizado, y ofrece una visión de cómo podría evolucionar este proyecto en el futuro.

Capítulo 2: Big Data y Paradigmas de Programación

2.1 Explicación de Big Data

El concepto de Big Data ha emergido como un componente crucial en la era de la información, representando conjuntos de datos tan extensos y complejos que las herramientas y técnicas tradicionales de procesamiento de datos no pueden manejarlos de manera eficiente. Este fenómeno no solo ha transformado la capacidad de almacenamiento y procesamiento de datos, sino que también ha revolucionado la manera en que las organizaciones abordan la toma de decisiones, la investigación y la innovación.

Históricamente, la gestión de datos se centraba en bases de datos relacionales y sistemas de almacenamiento estructurado. En las décadas de 1970 y 1980, el modelo relacional propuesto por E.F. Codd y el lenguaje SQL se convirtieron en estándares de la industria para la manipulación de datos estructurados. Sin embargo, con el avance de la tecnología y la digitalización de la sociedad, el volumen, la velocidad y la variedad de los datos comenzaron a superar las capacidades de estos sistemas tradicionales. Según White (2015), la cantidad de datos almacenados electrónicamente en 2013 era de aproximadamente 4.4 zettabytes, con una proyección de crecimiento a 44 zettabytes para 2020.

En la década de 2000, el término "Big Data" empezó a ganar tracción, impulsado por la creciente cantidad de datos generados por Internet, redes sociales, dispositivos móviles y sensores. La necesidad de nuevas soluciones se hizo evidente. Las tres V de Big Data: volumen, velocidad y variedad, se convirtieron en los pilares fundamentales para entender este fenómeno. Según Laney (2001), quien fue uno de los primeros en conceptualizar este marco, estas tres V proporcionan una base para comprender los desafíos y oportunidades presentados por Big Data.

Volumen hace referencia a la enorme cantidad de datos que se generan y almacenan continuamente. Hoy en día, las organizaciones deben gestionar petabytes[3] e incluso exabytes[4] de datos, que provienen de diversas fuentes como

[3] Es una de las unidades de medida de datos digitales más grandes y equivale aproximadamente a 1024 terabytes (TB).
[4] Al igual que el petabyte, el exabytes es una unidad de medida de datos digitales, siendo 1 exabytes (EB) = 1024 petabytes (PT)

redes sociales, dispositivos IoT[5], transacciones financieras y más. Según Petroc (2023), se espera que la cantidad de datos alcance los 180 zettabytes en 2025.

Velocidad se refiere a la rapidez con la que se generan, capturan y procesan estos datos. En muchas aplicaciones modernas, la capacidad de procesar datos en tiempo real es crucial para mantener la competitividad y tomar decisiones informadas de manera oportuna. Como señala White (2015), proyectos como MyLifeBits de Microsoft Research han demostrado la viabilidad de capturar y almacenar interacciones personales en tiempo real, lo que subraya la importancia de la velocidad en el procesamiento de datos.

Variedad denota la diversidad de tipos de datos que se manejan, incluyendo datos estructurados, semiestructurados y no estructurados. Esta diversidad requiere herramientas flexibles capaces de integrar y analizar diferentes formas de datos. White (2015) menciona que plataformas como Facebook, que alberga más de 240 mil millones de fotos y crece a un ritmo de 7 petabytes por mes, ejemplifican la variedad de datos que se manejan en la actualidad.

Además de las tres V clásicas de Big Data, otros dos aspectos cruciales se han vuelto fundamentales en la gestión de datos: **veracidad y valor**. Estos conceptos no se limitan a escenarios Big Data, sino que son esenciales para cualquier análisis de datos. La veracidad se enfoca en la calidad y confiabilidad de los datos, asegurando que la información sea precisa y utilizable para análisis válidos. El valor se refiere a la capacidad de extraer insights significativos y aplicables de estos vastos conjuntos de datos, transformando datos crudos en información útil para la toma de decisiones estratégicas.

El impacto de Big Data se extiende a múltiples sectores. En la salud, por ejemplo, permite el análisis de grandes cantidades de datos de pacientes para mejorar el diagnóstico y tratamiento de enfermedades (Ristevski & Chen, 2018). En el sector financiero, facilita la detección de fraudes y la gestión de riesgos (Mashruwala, 2024). En el ámbito del entretenimiento, ayuda a las empresas a personalizar el contenido para sus usuarios, mejorando la experiencia del cliente y optimizando las estrategias de marketing (Ahmed, 2024).

En términos de almacenamiento y procesamiento, el desarrollo del ecosistema Hadoop en 2006 por Doug Cutting y Mike Cafarella marcó un hito importante. Basado en el modelo de programación MapReduce desarrollado por Google, Hadoop permite el procesamiento distribuido de grandes conjuntos de datos a través de clústers de computadoras, haciendo factible el manejo de Big Data. Como explica

---

[5] El término IoT, o Internet de las cosas, se refiere a la red colectiva de dispositivos conectados y a la tecnología que facilita la comunicación entre los dispositivos y la nube, así como entre los propios dispositivos.
https://aws.amazon.com/es/what-is/iot/#:~:text=El%20t%C3%A9rmino%20IoT%2C%20o%20Internet,como%20entre%20los%20propios%20dispositivos.

Fernández (2022), *"el paradigma MapReduce es un enfoque de procesamiento de datos distribuido que permite dividir una tarea en subprocesos que se ejecutan en diferentes nodos de procesamiento, coordinados por un proceso Master"*.

La evolución de las tecnologías de Big Data también incluye la aparición de sistemas NoSQL como MongoDB, Cassandra y HBase, que ofrecen soluciones más flexibles para el almacenamiento y consulta de datos no estructurados. Además, el avance en tecnologías de procesamiento en tiempo real, como Apache Kafka y Apache Storm, ha mejorado significativamente la capacidad de las organizaciones para manejar flujos de datos continuos y reaccionar rápidamente a la información en tiempo real.

Big Data no solo implica grandes volúmenes de datos, sino también la capacidad de extraer valor significativo de estos datos a través de técnicas avanzadas de análisis. El desarrollo de tecnologías y paradigmas de programación adecuados es esencial para manejar y analizar eficientemente estos datos, permitiendo a las organizaciones convertir desafíos en oportunidades y mantenerse a la vanguardia en sus respectivos campos.

Con el crecimiento continuo de la cantidad de datos, como se prevé que alcanzará los 180 zettabytes en 2025 según Petroc (2023), la importancia de Big Data y sus aplicaciones seguirá aumentando, redefiniendo continuamente las capacidades y estrategias de las organizaciones en la era digital.

2.2 Tecnologías que posibilitan trabajar con Big Data

El concepto de Big Data no se refiere a una única tecnología, sino a la integración de diversas herramientas y métodos que facilitan el procesamiento y análisis de los enormes volúmenes de datos que existen en la actualidad. Para ejecutar aplicaciones de Big Data de manera eficiente, es fundamental contar tanto con hardware especializado como con software adecuado que permitan manejar, almacenar y analizar datos de manera efectiva. Según White (2015), "la combinación de varias tecnologías hace más fácil el tratamiento de los datos con los que contamos hoy en día".

2.2.1 Hadoop Ecosystem

El ecosistema Hadoop es una de las plataformas más conocidas y utilizadas para el procesamiento de Big Data. Está compuesto por varios componentes esenciales:

- **Hadoop Distributed File System (HDFS):** Es un sistema de archivos distribuido diseñado para almacenar grandes volúmenes de datos a través de múltiples nodos en un clúster, proporcionando alta disponibilidad y tolerancia a fallos. Según White (2015), *"HDFS divide los datos en bloques grandes y*

los distribuye a través de varios nodos del clúster, asegurando que los datos sean altamente disponibles y resistentes a fallos".

- **MapReduce:** Un modelo de programación y un motor de procesamiento que permite la ejecución de tareas en paralelo a través de un clúster, dividiendo los trabajos en pasos de mapeo y reducción (Dean & Ghemawat, 2004). Como lo describen los autores, "*MapReduce es un modelo de programación y una implementación asociada para procesar y generar grandes conjuntos de datos*".

- **YARN (Yet Another Resource Negotiator):** Un sistema para la gestión de recursos en el clúster, que permite la ejecución simultánea de diferentes aplicaciones de procesamiento de datos (Vavilapalli et al., 2013).

2.2.2 Apache Spark

Apache Spark es una herramienta de procesamiento de datos en memoria que proporciona una alternativa más rápida y flexible al MapReduce de Hadoop. Spark permite la ejecución de tareas en tiempo real y soporta un amplio rango de aplicaciones, desde procesamiento en lotes hasta streaming, aprendizaje automático y análisis de gráficos. La capacidad de Spark para realizar operaciones iterativas en memoria lo convierte en una opción ideal para tareas que requieren repetidos accesos a los datos, tal como lo destacan, Zaharia et al. (2012), *"Spark es el primer sistema que permite utilizar un lenguaje de programación de propósito general a velocidades interactivas para la extracción de datos en memoria en clústeres"*.

2.2.3 Apache Flume y Apache Kafka

Para la ingesta y transmisión de datos en tiempo real, Apache Flume y Apache Kafka son dos herramientas clave:

- **Apache Flume**: Un servicio distribuido para la recolección y transferencia eficiente de grandes cantidades de datos de logs a un sistema centralizado. Flume es particularmente útil para la ingesta de datos en flujos continuos y su transferencia a HDFS u otros sistemas de almacenamiento.

- **Apache Kafka**: Una plataforma de streaming distribuida que permite publicar y suscribir flujos de registros, almacenar registros de manera tolerante a fallos y procesar flujos de registros en tiempo real. Narkhede et al. (2017) afirman que *"Kafka es capaz de manejar millones de mensajes por segundo, siendo ideal para aplicaciones que requieren baja latencia y alta escalabilidad"*.

2.2.4 NoSQL Databases

Las bases de datos NoSQL han emergido como una solución crucial para el almacenamiento y recuperación de grandes volúmenes de datos no estructurados o semiestructurados.

Sadalage y Fowler (2012) señalan que *"las bases de datos NoSQL son ideales para escenarios donde se requiere flexibilidad y escalabilidad"*.

Algunas de las principales categorías incluyen:

- **Document Databases**: Como MongoDB, que almacenan datos en formatos tipo documento, como JSON o BSON.
- **Column-Family Stores**: Como Apache Cassandra, que almacenan datos en tablas donde cada fila puede tener un conjunto diferente de columnas.
- **Key-Value Stores**: Como Redis, que utilizan una estructura simple de pares clave-valor.
- **Graph Databases**: Como Neo4j, que están diseñadas para almacenar y consultar grafos, lo cual es útil para aplicaciones que involucran relaciones complejas entre datos.

2.2.5 Data Warehousing and Analytics

Además de las bases de datos NoSQL, existen soluciones de almacenamiento de datos y análisis que son fundamentales para el procesamiento de Big Data:

- **Amazon Redshift**: Un data warehouse en la nube que permite el análisis de petabytes de datos.
- **Google BigQuery**: Un almacén de datos totalmente gestionado que facilita el análisis de grandes volúmenes de datos con rapidez y flexibilidad.
- **Apache Hive**: Un data warehouse basado en Hadoop que proporciona un lenguaje de consulta similar a SQL para el análisis de datos grandes.

2.2.6 Herramientas de Machine Learning

Para el análisis avanzado de Big Data, las herramientas de aprendizaje automático (machine learning) son esenciales:

- **Apache Mahout**: Un proyecto de Apache que se centra en el desarrollo de algoritmos escalables de machine learning.

- **TensorFlow**: Una biblioteca de código abierto para el aprendizaje automático desarrollada por Google, que permite construir y entrenar modelos de aprendizaje profundo.

Para culminar, es importante señalar que esta tesina se basa en la simulación de un entorno de ejecución para jobs MapReduce. En el siguiente capítulo, se profundizará en el Ecosistema Hadoop y el paradigma MapReduce. Se explorará su historia, evolución y los componentes específicos que conforman Hadoop, proporcionando un marco de referencia esencial para comprender la implementación técnica del proyecto y cómo MapReduce facilita el procesamiento distribuido de grandes volúmenes de datos. Esta explicación será crucial para entender cómo nuestra herramienta emula un entorno de ejecución distribuido, permitiendo a los estudiantes experimentar con el paradigma MapReduce de manera práctica y accesible.

Capítulo 3: Ecosistema Hadoop y MapReduce

3.1 El Paradigma MapReduce

El paradigma MapReduce es un modelo de programación distribuida diseñado para procesar y generar grandes conjuntos de datos con un algoritmo paralelo. Este enfoque fue desarrollado por Google para manejar la creciente cantidad de datos que necesitaban ser procesados eficientemente. Según Dean y Ghemawat (2004), MapReduce simplifica el procesamiento de datos a gran escala al dividir las tareas en dos etapas fundamentales: Map y Reduce.

La función **Map**, que es aplicada a cada registro de entrada, produce un conjunto de pares intermedios clave-valor. Estos pares son luego agrupados por clave y la función **Reduce** es aplicada a cada grupo para sintetizar una salida final. Este modelo no solo facilita el procesamiento paralelo de grandes volúmenes de datos, sino que también proporciona un mecanismo para manejar fallos y balancear la carga de trabajo en un clúster distribuido.

Un ejemplo típico de la función Map es el problema de "contar palabras" (Word Count), donde cada palabra de un texto es emitida como una clave con un valor asociado de uno. La función Reduce luego suma los valores para cada clave, proporcionando un conteo total de las apariciones de cada palabra en el texto. Este ejemplo simple ilustra cómo MapReduce puede ser utilizado para tareas de agregación y resumen de datos en gran escala.

3.1.1 Historia y Evolución

La historia del paradigma MapReduce se remonta a principios de los años 2000, cuando Google enfrentaba el reto de procesar grandes cantidades de datos de manera eficiente. Jeffrey Dean y Sanjay Ghemawat publicaron un artículo en 2004 titulado "MapReduce: Simplified Data Processing on Large Clusters", donde describían el modelo MapReduce y su implementación en Google. Este modelo se diseñó para facilitar el procesamiento distribuido y paralelo de grandes conjuntos de datos mediante dos fases fundamentales: el mapeo (Map) y la reducción (Reduce). En su trabajo, Dean y Ghemawat (2004) explicaban cómo el paradigma permitía procesar petabytes de datos de manera eficiente y escalable, utilizando hardware común.

El modelo MapReduce fue adoptado rápidamente debido a su simplicidad y eficacia. Las organizaciones empezaron a buscar maneras de implementar esta técnica en sus infraestructuras para mejorar el procesamiento de datos. En 2006, Doug Cutting y Mike Cafarella desarrollaron Apache Hadoop, una implementación de código abierto del modelo MapReduce. Hadoop permitió a empresas y desarrolladores

acceder a una herramienta poderosa para el procesamiento de datos a gran escala sin los costos asociados a las soluciones propietarias. La elección de un proyecto de código abierto permitió una rápida adopción y mejora continua por parte de la comunidad.

Desde su introducción, Hadoop ha evolucionado considerablemente. Inicialmente, Hadoop consistía principalmente en el sistema de archivos distribuido HDFS (Hadoop Distributed File System) y el motor de procesamiento MapReduce. Sin embargo, con el tiempo, se añadieron nuevas funcionalidades y mejoras que expandieron su capacidad y versatilidad. En 2013, se introdujo YARN (Yet Another Resource Negotiator) como parte de Hadoop 2.0, que permitió una gestión más eficiente de los recursos del clúster y soportó múltiples modelos de programación, no solo MapReduce.

El avance de Hadoop y MapReduce ha sido impulsado por la necesidad de manejar datos en cantidades cada vez mayores y en formatos variados. La evolución de estos sistemas ha permitido a las organizaciones de todo el mundo implementar soluciones de Big Data efectivas y escalables. Hoy en día, Hadoop y MapReduce son componentes esenciales de muchas infraestructuras de datos, utilizados por empresas para tareas que van desde el análisis de datos hasta el aprendizaje automático.

El impacto de MapReduce y Hadoop en el campo del Big Data no puede subestimarse. Estas tecnologías han transformado la manera en que las organizaciones almacenan, procesan y analizan datos. La capacidad de procesar grandes volúmenes de datos de manera eficiente ha permitido el desarrollo de nuevas aplicaciones y servicios que antes no eran posibles.

En resumen, la historia y evolución de MapReduce y Hadoop demuestran cómo las innovaciones tecnológicas pueden abordar desafíos complejos en el procesamiento de datos y cómo la adopción de modelos de código abierto puede acelerar el desarrollo y la implementación de soluciones efectivas a nivel global.

3.2 Componentes del ecosistema Apache Hadoop

Como vimos, Big Data no es una única tecnología, sino una combinación de varias tecnologías que facilitan el tratamiento de los datos que se generan hoy en día. Para ejecutar aplicaciones de Big Data, es esencial disponer de hardware y software específicos (White, 2015). En este contexto, Hadoop ha emergido como una de las plataformas más robustas y ampliamente adoptadas para manejar Big Data.

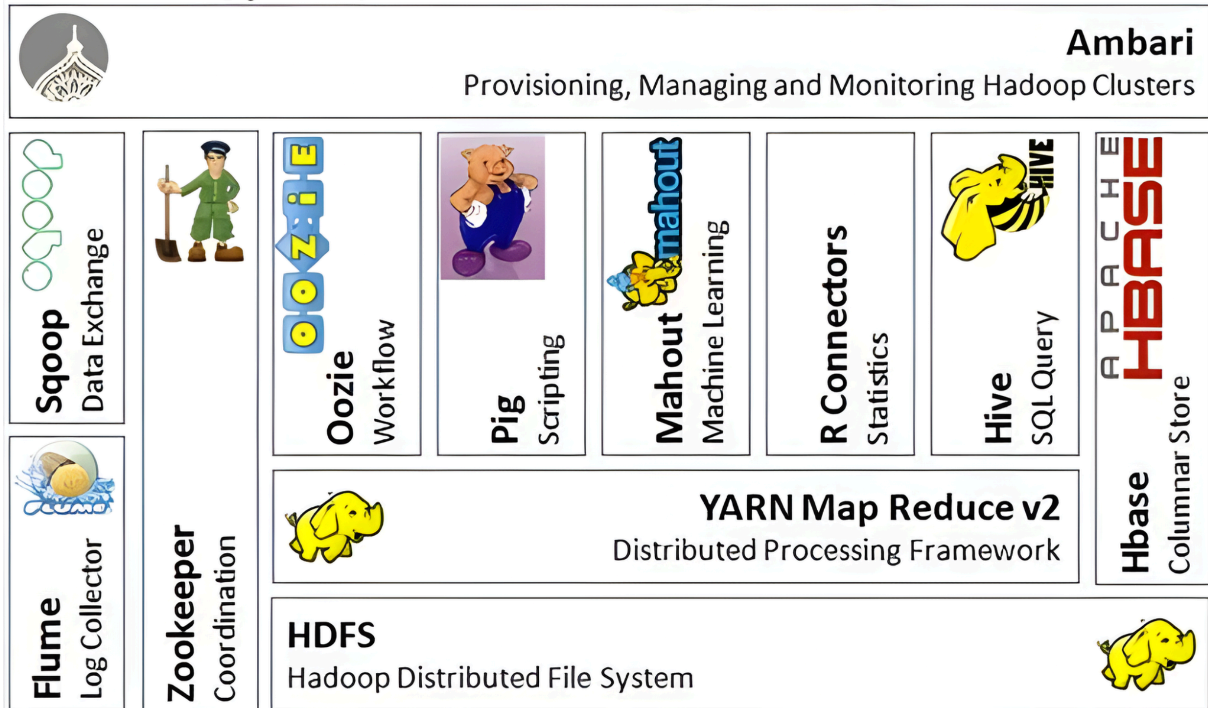


Fig. 3.2.1: Ecosistema Hadoop.

Fuente: (Bappalige, 2014)

Tal y como observamos en la Fig. 3.2.1, el ecosistema de Apache Hadoop, no es más que un conjunto de tecnologías que trabajan de manera coordinada para permitir el procesamiento de grandes volúmenes de datos. A continuación se dará un breve resumen de los principales componentes que junto al paradigma MapReduce componen a este framework.

3.2.1 HDFS (Hadoop Distributed File System)

HDFS es el sistema de archivos distribuido de Hadoop que permite almacenar grandes volúmenes de datos de manera fiable y eficiente. Diseñado para ejecutarse en hardware común, HDFS divide los datos en bloques grandes y los distribuye a través de varios nodos en un clúster, asegurando redundancia y tolerancia a fallos (Fernández, 2022). Este enfoque no solo mejora la disponibilidad de los datos, sino que también facilita la escalabilidad horizontal[6]. En un sistema HDFS típico, un archivo grande se fragmenta en bloques de tamaño fijo, generalmente 128 MB o 256

---

[6] La escalabilidad horizontal consiste en potenciar el rendimiento del sistema desde un aspecto de mejora global, a diferencia de aumentar la potencia de una única parte del mismo. https://www.juntadeandalucia.es/servicios/madeja/contenido/recurso/220

MB, y cada bloque se replica en múltiples nodos para garantizar la tolerancia a fallos. Según White (2015), “la replicación de datos en HDFS asegura que incluso si uno de los nodos falla, los datos todavía estarán disponibles en otros nodos del clúster”.

3.2.2 Job Tracker

El Job Tracker es el componente maestro de Hadoop, responsable de la gestión y coordinación de los jobs de MapReduce. Su función principal es recibir las solicitudes de trabajo de los clientes y dividir estos trabajos en tareas más pequeñas. Estas tareas son asignadas a los Task Trackers disponibles en el clúster. El Job Tracker monitoriza continuamente el progreso de las tareas, maneja la recuperación de fallos en caso de que algún nodo falle y reasigna las tareas fallidas a otros nodos, asegurando la correcta ejecución y finalización de los trabajos.

3.2.3 Task Tracker

Los Task Trackers son los componentes encargados de realizar el procesamiento de los datos en el clúster de Hadoop. Cada Task Tracker recibe tareas del Job Tracker y ejecuta las operaciones de Map y Reduce en los datos asignados. Los Task Trackers reportan su progreso y estado al Job Tracker de manera periódica. En caso de fallo de un Task Tracker, las tareas en curso son reasignadas a otros Task Trackers disponibles por el Job Tracker, garantizando la resiliencia y la tolerancia a fallos del sistema.

3.2.4 YARN (Yet Another Resource Negotiator)

YARN es un componente esencial de Hadoop que gestiona los recursos del clúster y permite la ejecución simultánea de múltiples aplicaciones. Introducido en Hadoop 2.0, YARN separa las funciones de gestión de recursos y programación de tareas, permitiendo una mayor flexibilidad y eficiencia en el uso de los recursos del clúster (Vavilapalli et al., 2013). YARN actúa como un sistema operativo para Hadoop, asignando recursos a las aplicaciones en función de sus necesidades y monitoreando su uso para garantizar que las tareas se completen de manera eficiente. Este sistema permite la ejecución simultánea de diferentes tipos de cargas de trabajo en el mismo clúster, optimizando el uso de los recursos y aumentando la productividad general del sistema. Según Vavilapalli et al. (2013), YARN transforma a Hadoop de un motor de procesamiento por lotes a una plataforma más general que puede ejecutar una variedad de aplicaciones de procesamiento de datos.

En versiones más recientes de Hadoop, el componente de gestión de recursos y ejecución de trabajos es YARN (Yet Another Resource Negotiator). YARN maneja los trabajos utilizando dos procesos principales: el Resource Manager y el Node

Manager. En el contexto de la ejecución de jobs, YARN reemplaza al Job Tracker y Task Tracker con el siguiente esquema:

- **Resource Manager (Job Tracker):** maneja todos los trabajos a ser procesados, teniendo en cuenta el mapa del clúster al momento de crear los procesos. Su función es similar a la del Job Tracker en las versiones anteriores de Hadoop.
- **Node Manager (Task Tracker):** son los encargados de realizar el procesamiento de los datos, ejecutando las tareas asignadas por el Resource Manager. Cada nodo en el clúster tiene un Node Manager que supervisa los contenedores de recursos en los que se ejecutan las tareas.

3.2.5 Apache Hive

Hive es una infraestructura de data warehouse construida sobre Hadoop que facilita el análisis y la consulta de grandes conjuntos de datos. Utiliza un lenguaje similar a SQL, denominado HiveQL, que permite a los usuarios ejecutar consultas sin necesidad de escribir programas MapReduce directamente (Thusoo et al., 2010). Hive convierte las consultas SQL en jobs MapReduce, simplificando el proceso de análisis de grandes volúmenes de datos para usuarios que están familiarizados con SQL. Esto reduce significativamente la barrera de entrada para los analistas de datos y los científicos de datos, permitiéndoles aprovechar la potencia de Hadoop sin necesidad de aprender un nuevo lenguaje de programación.

3.2.6 Apache Pig

Pig es una plataforma de alto nivel para la creación de programas que se ejecutan en Hadoop. Utiliza un lenguaje de scripting denominado Pig Latin, que permite a los desarrolladores escribir scripts de análisis de datos complejos de manera más sencilla y eficiente que utilizando Java para MapReduce (Olston et al., 2008). Pig Latin es un lenguaje de flujo de datos que incluye operadores para realizar tareas comunes como el filtrado, la unión y la agrupación de datos. La flexibilidad y simplicidad de Pig lo hacen ideal para prototipos rápidos y análisis ad-hoc, permitiendo a los usuarios enfocarse en la lógica de negocio sin preocuparse por los detalles de implementación en MapReduce. Como Olston et al. (2008) señalan, *“Pig busca un punto óptimo entre estos dos extremos, ofreciendo primitivas de manipulación de datos de alto nivel, como ‘proyección’ y ‘unión’, pero en un estilo mucho menos declarativo que SQL.”*.

3.2.7 Apache HBase

HBase es una base de datos NoSQL distribuida y orientada a columnas que se ejecuta sobre HDFS. Está diseñada para proporcionar acceso rápido y aleatorio a

grandes cantidades de datos estructurados, permitiendo operaciones de lectura y escritura de baja latencia (George, 2011). HBase se basa en el modelo de almacenamiento de Bigtable de Google y es ideal para aplicaciones que requieren lecturas y escrituras rápidas y consistentes en grandes volúmenes de datos. HBase soporta millones de operaciones por segundo y es capaz de manejar grandes cantidades de datos, escalando horizontalmente a medida que aumentan las necesidades de almacenamiento y procesamiento.

3.3 Etapas del Paradigma MapReduce

El paradigma MapReduce, fundamental para el procesamiento de grandes volúmenes de datos en un entorno distribuido, se divide en cuatro fases esenciales: Map, Shuffle, Sort y Reduce. Además, existe una función opcional de optimización, denominada Combine, tal como se observa en la Fig. 3.3.1, de la cual se llevará a cabo una explicación al final del capítulo.

Estas etapas trabajan en conjunto para transformar y consolidar datos de entrada en resultados significativos de una manera eficiente y escalable. A continuación, en la Fig. 3.3.1, puede verse como es el flujo de los datos en este paradigma, y cómo estos pasan de una etapa a otra.

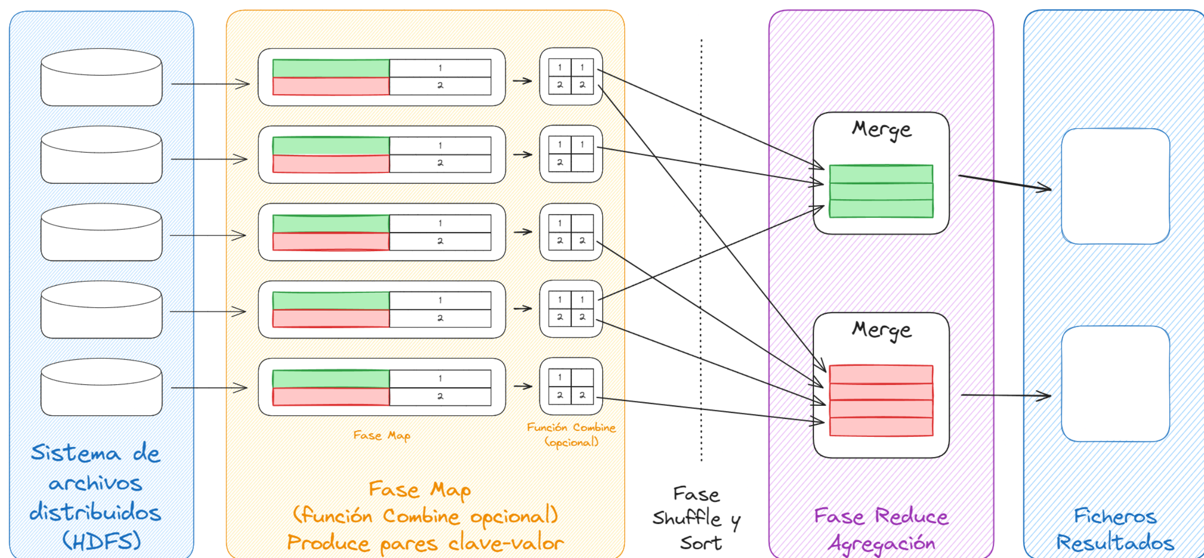


Fig. 3.3.1: Etapas del Paradigma MapReduce

3.3.1 Map

La fase Map es el primer paso en un job MapReduce. Durante esta fase, los datos de entrada se dividen en fragmentos más pequeños y se procesan en paralelo por múltiples nodos en el clúster. Cada Task Tracker ejecuta la función de mapeo definida por el usuario a los datos de entrada, generando pares clave-valor intermedios. Esta función puede ser personalizada según las necesidades

específicas de la aplicación. Según Dean y Ghemawat (2004), "*la función Map toma un par de entrada y produce un conjunto de pares intermedios. … . El framework de MapReduce agrupa todos los valores intermedios asociados con la misma clave intermedia y los pasa a la función Reduce.*" Este proceso permite la distribución y paralelización del procesamiento de datos, lo cual es esencial para manejar grandes volúmenes de información de manera eficiente.

3.3.2 Shuffle

La fase Shuffle es una de las operaciones internas más importantes en la ejecución de un job MapReduce. Durante esta fase, los pares clave-valor intermedios generados en la fase Map son transferidos a los nodos donde se ejecutarán las tareas de reducción. Esta transferencia se realiza de manera eficiente para minimizar el tráfico de red y garantizar que todos los valores asociados con una misma clave lleguen al mismo nodo de reducción. El shuffle es crítico para organizar los datos antes de la reducción, aunque es una operación interna y no requiere programación explícita por parte del usuario.

El proceso de shuffle implica la redistribución de datos entre los nodos para asegurar que los valores asociados con cada clave intermedia sean agrupados correctamente. Este agrupamiento es esencial para el funcionamiento eficiente de la fase Reduce, ya que permite que cada nodo de reducción procese todos los valores correspondientes a una clave específica de manera centralizada.

3.3.3 Sort

La fase Sort, que ocurre en paralelo con el shuffle, ordena los pares clave-valor intermedios por sus claves. Este ordenamiento es necesario para que la fase Reduce pueda procesar de manera eficiente los datos agrupados por claves. El framework MapReduce asegura que los datos estén correctamente ordenados antes de pasar a la función de reducción. La combinación de shuffle y sort garantiza que todos los datos estén correctamente preparados para la etapa final de reducción.

El ordenamiento de los datos en la fase Sort permite que el proceso de reducción se realice de manera más estructurada y eficiente. Esto es particularmente importante en aplicaciones donde el orden de los datos tiene un impacto significativo en los resultados finales. Por ejemplo, en un análisis de ventas, ordenar los datos por producto antes de la reducción permite calcular de manera precisa las ventas totales por producto.

3.3.4 Reduce

La fase Reduce es el paso final en el flujo de trabajo de MapReduce. En esta fase, los pares clave-valor intermedios, ya ordenados y transferidos, son procesados por

la función de reducción definida por el usuario. Esta función agrupa los valores asociados con cada clave intermedia y produce el resultado final para cada grupo de claves. Por ejemplo, en el caso de un análisis de frecuencia de palabras, la función Reduce tomaría todos los pares clave-valor con la misma clave (palabra) y sumaría los valores (cantidad de ocurrencias) para obtener el conteo total de cada palabra en el conjunto de datos original.

La fase Reduce consolida los datos intermedios en resultados finales que pueden ser almacenados o utilizados para análisis adicionales. La capacidad de manejar eficientemente grandes volúmenes de datos y producir resultados significativos es una de las principales fortalezas del paradigma MapReduce. Como destacan Dean y Ghemawat (2004), “*la función Reduce toma los resultados agrupados de la fase Map y los combina en un conjunto más pequeño de resultados finales”.*

3.3.5 Combine

La función Combine es un mecanismo de optimización opcional, pero altamente beneficioso en el proceso de MapReduce. Su objetivo es disminuir la carga de datos en la fase Reduce, al consolidar los resultados intermedios producidos por los mappers. La función Combine se ejecuta en el mismo nodo donde se realizó la tarea Map, trabajando exclusivamente con la salida producida por la misma antes de que los datos se escriban en el disco.

Al consolidar estos resultados de manera local en el nodo mapper, este mecanismo de optimización reduce la cantidad de datos que necesitan ser transferidos a través de la red a los nodos de reducción. Esto no solo mejora la eficiencia de la red, sino que también disminuye la carga de procesamiento en los nodos de reducción, acelerando el tiempo total de procesamiento. Hadoop no garantiza cuántas veces se invocará esta función, ya que internamente es solo una tarea de optimización. Esta función actúa como una mini-reducción local, mejorando significativamente la eficiencia del proceso MapReduce.

3.4 Ejemplo de Conteo de Palabras con MapReduce

En el siguiente ejemplo, se detalla la resolución del problema de Word Count mediante el enfoque MapReduce. El objetivo del problema es contar la frecuencia de aparición de cada palabra en un conjunto de frases. A continuación se analizará cada etapa del proceso.

Fase Map

En esta fase, los datos de entrada se dividen en fragmentos que son procesados por nodos mappers. Cada nodo mapper toma una parte del texto y emite pares

clave-valor. En nuestro ejemplo, consideramos tres nodos mappers que procesan las siguientes frases:

- **Nodo 1:** "tres llamas durmiendo"
- **Nodo 2:** "llamas a las tres"
- **Nodo 3:** "apaga las llamas de las tres velas"

Cada nodo mapper produce pares clave-valor para cada palabra encontrada en la frase, con la palabra como clave y el valor 1 indicando una ocurrencia de esa palabra:

- **Nodo 1:** ("tres", 1), ("llamas", 1), ("durmiendo", 1)
- **Nodo 2:** ("llamas", 1), ("a", 1), ("las", 1), ("tres", 1)
- **Nodo 3:** ("apaga", 1), ("las", 1), ("llamas", 1), ("de", 1), ("las", 1), ("tres", 1), ("velas", 1)

Fase Shuffle

En la fase Shuffle, los pares clave-valor generados por los nodos mappers son redistribuidos y agrupados por clave. El objetivo es reunir todas las instancias de cada clave (en este caso, palabras) en un solo lugar. Una vez finalizada la fase Shuffle, los pares clave-valor agrupados son:

- **Clave "tres":** [1 (Nodo1), 1 (Nodo 2), 1 (Nodo 3)]
- **Clave "llamas":** [1 (Nodo1), 1 (Nodo 2), 1 (Nodo 3)]
- **Clave "durmiendo":** [1 (Nodo 2)]
- **Clave "a":** [1 (Nodo 2)]
- **Clave "las":** [1 (Nodo 2), 1 (Nodo 2), 1 (Nodo 3)]
- **Clave "apaga":** [1 (Nodo 3)]
- **Clave "de":** [1 (Nodo 3)]
- **Clave "velas":** [1 (Nodo 3)]

Fase Sort

Durante la fase Sort, los pares clave-valor se ordenan por clave. Esto asegura que todas las tuplas que continúen hacia la fase reduce lo hagan ordenadas por clave, facilitando la agregación de valores en la siguiente fase. El resultado de la fase de Sort es:

- **Clave "a":** [1]
- **Clave "apaga":** [1]
- **Clave "de":** [1]
- **Clave "durmiendo":** [1]
- **Clave "las":** [1, 1, 1]
- **Clave "llamas":** [1, 1, 1]
- **Clave "tres":** [1, 1, 1]
- **Clave "velas":** [1]

Fase Reduce

En la fase Reduce, se agrupan y suman los valores asociados a cada clave. Esta fase combina las instancias que se le asignaron de cada clave, sumando los valores para obtener el conteo total de cada palabra. El resultado final de la fase Reduce es:

- **Clave "a":** 1
- **Clave "apaga":** 1
- **Clave "de":** 1
- **Clave "durmiendo":** 1
- **Clave "las":** 1 + 1 + 1 = 3
- **Clave "llamas":** 1 + 1 + 1 = 3
- **Clave "tres":** 1 + 1 + 1 = 3
- **Clave "velas":** 1

Esto indica que las palabras "tres", "llamas", y "las" aparecen 3 veces cada una en el conjunto de frases procesadas, mientras que las palabras "durmiendo", "a", "apaga", "de", y "velas" aparecen una vez cada una.

Optimización con Combine

Para optimizar el proceso, se puede utilizar una función Combine en los nodos mappers. La función Combine realiza una suma local de las ocurrencias de cada clave antes de enviar los datos a la fase Reduce. Esto reduce la cantidad de datos que deben ser transferidos entre las fases Map y Reduce, mejorando la eficiencia del sistema.

3.5 Conclusión

En resumen, las etapas del paradigma MapReduce — Map, Shuffle, Sort y Reduce — junto con la función opcional Combine, constituyen un marco robusto y eficiente para el procesamiento de grandes volúmenes de datos. Estas etapas, ejecutadas de manera distribuida y paralela, permiten transformar datos brutos en información valiosa, facilitando la toma de decisiones informadas en diversas aplicaciones industriales y científicas.

Capítulo 4: Sistemas Distribuidos y Comunicación en Tiempo Real

En este capítulo, se examinarán los conceptos fundamentales de los sistemas distribuidos y los mecanismos de comunicación en tiempo real utilizados en la web. Estos conceptos son cruciales para entender cómo se logrará la comunicación efectiva entre los nodos del clúster en la aplicación propuesta. Se abordarán tecnologías como WebSockets y WebRTC, y su aplicación en sistemas distribuidos, proporcionando una base sólida para la implementación técnica del proyecto.

4.1 Explicación de Sistemas Distribuidos

Los sistemas distribuidos son una arquitectura clave en la informática moderna, caracterizada por la distribución de componentes de software y hardware a través de múltiples computadoras conectadas por una red. Esta configuración permite que los sistemas funcionen de manera coordinada y coherente, aunque sus componentes estén físicamente separados. Según Tanenbaum y Van Steen (2017), "*un sistema distribuido es un conjunto de computadoras independientes que aparece ante sus usuarios como un único sistema coherente*". Esta definición destaca la ilusión de unicidad y coherencia que es fundamental para la operativa de estos sistemas.

En un sistema distribuido, los recursos de cómputo, almacenamiento y datos están dispersos entre varias máquinas, lo cual proporciona una serie de beneficios, como la escalabilidad, la resiliencia y la disponibilidad.

La **escalabilidad** se refiere a la capacidad del sistema para manejar un aumento en la carga de trabajo al agregar más recursos. Esto es esencial en aplicaciones que experimentan un crecimiento constante en la cantidad de datos y usuarios, como es el caso en el contexto de Big Data.

La **resiliencia**, por otro lado, es la capacidad de un sistema distribuido para continuar funcionando a pesar de fallos en algunos de sus componentes. Esta característica es crítica, ya que reduce el riesgo de interrupciones del servicio. Para lograr esto, se implementan diversas técnicas de tolerancia a fallos, como la replicación de datos y la conmutación por error. Según Coulouris et al. (2012), el diseño de sistemas distribuidos debe considerar la incertidumbre y la variabilidad inherentes en la infraestructura de red y hardware.

La **disponibilidad** se refiere a la accesibilidad del sistema y sus servicios en todo momento. Un sistema distribuido bien diseñado asegura que los recursos estén disponibles y accesibles para los usuarios, minimizando el tiempo de inactividad. Para ello, es fundamental contar con una infraestructura de red robusta y una

estrategia de replicación de datos efectiva, que garantice que los datos sean accesibles incluso en caso de fallos de hardware.

4.1.1 Clasificación

Los sistemas distribuidos se pueden clasificar en diferentes tipos según su arquitectura y propósito.

En el ámbito de la **computación en clúster**, un grupo de computadoras trabaja de manera conjunta para ejecutar tareas que requieren grandes cantidades de procesamiento. Este modelo es ampliamente utilizado en aplicaciones científicas y de ingeniería que demandan alto rendimiento.

La **computación en la nube**, por otro lado, ofrece servicios de cómputo a través de Internet, permitiendo a las organizaciones escalar sus recursos de manera flexible y pagar solo por lo que usan. Como nos rectifican y explican, Armbrust et al. (2010) en su artículo, la computación en la nube promete una reducción significativa en el costo de la TI al ofrecer una infraestructura de servicios compartidos.

En los **sistemas peer-to-peer**, cada nodo actúa simultáneamente como cliente y servidor, compartiendo recursos directamente entre sí sin necesidad de un servidor central. Este modelo es común en aplicaciones de intercambio de archivos y comunicaciones descentralizadas.

Finalmente, los **sistemas distribuidos de archivos**, como el Hadoop Distributed File System (HDFS), se utilizan para almacenar y gestionar grandes volúmenes de datos de manera distribuida, facilitando el acceso y procesamiento de estos datos a través de múltiples nodos.

El diseño y la implementación de sistemas distribuidos requiere una comprensión profunda de diversos desafíos:

- La **coherencia de datos**, crucial para asegurar que todos los nodos del sistema tengan una visión consistente de los datos.
- La **sincronización de procesos**, necesaria para coordinar las acciones de los diferentes nodos y garantizar que las tareas se completen de manera ordenada.
- La **seguridad**, aspecto vital para proteger los datos y las comunicaciones entre los nodos del sistema.
- La **gestión de la red**, centrada en mantener una conectividad confiable y eficiente entre los componentes distribuidos.

En resumen, los sistemas distribuidos son una pieza fundamental en la infraestructura de TI moderna, permitiendo la creación de aplicaciones robustas, escalables y altamente disponibles. La comprensión de sus principios y desafíos es esencial para el desarrollo y despliegue de soluciones tecnológicas avanzadas, como la herramienta de simulación de clústers MapReduce propuesta en esta tesina.

4.2 Mecanismos de Comunicación en Tiempo Real para la Web

En los sistemas distribuidos, la comunicación eficiente entre los nodos es crucial para asegurar un rendimiento óptimo y la coherencia de los datos. Los mecanismos de comunicación en tiempo real han evolucionado significativamente, permitiendo una interacción casi instantánea entre los componentes del sistema distribuido. Este apartado aborda dos de las tecnologías más destacadas en este campo: WebSockets y WebRTC.

4.2.1 WebSockets

WebSockets es una tecnología que proporciona canales de comunicación bidireccionales sobre una única conexión TCP. Esta tecnología es especialmente útil para aplicaciones que requieren actualizaciones en tiempo real, como chats en línea, juegos multijugador y paneles de control en vivo. Podemos afirmar que "*El objetivo de esta tecnología es proporcionar un mecanismo para aplicaciones basadas en navegador que necesitan comunicación bidireccional con servidores que no dependan de la apertura de múltiples conexiones HTTP*" (Fette & Melnikov, 2011).

La implementación de WebSockets comienza con un handshake HTTP entre el cliente y el servidor, después del cual se establece una conexión persistente. Esta conexión permite la transmisión de datos en ambas direcciones sin la sobrecarga de las cabeceras HTTP. Fette y Melnikov (2011) destacan que la reducción de latencia es crucial para la eficiencia de aplicaciones distribuidas que manejan grandes cantidades de datos.

En el contexto de nuestra aplicación, WebSockets se utiliza predominantemente para el proceso de signaling, esencial para establecer las conexiones con WebRTC. Esta tecnología gestiona la inicialización de la comunicación en tiempo real, garantizando que los datos de configuración se transmitan con la mínima latencia, lo cual es crucial para mantener una conexión eficiente y sin demoras.

4.2.2 WebRTC

WebRTC (Web Real-Time Communication) es otra tecnología que permite la comunicación en tiempo real, pero está diseñada específicamente para la transmisión de audio, video y datos entre navegadores sin necesidad de intermediarios. Según Loreto y Romano (2014), "*Por primera vez, los navegadores pueden intercambiar directamente contenido de tiempo real con otros navegadores de igual a igual*". Esta característica lo convierte en una herramienta poderosa para aplicaciones que requieren comunicación directa entre pares (peer-to-peer).

WebRTC utiliza varios componentes clave para lograr la comunicación en tiempo real, incluyendo RTCPeerConnection, RTCDataChannel y MediaStreams. RTCPeerConnection es responsable de establecer y gestionar la conexión entre pares, mientras que RTCDataChannel permite la transmisión de datos arbitrarios con baja latencia. MediaStreams gestiona la captura y transmisión de audio y video. La capacidad de WebRTC para manejar datos complejos y mantener conexiones estables entre pares lo convierte en una herramienta valiosa para sistemas distribuidos (Loreto & Romano, 2014).

En el marco de nuestra aplicación, WebRTC se implementa para manejar la transmisión de datos más robusta y flexible, garantizando una comunicación directa y eficiente entre los nodos del clúster.

En este sistema, los nodos intercambian no sólo datos de texto, sino que también, a través de este protocolo, se maneja toda la sincronización necesaria para el funcionamiento y la ejecución de los trabajos distribuidos. La capacidad de WebRTC para gestionar estos intercambios de manera eficiente es crucial para mantener una comunicación fluida y sin interrupciones en la arquitectura de sistemas distribuidos modernos.

4.2.3 Integración de WebSockets y WebRTC en la Aplicación Propuesta

La integración de WebSockets y WebRTC en la aplicación propuesta permitirá una comunicación efectiva y eficiente entre los nodos del clúster. WebSockets se utilizará principalmente para el signaling, asegurando que la configuración inicial de las conexiones WebRTC sea rápida y fiable. Por otro lado, WebRTC se implementará para manejar la transmisión de datos más complejos y garantizar que la comunicación entre nodos sea robusta y flexible.

Esta combinación proporciona una base sólida para la aplicación distribuida propuesta, permitiendo una coordinación y comunicación eficaz entre los nodos del clúster. La integración de estas tecnologías es esencial para emular un entorno de

Big Data distribuido, proporcionando a los estudiantes una experiencia práctica y realista en el manejo de datos en sistemas distribuidos.

En conclusión, la integración de tecnologías de comunicación en tiempo real como WebSockets y WebRTC es fundamental para el éxito del sistema distribuido propuesto en esta tesis. Estas tecnologías no solo facilitan la transmisión rápida y eficiente de datos, sino que también aseguran que los nodos del clúster puedan trabajar de manera coherente y sincronizada, intentando simular un entorno de ejecución de jobs MapReduce.

Capítulo 5: Investigación previa y Soluciones Actuales

El presente capítulo aborda la investigación previa y las soluciones actuales para la ejecución de jobs MapReduce, enfocándose en las tecnologías y métodos disponibles para ejecutar código Python en el navegador y en cómo estas herramientas pueden aplicarse en un contexto educativo. A través de esta revisión, se busca identificar los problemas que debíamos solucionar y evaluar las alternativas existentes para finalmente justificar la elección de ejecutar MapReduce en el navegador como la mejor opción.

En primer lugar, se analizan diversas herramientas que permiten la ejecución de Python en el navegador, como Pyodide, Brython, Skulpt y Transcrypt. Cada una de estas herramientas ofrece diferentes ventajas y limitaciones que influyen en su adecuación para distintos contextos de aplicación. La elección de la herramienta adecuada se basa en factores como la facilidad de configuración, la compatibilidad con bibliotecas existentes y el rendimiento en entornos web.

Posteriormente, se presenta el código base para la implementación de las funciones map, combine y reduce en el navegador, destacando la necesidad de contar con una estructura que gestione eficientemente la distribución de tareas y el manejo de resultados intermedios. La integración de esta lógica en un entorno basado en React y Pyodide se justifica por su flexibilidad y capacidad para simular un entorno de ejecución robusto y eficiente.

Finalmente, se revisan otras soluciones existentes para la ejecución de jobs MapReduce, incluyendo implementaciones tradicionales como Hadoop y alternativas basadas en la nube como Amazon EMR y Google Cloud Dataflow. Estas soluciones, aunque potentes, presentan desafíos en términos de complejidad de configuración y costos, lo que las hace menos adecuadas para un entorno educativo donde la simplicidad y la accesibilidad son prioritarias.

En conclusión, este capítulo establece las bases para comprender por qué la ejecución de MapReduce en el navegador se presenta como una solución viable y adecuada para la enseñanza del paradigma MapReduce.

5.1 Soluciones Actuales para la Ejecución de Jobs MapReduce

En la actualidad, la ejecución de Jobs MapReduce es un aspecto crucial en el procesamiento de grandes volúmenes de datos. Este subcapítulo se enfoca en las herramientas utilizadas durante el dictado de la asignatura y otras soluciones existentes en la industria para la ejecución de jobs MapReduce.

5.1.1 Herramienta Utilizada en el Dictado de la Asignatura

Para que los alumnos puedan resolver problemas utilizando el paradigma MapReduce, la cátedra dispone de un módulo escrito en Python denominado MRE (emulador MapReduce).

Este módulo define una clase **Job**, la cual al momento de instanciar requiere de cuatro parámetros obligatorios:

1. **Directorio de Entrada**: La ruta del directorio donde se encuentran los datos de entrada, generalmente una serie de archivos de texto en formato TSV (valores separados por tabulación).
2. **Directorio de Salida**: La ruta del directorio donde se desea escribir el resultado final.
3. **Función *map***: La función *mapeo* que se desea ejecutar sobre los datos de entrada.
4. **Función *reduce***: La función de *reducción* correspondiente.

Además, se pueden especificar funciones adicionales mediante métodos de instancia:

- **setCombiner**: Para definir la función *combine*.
- **setShuffleCmp**: Para definir la función de comparación para el *shuffle*.
- **setSortCmp**: Para definir la función de comparación para el *sort*.

La función *map* recibe tres parámetros:

- **key**: El primero de los valores separados por tabulación, o el número de línea (offset) si solo hay un valor.
- **value**: El resto de los valores separados por tabulación.
- **context**: Instancia de la clase “Context” (definida en el módulo MRE), utilizada para escribir resultados mediante el método “write” y para acceder a parámetros globales compartidos por todos los nodos del clúster.

A continuación se presenta un ejemplo sencillo para resolver el problema de "Word Count" (Contador de Palabras), donde la entrada es un conjunto de archivos que contienen diferentes palabras, y la salida esperada es una lista que indica la cantidad de veces que aparece cada palabra en todo el conjunto de datos.

```
directorio_entrada = "/WordCount/input/"
directorio_salida = "/WordCount/output/"

def fmap(key, value, context):
    palabras = value.split()  # separa las palabras por espacio
    for palabra in palabras:  # itera por cada una de las palabras
         context.write(palabra, 1)   # escribe (palabra, 1), indicando que dicha
palabra se encontró una vez

def fred(key, values, context):
    total_palabra = 0  # inicializa un totalizador en 0 para la palabra
    for subtotal in values:  # itera por cada subtotal generado en la etapa map
        total_palabra += subtotal  # actualiza el total
    context.write(key, total_palabra)  # escribe (palabra, total_palabra)

# instancia el job con los parámetros correspondientes
job = Job(directorio_entrada, directorio_salida, fmap, fred)

# utiliza la misma función de reducción como función combine
job.setCombiner(fred)

# inicia el procesamiento
success = job.waitForCompletion()

# el resultado de la ejecución se encuentra disponible en el directorio de
salida especificado
```

Esta herramienta proporciona a los estudiantes un marco de trabajo sencillo y efectivo para entender y aplicar el paradigma MapReduce en la resolución de diferentes problemas.

5.1.2 Otras Soluciones Existentes

La ejecución de Jobs MapReduce puede abordarse de diversas maneras, cada una con sus propias ventajas y desafíos. A continuación, se presentan algunas de las soluciones más comunes y ampliamente adoptadas en la industria y la academia:

5.1.2.1 Implementación de Hadoop con un Clúster de Computadoras

Apache Hadoop es una de las implementaciones más conocidas del paradigma MapReduce. Permite el procesamiento distribuido de grandes conjuntos de datos a través de un clúster de computadoras.

Proceso de Implementación

1. *Configuración del Clúster*: Se debe configurar un clúster de computadoras, cada una funcionando como nodo. Este proceso incluye la instalación de Hadoop en cada nodo y la configuración de las redes y sistemas de almacenamiento compartido.
2. *Configuración del HDFS (Hadoop Distributed File System)*: HDFS se utiliza para almacenar los datos de entrada y salida de manera distribuida y redundante, lo que garantiza la fiabilidad y disponibilidad de los datos.
3. *Escritura de Jobs MapReduce*: Los jobs MapReduce se escriben en Java (o mediante APIs de otros lenguajes) y se ejecutan en el clúster, donde Hadoop se encarga de distribuir las tareas de map y reduce entre los nodos.

Ventajas

- *Escalabilidad*: Hadoop puede manejar petabytes de datos distribuidos en cientos o miles de nodos.
- *Fiabilidad*: HDFS proporciona almacenamiento redundante, asegurando que los datos no se pierdan incluso si fallan algunos nodos.
- *Ecosistema Rico*: Hadoop tiene un ecosistema extenso que incluye herramientas como Hive, Pig, y HBase, que facilitan tareas de análisis y almacenamiento.

Desventajas

- *Complejidad*: La configuración y administración de un clúster Hadoop pueden ser complejas y requieren conocimientos avanzados.
- *Costos de Infraestructura*: Requiere una infraestructura considerable, tanto en términos de hardware como de administración.
- *Eficiencia para Trabajos Pequeños*: Puede ser ineficiente para trabajos pequeños debido a la sobrecarga de gestión y configuración.

5.1.2.2 Soluciones Basadas en la Nube

Las soluciones en la nube ofrecen una alternativa más sencilla y escalable para la ejecución de jobs MapReduce, eliminando muchas de las complejidades de configuración y administración de la infraestructura física.

Amazon EMR (Elastic MapReduce)

Amazon EMR es un servicio gestionado de Amazon Web Services (AWS) que permite el procesamiento de grandes volúmenes de datos utilizando Hadoop, Spark, y otros marcos de procesamiento.

Ventajas

- *Escalabilidad Dinámica*: Permite escalar el clúster hacia arriba o hacia abajo según sea necesario.
- *Integración con AWS:* Se integra bien con otros servicios de AWS, como S3, DynamoDB, y Redshift, facilitando la gestión de datos.
- *Administración Simplificada*: AWS se encarga de la administración de la infraestructura, lo que reduce la carga administrativa.

Desventajas

- *Costo*: Puede ser costoso para cargas de trabajo intensivas y continuas.
- *Dependencia de la Nube*: Depende de la infraestructura de AWS, lo que puede no ser adecuado para todos los casos de uso.

Google Cloud Dataflow

Google Cloud Dataflow es un servicio gestionado para el procesamiento de datos en tiempo real y por lotes, basado en el modelo de programación Apache Beam.

Ventajas

- *Procesamiento Unificado*: Permite la ejecución de pipelines de datos tanto en tiempo real como por lotes.
- *Infraestructura Gestionada*: Google se encarga de la administración de la infraestructura.
- *Integración con Google Cloud*: Se integra bien con otros servicios de Google Cloud, como BigQuery y Cloud Storage.

Desventajas

- *Costo*: Puede ser costoso, especialmente para grandes volúmenes de datos.
- *Curva de Aprendizaje*: Requiere familiaridad con el modelo de programación Apache Beam y la plataforma Google Cloud.

5.1.2.3 Soluciones Híbridas y Alternativas

Las soluciones híbridas y alternativas ofrecen enfoques variados para la ejecución de jobs MapReduce, combinando características de diferentes paradigmas y tecnologías para abordar las limitaciones de las implementaciones tradicionales. Estas soluciones no solo implementan el modelo de programación MapReduce, sino que también integran capacidades adicionales que permiten un procesamiento de datos más flexible, rápido y eficiente. A continuación, se detallan algunas de estas soluciones, proporcionando una visión general de sus características y capacidades.

Apache Spark

Aunque no es una implementación pura de MapReduce, Apache Spark soporta el modelo de programación MapReduce y ofrece un motor de procesamiento unificado para operaciones en memoria.

Ventajas

- *Velocidad*: Spark es significativamente más rápido que Hadoop para muchas tareas debido a su capacidad para realizar operaciones en memoria.
- *Flexibilidad*: Permite operaciones complejas y análisis de datos avanzados.
- *Ecosistema*: Spark tiene un ecosistema rico con bibliotecas para SQL, machine learning, y procesamiento de gráficos.

Desventajas

- *Consumo de Memoria*: Puede requerir grandes cantidades de memoria, lo que puede ser un desafío para infraestructuras limitadas.
- *Curva de Aprendizaje*: La curva de aprendizaje puede ser pronunciada para usuarios nuevos.

Dask

Dask es una biblioteca de Python para el paralelismo avanzado que permite el procesamiento distribuido utilizando el paradigma MapReduce y otros modelos de programación.

Ventajas

- *Integración con Python*: Se integra bien con el ecosistema de Python, facilitando su uso para desarrolladores familiarizados con este lenguaje.
- *Escalabilidad*: Permite escalar desde un solo servidor hasta clústers de alta capacidad.

- *Simplicidad*: Más fácil de configurar y usar en comparación con Hadoop.

Desventajas

- *Ecosistema Menos Maduro*: No tiene un ecosistema tan amplio y maduro como Hadoop o Spark.
- *Rendimiento*: Puede no ser tan eficiente como Spark para ciertas tareas de gran escala.

Estas soluciones muestran la diversidad de enfoques disponibles para la ejecución de jobs MapReduce, cada una con sus propias ventajas y desafíos. Sin embargo, muchas de estas soluciones pueden ser considerablemente costosas, ya sea en términos de configuración, implementación o costos monetarios. En el contexto educativo, donde el objetivo principal es enseñar a los estudiantes los principios fundamentales del paradigma MapReduce, resulta impráctico emplear soluciones que requieran una configuración compleja o una inversión significativa en infraestructura.

Por lo tanto, se busca una alternativa más accesible y de menor escala, que fuera adecuada para la enseñanza y demostración del funcionamiento del framework MapReduce. Por ello es que la intención es la de desarrollar una herramienta que permita a los estudiantes comprender cómo se distribuye el trabajo entre los nodos y cómo se ejecutan las distintas fases del proceso, todo esto minimizando la necesidad de configuración y evitando la instalación de software complejo. Es así que surge la idea de llevar a cabo las ejecuciones de estos jobs en un ambiente común a todos los estudiantes y de fácil acceso, los navegadores. A continuación, explicaremos y profundizaremos en herramientas y desafíos existentes que esto conlleva.

5.2 Ejecutar Python en el Navegador

Como se mencionó anteriormente, el marco de trabajo que se utiliza en la cátedra para resolver problemas aplicando el paradigma MapReduce, implica utilizar un módulo escrito en Python, por ende, para facilitar el aprendizaje y transición entre una herramienta y otra, nuestro sistema tiene que ser capaz de ejecutar código Python en el navegador. A su vez, esto también posibilita experimentar y prototipar soluciones de manera más rápida.

Ejecutar Python en el navegador es una alternativa que ha ganado relevancia en los últimos años, gracias al desarrollo de nuevas tecnologías y a la creciente necesidad de herramientas que permitan la programación y ejecución de código de manera más accesible y flexible.

El surgimiento de herramientas que permiten la ejecución de Python en el navegador abrió nuevas posibilidades para la creación de aplicaciones interactivas y dinámicas, sin la necesidad de configurar entornos de desarrollo complejos o dependencias externas. Esta capacidad es particularmente útil en contextos educativos, de prototipado rápido y en entornos donde la accesibilidad y la portabilidad son cruciales.

Para ejecutar código Python en el navegador, existen varias herramientas y métodos, cada una con sus propias ventajas y desventajas. A continuación, se detallan algunas de las más destacadas:

5.2.1 Pyodide

Pyodide es una implementación de Python para WebAssembly que permite ejecutar Python en el navegador con una amplia gama de bibliotecas científicas, como NumPy, Pandas y Matplotlib. Pyodide es especialmente útil para aplicaciones que requieren cálculos científicos y análisis de datos en tiempo real. (The Pyodide development team, 2021)

<u>Ventajas</u>

- Permite la ejecución de código Python completo en el navegador.
- Soporta una amplia gama de bibliotecas científicas.
- Facilita la creación de aplicaciones interactivas para análisis de datos.

<u>Desventajas</u>

- El rendimiento puede no ser tan alto como en entornos nativos debido a la sobrecarga de WebAssembly.
- La inicialización puede ser lenta debido a la carga de bibliotecas grandes.

5.2.2 Brython

Brython es una implementación de Python 3 que se ejecuta en el navegador y permite interactuar con el DOM (Document Object Model) de manera similar a JavaScript. Brython es ideal para crear aplicaciones web interactivas y dinámicas utilizando Python. (Brython Documentation, n.d.)

<u>Ventajas</u>

- Facilita la integración de Python en aplicaciones web.
- Proporciona una sintaxis familiar para los desarrolladores de Python.

- Permite el uso de bibliotecas estándar de Python para manipulación del DOM.

Desventajas

- No es compatible con todas las bibliotecas de Python.
- El rendimiento puede ser inferior al de aplicaciones JavaScript nativas.

5.2.3 Skulpt

Skulpt es un intérprete de Python escrito en JavaScript que se ejecuta en el navegador. Skulpt es ligero y adecuado para fines educativos y pequeños proyectos que no requieren bibliotecas externas complejas. (Skulpt Documentation, 2015)

Ventajas

- Ligero y fácil de integrar en aplicaciones web.
- Ideal para fines educativos y aprendizaje de Python.
- Permite ejecutar la mayoría del código Python básico.

Desventajas

- Limitado en términos de soporte para bibliotecas externas.
- No está optimizado para aplicaciones complejas o que requieren alto rendimiento.

5.2.4 Transcrypt

Transcrypt es una herramienta que compila código Python a JavaScript, permitiendo la integración de Python con aplicaciones web tradicionales. Transcrypt es útil para desarrolladores que desean escribir lógica en Python pero necesitan interoperabilidad con el ecosistema JavaScript. (Transcrypt Documentation, 2014)

Ventajas

- Proporciona interoperabilidad con el ecosistema JavaScript.
- Facilita la escritura de lógica compleja en Python para aplicaciones web.
- Genera código JavaScript eficiente y optimizado.

Desventajas

- Requiere un paso de compilación adicional.

- No soporta todas las características de Python, especialmente aquellas dependientes de C-extensions.

Estas herramientas representan una evolución significativa en la capacidad de ejecutar Python en el navegador, cada una ofreciendo diferentes beneficios y limitaciones que deben ser considerados según el contexto de la aplicación. La elección de la herramienta adecuada dependerá de los requisitos específicos del proyecto, incluyendo la complejidad del código, la necesidad de bibliotecas externas, y las restricciones de rendimiento y compatibilidad.

Finalmente, en nuestra aplicación optamos por utilizar una biblioteca de React, denominada *react-py*[7], la cual en su núcleo emplea Pyodide. Esta decisión se basó en la capacidad de Pyodide para ejecutar un entorno completo y aislado de Python en el navegador, lo cual es crucial para nuestros requerimientos.

De esta manera, se eliminan las barreras de entrada asociadas a la configuración de un nuevo entorno de desarrollo, por lo que el usuario final ya no debe preocuparse por el trabajo que conlleva esto, sino únicamente en escribir el código que desea ejecutar.

Además, Pyodide ofrece una gran flexibilidad al permitir la utilización de numerosas bibliotecas de Python, facilitando así la expansión y evolución del sistema conforme se vayan necesitando nuevas funcionalidades. Esta elección nos ha permitido simular un entorno de ejecución robusto y eficiente, satisfaciendo las necesidades específicas de nuestro proyecto.

5.3 Análisis y Aplicación de Código Base para Funciones Map, Combine y Reduce

La implementación de un sistema MapReduce en el navegador presenta varios desafíos técnicos, entre los cuales se encuentra la necesidad de desarrollar un código base que permita ejecutar las funciones *map*, *combine* y *reduce* de manera eficiente y en el orden correcto.

5.3.1 Necesidad de un Código Base

Para lograr una ejecución eficiente de un job MapReduce en un entorno distribuido basado en el navegador, es esencial contar con un código base que facilite la integración y ejecución de las funciones *map*, *combine* y *reduce* definidas por el Job Tracker, asegurándose que los resultados intermedios y finales se manejen de manera coherente.

[7] Librería de React que permite la ejecución de Python en el navegador. https://elilambnz.github.io/react-py/

El código base se implementó en Python, aprovechando su simplicidad y la amplia disponibilidad de bibliotecas que facilitan el desarrollo y la prueba de funciones. La lógica de la aplicación y la distribución de datos entre los nodos se manejan en React, utilizando JavaScript, lo que permite una integración fluida y eficiente entre las distintas partes del sistema.

5.3.2 Implementación del Código Base

El módulo **MapReduceJob** es el que será utilizado en los navegadores por cada uno de los nodos unidos al clúster, a través de Pyodide. Este módulo define una clase MapReduceJob, que tiene como objetivo representar el estado interno de ejecución del algoritmo MapReduce, ejecutar cada etapa, calcular estadísticas locales de cada nodo e informar los resultados de la ejecución.

A continuación, se explicará en detalle cada una de las variables y métodos de instancia de la correspondiente clase **MapReduceJob**. En caso de querer profundizar en el propio código, puede dirigirse al **Anexo 4: Código del módulo de Python MapReduceJob**, para más detalle.

Variables de instancia

- *map_results, combine_results y reduce_results*

  Diccionarios que almacenan las claves generadas durante cada etapa, junto a la lista de valores asociados a dicha clave.

- *current_results*

  Utilizado para simplificar el referenciamiento al diccionario correspondiente a cada etapa al momento de escribir claves.

- *statistics*

  Diccionario que almacena las estadísticas obtenidas para cada etapa, tales como la cantidad de invocaciones, el tamaño de la entrada procesada, el tiempo total de procesamiento de la etapa, y el tamaño de la salida generada.

- *input_size*

  Tamaño de la entrada procesada en la etapa.

- *invocations*

  Cantidad de invocaciones de la función correspondiente a la etapa.

- *execute_phase*

Booleano utilizado para indicar si la función combine debe ejecutarse (en base a si se especificó código para dicha función o no).

Métodos de instancia

- *write*

  Método que recibe como argumento la clave y el valor a escribir. Este método simplemente incorpora la clave y valor recibido al diccionario de la etapa correspondiente, es decir, crea una entrada nueva en el diccionario en caso de que la clave sea nueva, o agrega el valor recibido a la lista de valores asociados en caso de que la clave exista.

- *map*

  Ejecuta la etapa map. Para lograr esto, por cada línea del archivo de entrada, ejecuta la función map (*fmap*), definida por el Job Tracker, enviando como argumento la línea procesada. Al finalizar se establece que *input_size* es igual al tamaño del archivo procesado e *invocations* a la cantidad de líneas de dicho archivo.

- *combine*

  Este mecanismo de optimización se ejecuta si ha sido especificado previamente. Al finalizar la etapa map, se invoca la función combine (*fcomb*) definida por el Job Tracker en el mismo nodo donde se ejecutó la etapa. Esta función recibe como parámetro cada par clave-valor generado por el propio nodo durante la etapa map. Si se ejecutó el combine, el número de invocaciones será igual a la cantidad de claves generadas en la etapa map; en caso contrario, será 0.

- *reduce*

  Ejecuta la etapa reduce. Para lograr esto, invoca a la función reduce (*fred*), definida por el Job Tracker, suministrándole como parámetro cada par clave-valor, del conjunto de claves que le haya asignado el Job Tracker. Al finalizar se establece que *input_size* es igual al tamaño del archivo que contiene las claves asignadas e *invocations* a la cantidad de claves.

- *execute*

  Ejecuta el Job MapReduce. Para lograr esto, en primer lugar ejecuta la etapa map y posteriormente el mecanismo de optimización combine (en caso de que se haya especificado), luego guarda las estadísticas en un archivo json y elimina archivos temporales. Finalmente, cuando

se hayan definido las claves a reducir por el nodo, ejecuta la etapa reduce, guarda las estadísticas y elimina archivos temporales.

En el módulo también se definen diferentes **helpers** que son utilizados por la clase MapReduceJob:

- *log_error*

  Guarda en un archivo el detalle de una excepción, informando la etapa en la cuál fue capturada.

- *load_json*

  Retorna el contenido y tamaño de un archivo JSON.

- *read_code*

  Ejecuta el código definido en un archivo y retorna si está vacío. Este helper se utiliza para leer el código *fmap*, *fcomb* y *fred* definido por el Job Tracker.

- *write_keys*

  Escribe un diccionario en un archivo JSON, convirtiendo las claves de tipo tupla a strings, y retorna el tamaño del archivo.

- *save_statistics*

  Guarda las estadísticas generadas en un archivo JSON.

- *clean_up*

  Elimina el archivo temporal que almacena los posibles errores ocurridos durante la ejecución.

- *safe_execute*

  Decorador que se utiliza para ejecutar la etapa map, reduce y la función combine. Los decoradores son utilizados para agregar comportamiento a una función. En este caso, previamente a la ejecución de cada etapa, se utiliza para inicializar el valor de la variable *execute_phase* leyendo el código correspondiente a la etapa, setear la variable *current_results* a los resultados de la etapa actual e iniciar un contador de tiempo. Luego, una vez ejecutada la etapa, se setean las estadísticas obtenidas, se escriben los resultados de la etapa y se muestra un mensaje de éxito. A su vez, se capturan posibles errores que puedan ocurrir durante la ejecución, para que puedan ser informados al usuario.

5.3.3 Integración con React

Mientras que el código base en Python maneja la lógica de las funciones *map*, *combine* y *reduce*, el manejo de estados de la aplicación y la distribución de datos entre los nodos se realiza en React, utilizando JavaScript. Esta integración permite una gestión eficiente de la comunicación entre nodos y asegura que los datos se transfieran y procesen de manera coherente.

La implementación de este código base, junto con la elección de Pyodide y React, nos permite crear un entorno de ejecución eficiente y flexible, adecuado para la ejecución de funciones MapReduce en un clúster distribuido basado en el navegador.

En el próximo capítulo, se presentará la implementación específica de la aplicación propuesta, proporcionando un análisis detallado de la arquitectura, así como todos los aspectos relacionados con el diseño y desarrollo de la misma. Este enfoque permitirá a los estudiantes no solo aprender los conceptos teóricos, sino también experimentar de primera mano la práctica de ejecutar jobs MapReduce en un entorno controlado y simplificado.

Capítulo 6: Implementación de la Aplicación

En capítulos previos, se discutieron las bases del paradigma MapReduce, su relevancia en el procesamiento de grandes volúmenes de datos, y las soluciones existentes. Sin embargo, muchas de estas soluciones son complejas y costosas, lo que las hace imprácticas para contextos educativos. Este capítulo se centra en la implementación de una aplicación que permite enseñar y comprender MapReduce de manera accesible y eficiente, ejecutándose directamente en el navegador.

El propósito de esta aplicación es proporcionar a los estudiantes una herramienta práctica para experimentar con MapReduce sin necesidad de una infraestructura sofisticada. Utilizando tecnologías como React y Pyodide, hemos creado un entorno de ejecución distribuido que facilita la comprensión de los conceptos fundamentales de MapReduce.

Este capítulo se estructura en las siguientes secciones:

- **Arquitectura de la Aplicación**: Descripción de los componentes, sus funciones y su interoperabilidad.
- **Flujo de Comunicación de la Aplicación**: Explicación del intercambio de datos entre el frontend y el backend, y la comunicación entre los nodos del sistema.
- **Explicación del Funcionamiento de la Aplicación**: Detalle de las funcionalidades específicas, incluyendo los roles de Master y Slaves, la gestión de tareas y estadísticas, mecanismos de manejo de errores y recuperación ante fallos, la subida de los archivos, la gestión de sesiones, y el circuito de mensajes diseñado.

Esta implementación no solo enseña los conceptos teóricos de MapReduce, sino que también proporciona una experiencia práctica, permitiendo a los estudiantes observar cómo se distribuyen y procesan las tareas en un entorno de clúster simulado. A continuación, se presenta un análisis detallado de cada componente y proceso de la aplicación.

6.1 Arquitectura de la Aplicación

En este apartado, se presenta una descripción exhaustiva de la arquitectura de la aplicación, que se estructura en torno a varios componentes clave: el Frontend (Next-App), el Backend, el servidor NGINX y el Cert-Generator. La Fig. 6.1.1 ilustra estos componentes y sus interacciones, proporcionando una representación visual que facilita la comprensión de su configuración y rol dentro de la arquitectura global.

A continuación, se ofrece una explicación detallada de cada componente y su función específica en el funcionamiento de la aplicación.

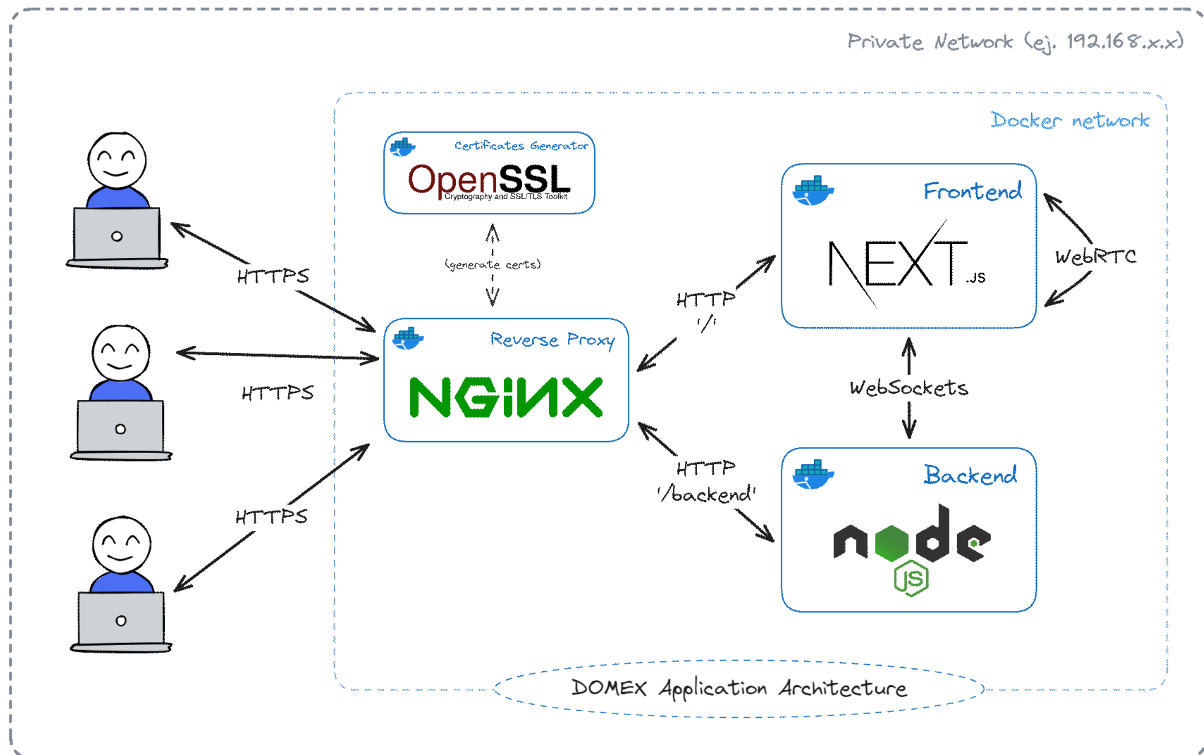


Fig. 6.1.1: Arquitectura de la aplicación DOMEX

6.1.1 Frontend (Next.js)

El **Frontend**, desarrollado con el framework Next.js, constituye el núcleo de la lógica y la interacción de la aplicación. Next.js fue seleccionado debido a su robustez y popularidad en la industria del desarrollo web, ofreciendo un conjunto de características que optimizan tanto el rendimiento como la capacidad de desarrollo. Este componente gestiona no solo la interfaz visual de la aplicación, sino que también se encarga de ejecutar los jobs MapReduce y coordinar la sincronización de mensajes entre los nodos a través de WebRTC.

Uno de los aspectos más destacados del Frontend es su capacidad para cargar componentes críticos como react-py. Este componente utiliza Pyodide, lo cual nos permite la ejecución de código Python dentro del entorno JavaScript, y así, llevar a cabo la ejecución de los jobs MapReduce directamente desde el navegador. La integración de Pyodide permite que el Frontend maneje tareas complejas de procesamiento de datos sin depender de servidores externos, mejorando la eficiencia y la escalabilidad del sistema.

Además, el Frontend desempeña un papel crucial en la gestión de la comunicación entre los nodos a través de WebRTC. Esta tecnología permite establecer conexiones

peer-to-peer para la transmisión de datos en tiempo real, esencial para la sincronización de tareas distribuidas y la ejecución de jobs MapReduce. La capacidad del Frontend para manejar estas conexiones de manera eficiente es fundamental para garantizar una experiencia de usuario fluida y una correcta ejecución de las tareas distribuidas.

6.1.2 Backend (Node.js y Express.js)

El **Backend**, construido con Node.js y el framework Express.js, cumple una función específica pero esencial en la arquitectura de la aplicación. Su principal responsabilidad es actuar como servidor de señalización para la configuración de conexiones entre los nodos del Frontend mediante WebRTC. Esta función es crucial para facilitar la comunicación entre los diversos componentes del Frontend, asegurando que los nodos puedan coordinarse y sincronizarse de manera efectiva para el procesamiento distribuido.

La implementación del Backend en Node.js aprovecha su capacidad para manejar múltiples conexiones simultáneas y su eficiencia en la transmisión de datos. Express.js, como framework ligero y flexible, simplifica el manejo de rutas y solicitudes, facilitando la comunicación entre el Frontend y el Backend. La comunicación entre estos dos componentes se realiza exclusivamente a través de websockets, lo que proporciona una transmisión de datos rápida y bidireccional, optimizando el rendimiento de la aplicación.

6.1.3 Servidor NGINX

El **servidor NGINX** se integra en la arquitectura como un reverse proxy, desempeñando un papel crucial en la gestión del tráfico HTTPS. Su función principal es manejar las peticiones entrantes, tanto para el Frontend como para el Backend, y ocultar estos componentes detrás de una única interfaz pública. Esta configuración no solo mejora la seguridad al cifrar las comunicaciones, sino que también optimiza la entrega de contenido y equilibra la carga de trabajo entre los diferentes componentes.

NGINX actúa como intermediario en el tráfico entre los clientes y los servidores internos, manejando todo lo referido al cifrado y descifrado de los mensajes, y facilitando la gestión eficiente de las conexiones, reduciendo la carga directa sobre los servicios de Backend y Frontend. Además, NGINX puede realizar funciones de caching y compresión de datos, lo que contribuye a una mejora en la velocidad de respuesta y en la eficiencia general de la aplicación. Para más información de la configuración específica del servidor NGINX puede dirigirse al **Anexo NGINX: Configuración del Servidor NGINX.**

6.1.4 Cert-Generator

El **Cert-Generator** es un componente que se incorporó al final del desarrollo para abordar la necesidad de operar en un entorno HTTPS. Esta necesidad surgió debido a que ciertos componentes del Frontend, como los service workers, requieren HTTPS para funcionar correctamente. El Cert-Generator se encarga de generar los certificados SSL necesarios para habilitar la comunicación segura entre los componentes de la aplicación.

Este componente ejecuta un script al inicio de la aplicación que genera los certificados y los almacena en un volumen compartido. Estos certificados son luego utilizados por el servidor NGINX para cifrar las comunicaciones. La integración del Cert-Generator asegura que todas las interacciones entre el Frontend, el Backend y los clientes estén protegidas mediante cifrado, cumpliendo con los estándares de seguridad y garantizando la integridad de los datos transmitidos.

6.1.5 Docker y Redes

La utilización de **Docker** como plataforma de orquestación es fundamental para la integración y gestión de los componentes de la aplicación. Docker permite encapsular cada componente en contenedores independientes, que se comunican a través de redes definidas dentro del entorno Docker. Esta configuración facilita la interconexión de los diferentes servicios y asegura que la aplicación pueda desplegarse y ejecutarse eficientemente en redes locales, ya sea a través de conexiones Wi-Fi o cableadas.

Docker proporciona una capa adicional de flexibilidad y escalabilidad, permitiendo que la aplicación se adapte fácilmente a diferentes entornos operativos y configuraciones de red. La capacidad de Docker para gestionar redes y volúmenes simplifica la administración de la aplicación y asegura que todos los componentes funcionen en armonía, sin interferencias entre ellos. Para entrar más en detalle de la implementación dockerizada de la aplicación, puede dirigirse al **Anexo 2: Dockerización de la aplicación**, donde se explicarán más en detalle cada uno de los contenedores creados.

En resumen, la arquitectura de la aplicación está meticulosamente diseñada para proporcionar seguridad, eficiencia, adaptabilidad y escalabilidad. Cada componente, desde el Frontend y el Backend hasta NGINX y el Cert-Generator, desempeña un papel específico y crítico en el funcionamiento global del sistema. La combinación de estos elementos, junto con la flexibilidad de Docker, garantiza una implementación robusta y adaptable a diversas necesidades operativas, asegurando un rendimiento óptimo y una experiencia de usuario simple y satisfactoria.

6.2 Flujo de Comunicación de la Aplicación

En este apartado, se detalla el flujo de comunicación entre los principales componentes de la aplicación, enfatizando la interacción entre el Frontend y el Backend, así como la comunicación entre los nodos utilizando WebRTC. Este capítulo examina cómo se implementan y coordinan estos protocolos para lograr una comunicación fluida, eficiente y segura, vital para la ejecución de tareas en tiempo real y la sincronización de los nodos en un entorno distribuido.

Veremos como la comunicación entre el Frontend y el Backend se realiza a través de WebSockets, un protocolo que permite conexiones bidireccionales persistentes, asegurando un intercambio continuo de mensajes y eventos críticos para la operación de la aplicación. Además, se explora la comunicación entre los nodos mediante WebRTC, un protocolo peer-to-peer que facilita la transmisión directa de datos entre navegadores, empleando una topología de red full-mesh para garantizar la sincronización y distribución de tareas.

Este capítulo, por lo tanto, ofrece una visión completa del diseño y funcionamiento de los mecanismos de comunicación en la aplicación, mostrando cómo estos componentes se integran para ofrecer una experiencia de usuario coherente y robusta. A continuación, se profundizará en la aplicación de cada protocolo utilizado sobre el sistema.

6.2.1 Frontend y Backend

La comunicación entre el Frontend y el Backend, como hemos mencionado anteriormente en este informe, se realiza a través del protocolo de *WebSockets*. Esta decisión se tomó debido a la necesidad de un protocolo que permitiera una comunicación bidireccional eficiente, fundamental para las operaciones en tiempo real que realiza la aplicación.

Los WebSockets son ideales para aplicaciones que requieren una comunicación constante y de baja latencia entre el cliente y el servidor. A diferencia del protocolo HTTP tradicional, que sigue un modelo de petición-respuesta, los WebSockets permiten una conexión persistente, donde tanto el cliente como el servidor pueden enviar mensajes de manera continua.

<u>Descripción del Proceso</u>

La Fig. 6.2.1.1 ilustra el mecanismo de comunicación entre el backend (servidor de señalización) y el frontend (navegadores). En este esquema, se observa cómo los navegadores (representados por los iconos de Chrome y Firefox, Nodo 1 y Nodo 2 respectivamente) se comunican con el servidor a través de WebSockets para

establecer y mantener las conexiones necesarias para la sincronización y la coordinación entre los diferentes nodos.

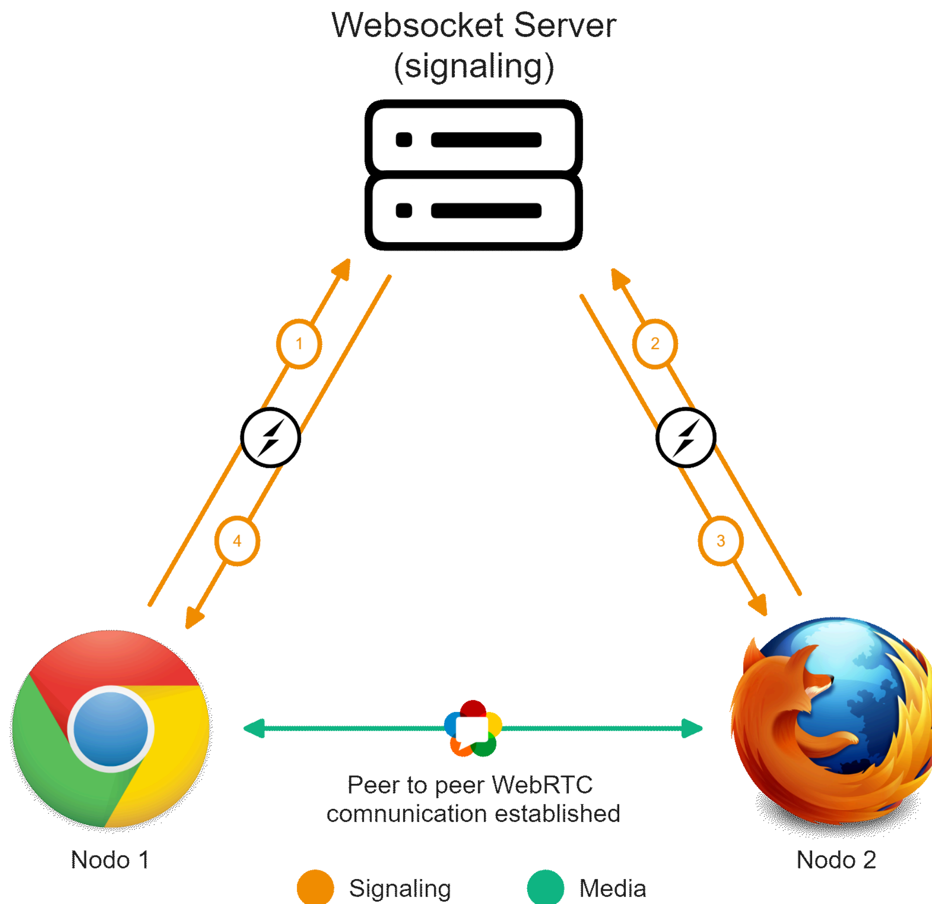


Fig. 6.2.1.1: Mecanismo de establecimiento de comunicación WebRTC y WebSocket

En este intercambio de mensajes podemos mencionar las siguientes funciones principales:

Conexión Inicial: Cuando un usuario abre la aplicación en su navegador, e ingresa a un clúster, se establece una conexión WebSocket con el servidor. Esta conexión permite el intercambio continuo de mensajes entre el cliente y el servidor.

Intercambio de Mensajes: A través de esta conexión, se envían y reciben diversos eventos que facilitan la sincronización de estado, la gestión de los clusters y la comunicación en tiempo real.

Señalización para WebRTC: La conexión WebSocket también se utiliza para la señalización de WebRTC, necesaria para establecer las conexiones peer-to-peer (P2P) entre los navegadores. Los nodos intercambian mensajes de señalización para negociar la conexión WebRTC. Esto incluye el

intercambio de ofertas (offer), respuestas (answer) y candidatos de ICE (Interactive Connectivity Establishment).

En el contexto de nuestra aplicación, el Backend, implementado en Node.js y Express.js, como mencionamos anteriormente, maneja varios eventos relacionados con la comunicación mediante Websockets. Estos eventos se pueden agrupar en dos categorías principales: la gestión de los clústers y el establecimiento de la comunicación entre los peers usando WebRTC.

Gestión de los Clústers

El archivo *registerCluster.ts*, perteneciente al código fuente de la aplicación, define cómo el Backend maneja la gestión de los clústers. A continuación, se detallan los principales eventos:

- *Persistencia de la sesión:* Cuando un nodo se conecta, su sesión se guarda en el almacén de sesiones en memoria (*clustersSessionStore*).
- *Emisión de detalles de la sesión*: Se emiten los detalles de la sesión al nodo, incluyendo el ID de la sesión, el ID del nodo y si es el propietario del clúster (nodo *master*).
- *Unirse al clúster:* El nodo se une al clúster correspondiente.
- *Gestión de nodos en el clúster*: Se emite una lista de nodos en el clúster al nuevo nodo y se notifica a los demás nodos sobre la nueva conexión.
- *Eventos de salida y expulsión:* Se manejan eventos cuando un nodo abandona el clúster o es expulsado, actualizando el estado en el almacén de sesiones y notificando a los demás nodos.
- *Bloqueo y desbloqueo de clústers:* Permite al nodo master bloquear o desbloquear el clúster, gestionando el acceso de nuevos nodos.

Establecimiento de la Comunicación entre Peers con WebRTC

En el archivo *registerWebRTC.ts*, también perteneciente al código fuente de la aplicación, se definen los eventos relacionados con el establecimiento de la comunicación entre los peers mediante WebRTC. A continuación, se explican los eventos clave:

- *Envío de señal:* Cuando un nodo quiere establecer una conexión WebRTC, envía una señal a otro nodo. Esta señal incluye la información necesaria para iniciar la conexión.
- *Retorno de señal:* El nodo que recibe la señal responde con una señal de retorno, completando el proceso de establecimiento de la conexión.

Estos eventos son fundamentales para establecer y mantener las conexiones WebRTC entre los nodos, permitiendo la comunicación en tiempo real necesaria para la sincronización de tareas distribuidas.

6.2.2 Comunicación entre peers (frontend)

La comunicación entre peers en la aplicación se realiza mediante WebRTC, un protocolo que permite la comunicación en tiempo real directamente entre navegadores. Este protocolo es fundamental para la transmisión de datos referidos a la ejecución de los jobs entre los diferentes nodos (browsers).

Descripción del Proceso

Tal como observamos en la Fig. 6.2.2.1, la comunicación se establece siguiendo una topología full-mesh, donde cada nodo establece una conexión con cada uno de los demás nodos. Este enfoque, aunque no escala bien con un gran número de pares debido al número exponencial de conexiones necesarias, se eligió por su simplicidad y por el hecho de que la aplicación está diseñada para operar con un número moderado de nodos.

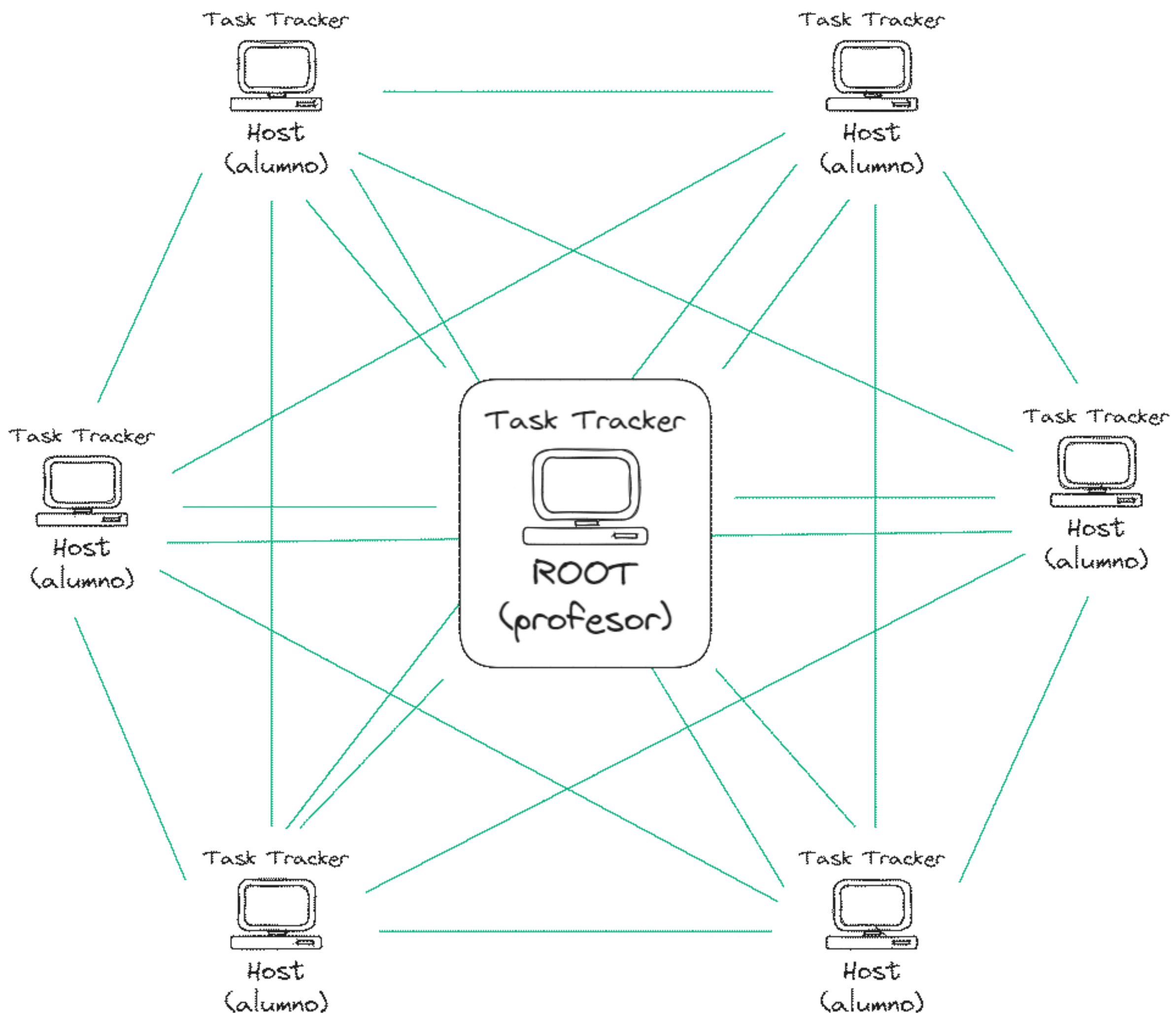


Fig. 6.2.2.1: Topología full-mesh de conexiones entre los nodos

En una topología full-mesh, el número total de conexiones (edges) en la red es $\frac{n(n-1)}{2}$, donde n es el número de pares. Esta configuración asegura que cada nodo pueda comunicarse directamente con todos los demás nodos, lo que facilita la sincronización y la distribución de tareas en el contexto del paradigma MapReduce.

Mecanismo de Comunicación

El proceso de comunicación con WebRTC implica varios pasos:

1. *Señalización*: Antes de establecer una conexión directa, los peers necesitan intercambiar información de configuración a través de un servidor de señalización. En nuestra aplicación, este rol lo cumple el Backend mediante WebSockets.

2. *Intercambio de ofertas y respuestas:* Un peer crea una oferta de conexión que se envía al otro peer. El segundo peer responde con una respuesta que acepta la conexión.

3. *Intercambio de candidatos ICE:* Ambos peers intercambian información sobre sus direcciones IP y puertos mediante los candidatos ICE (Interactive Connectivity Establishment).

4. *Establecimiento de la conexión:* Una vez que ambos peers han intercambiado toda la información necesaria, se establece una conexión directa entre ellos.

5. *Transmisión de datos:* Con la conexión establecida, los peers pueden intercambiar datos en tiempo real, ya sea audio, video o datos arbitrarios.

Este flujo de comunicación asegura que los usuarios de la aplicación puedan interactuar de manera eficiente y segura, aprovechando las ventajas que ofrecen los websockets y WebRTC.

En resumen, el flujo de comunicación de la aplicación se basa en el uso de WebSockets para la interacción entre el Frontend y el Backend, y WebRTC para la comunicación entre los peers. Esta combinación de tecnologías proporciona una plataforma robusta y eficiente para la ejecución de tareas distribuidas y la sincronización en tiempo real, garantizando una experiencia de usuario óptima y una comunicación segura.

6.3 Explicación del Funcionamiento de la Aplicación

Al ingresar al sistema, la primera página que se visualiza es la de autenticación, donde el usuario debe ingresar un nombre de manera obligatoria para poder identificarlo del resto de nodos. Desde esta página, se ofrece la opción de crear un nuevo clúster o unirse a uno existente, lo cuál determina el rol que asumirá el nodo dentro de la aplicación.

Ingresar

Usuario*

Indentificador de cluster

UNIRSE

CREAR UN CLUSTER

Fig. 6.3.1: Página de inicio de la aplicación

6.3.1 Dos Tipos de Nodos: Master y Slaves

- **Master (Job Tracker):** Si el usuario decide crear un nuevo clúster, puede optar por especificar un identificador para el mismo. En caso de no hacerlo, el sistema generará un identificador automáticamente. Al crear el clúster, el usuario asume el rol de Master, también conocido como Job Tracker, responsable de la coordinación general del clúster y la distribución de tareas.

- **Slave (Task Tracker):** Si el usuario opta por unirse a un clúster existente, debe ingresar el identificador del mismo. El servidor verificará que el nombre del usuario no se repita con el de otro nodo previamente conectado al cluster, que el cluster exista y que no se encuentre bloqueado, es decir, que no haya ningún job MapReduce en ejecución. En este caso, el usuario asume el rol de Slave, también conocido como Task Tracker, encargado de ejecutar las tareas asignadas por el Master.

En ambos casos, el identificador del nodo es obligatorio. Si se intenta proceder sin especificar un nombre, sin completar el identificador del clúster al unirse a uno existente, o especificando un identificador que no sea único al momento de crear uno nuevo, el sistema mostrará advertencias adecuadas para solicitar la información faltante o corregir la provista.

Posteriormente, el usuario será redirigido a la interfaz asociada al clúster. En la parte superior de esta página, para ambos roles, se muestra una barra de estado donde se indica el identificador del cluster, y un menú desplegable que muestra el nombre del nodo y una opción para abandonar el clúster.

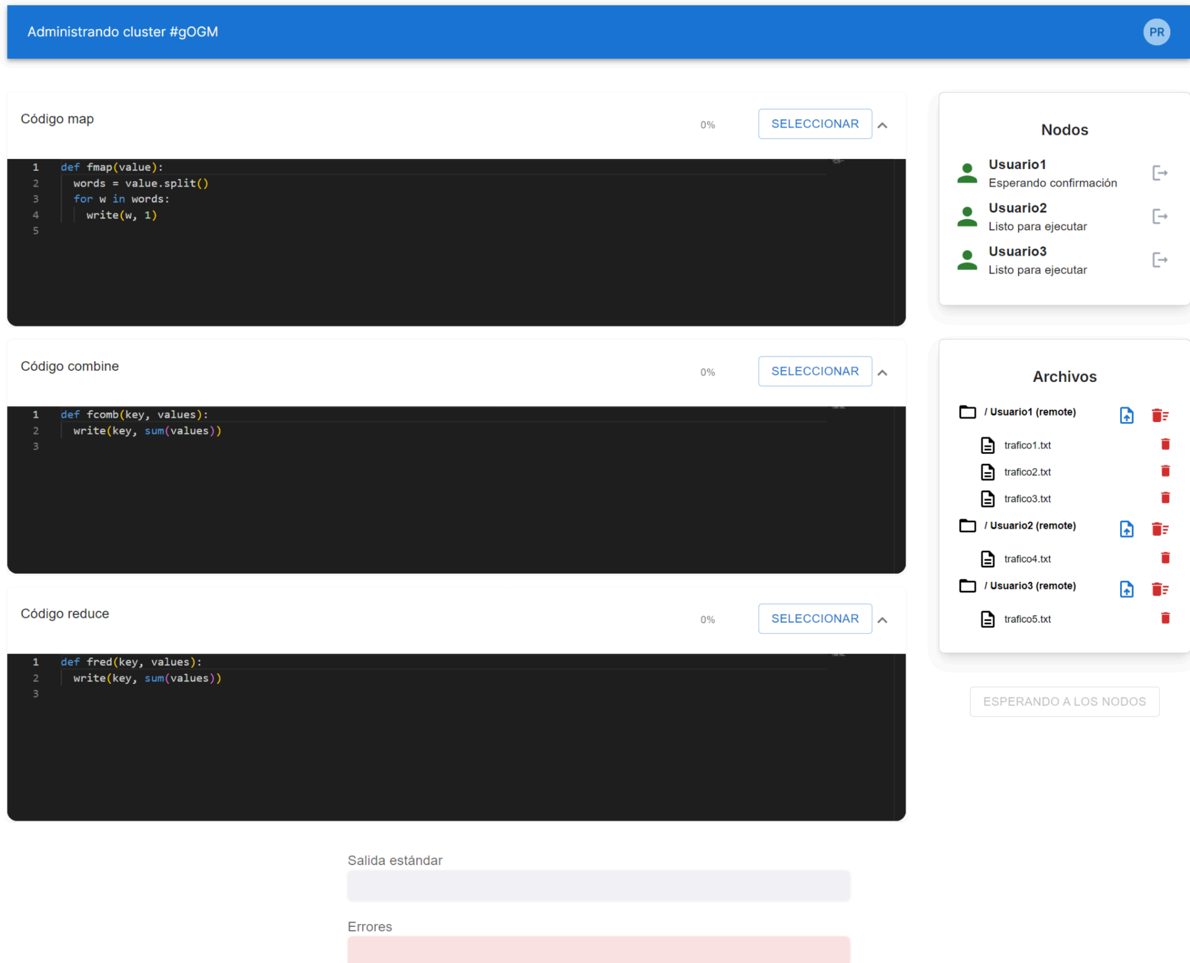


Fig. 6.3.1.1: Interfaz visible para el Master del clúster

6.3.2 Funciones del Master

6.3.2.1 Definición de funciones

El Master es responsable de definir las funciones *map*, *combine* y *reduce*. Para lograr esto, puede importar los archivos *.py* que contengan las definiciones correspondientes, o utilizar los editores de código que se encuentran en la parte izquierda de la interfaz. Estas funciones deben cumplir con ciertos requisitos sintácticos:

- **Función map:** Debe llamarse *fmap* y recibir un parámetro, el cual representa una línea de los archivos de entrada.
- **Función combine:** Debe llamarse *fcomb* y recibir dos parámetros, donde el primero representa una clave (generada en la etapa map) y el segundo es una lista que contiene los valores asociados a dicha clave. Esta función es

opcional, por lo tanto, el Master puede optar por no especificar código para la misma, evitando así su ejecución.

- **Función reduce:** Debe llamarse *fred* y recibir dos parámetros, donde el primero representa una clave (generada en la etapa map o por la función combine) y el segundo es una lista que contiene todos los valores asociados a dicha clave.

Al dar inicio al procesamiento, se realiza un validación sintáctica sobre los tres códigos ingresados, y se verifica que las funciones se declaren de la forma esperada. En caso de que alguna de estas validaciones falle, se muestra el error en el editor correspondiente.

6.3.2.2 Monitoreo de Estado y Gestión de Conexiones

El Master monitorea continuamente el estado de los nodos Slaves, utilizando un sistema de colores para indicar el estado de la conexión:

- **Rojo:** La conexión por socket aún no se ha establecido, o ha ocurrido un error durante la ejecución del nodo.
- **Amarillo:** La conexión peer-to-peer aún no se ha establecido.
- **Verde:** Tanto la conexión por socket como la conexión peer-to-peer se han establecido correctamente.

Además, el Master puede monitorear los estados de ejecución de los nodos Slaves, los cuales pueden observarse en la Fig. 6.3.1.1 en el listado de nodos, debajo del nombre de cada uno. Los posibles estados son:

- **Esperando confirmación:** El nodo no ha confirmado sus datos de entrada o no está listo para ejecutar.
- **Listo para ejecutar:** El nodo ha confirmado sus datos y está listo para ejecutar. Esto ocurre cuando el nodo presiona el botón correspondiente, el cual se habilita una vez que el entorno de Pyodide se haya iniciado correctamente.
- **Ejecutando map:** El nodo se encuentra ejecutando la etapa map. Esto ocurre siempre y cuando el nodo sea un nodo mapper, es decir, que disponga de al menos un archivo de entrada.
- **Ejecutando combine:** El nodo se encuentra ejecutando la función combine. Esto ocurre solo cuando el master ha especificado código para esta función, y además, es un nodo mapper.

- **Esperando claves:** El nodo se encuentra esperando a que el resto de nodos le envíen las claves (junto con sus valores asociados) que el Master le asignó para procesar durante la etapa reduce.

- **Ejecutando reduce:** El nodo se encuentra ejecutando la etapa reduce. Esto ocurre solo cuando al nodo se le ha asignado al menos una clave para reducir.

- **Ejecución finalizada:** El nodo ha completado su procesamiento y ha enviado los resultados al Master.

- **Error en map / combine / reduce:** Indica que hubo un fallo en la ejecución de una de las etapas.

Por otro lado, en la parte superior derecha correspondiente a cada sección donde se definen las funciones, se podrá visualizar un porcentaje de progreso, que irá incrementando a medida que los nodos completen cada etapa. En el caso de los Slaves, se muestra un spinner de carga al momento de ejecutar cada etapa, y una vez finalizada, muestra un tick verde en caso de que se haya completado correctamente.

Así mismo, en la parte inferior de la interfaz, se muestran dos secciones. La primera, en el caso de los Slaves, informa a medida que se van completando las etapas, y muestra la salida estándar (por ejemplo, los prints que se realizan dentro del código), lo cual resulta útil para debuguear y resolver problemas. En el caso del Master, simplemente se informan las etapas que va completando cada nodo; no se muestra la salida estándar de cada uno de ellos, para evitar un tráfico de datos excesivo a través de la red. Luego, la segunda sección muestra los errores en ejecución, indicando el nodo correspondiente en el caso del Master.

A su vez, el nodo master tiene la posibilidad de expulsar a los nodos del clúster. Para ello, debe presionar el botón correspondiente, el cual se encuentra posicionado sobre la parte derecha del listado de nodos. Los nodos recibirán una notificación sobre esta acción y serán redirigidos a la página inicial de la aplicación.

6.3.2.3 Asignación y Distribución de Tareas

Una vez que todos los nodos se encuentren listos para ejecutar, el Master tendrá habilitado un botón para dar inicio al procesamiento. Al presionarlo envía los códigos a cada uno de los Slaves y da la señal para que inicien la ejecución. En ese momento, el clúster se bloquea, evitando que se conecten nuevos nodos.

En caso de ser necesario, una vez iniciada la ejecución, el Master tiene la posibilidad de detener el job con un botón dedicado. Al presionarlo, el job vuelve a su estado inicial, es decir, todos los nodos pasan al estado “Esperando confirmación”, y se desbloquea el clúster.

A medida que los nodos slave finalizan la ejecución de la etapa map y la función combine, reportan las claves generadas al Master, indicando a su vez la cantidad de veces que se generó cada clave. Cuando todos los slaves hayan informado sus resultados al Master, comenzará un proceso de redistribución, donde a cada nodo se le asignará un subconjunto de claves que procesará durante la etapa reduce.

Para lograr esto, se combinan todos los pares clave - cantidad informados por los nodos, de esta forma, el Master sabrá exactamente cuáles son las claves únicas generadas entre todos los nodos, y cuantas veces se generó cada clave.

Con esta información, el Master asignará una determinada cantidad de claves a cada nodo. Esta cantidad está dada por el resultado de la operación *Math.ceil(keys.length / clusterNodes.length)*, es decir, la división de la cantidad total de claves únicas entre la cantidad total de nodos.

Luego, el Master recorre los pares clave - cantidad, asignando subconjuntos a cada uno de los nodos, hasta que ya no queden claves por repartir. En este procedimiento de redistribución puede que no a todos los nodos se les haya asignado la misma cantidad de claves. Esto dependerá de la cantidad de claves únicas que se hayan generado.

Por ejemplo, si había 5 Slaves y se generaron 10 claves únicas, a cada Slave se le asignarán 2 claves. En cambio, si había 10 Slaves y se generaron 5 claves únicas, la cantidad de claves a recibir por cada nodo es *Math.ceil(keys.length / clusterNodes.length) → Math.ceil(5 / 10) → Math.ceil(0.5) = 1*, por lo tanto, solo 5 Slaves recibirán una única clave, y los otros 5 no recibirán ninguna (es decir, no ejecutarán la etapa reduce).

A su vez, cada nodo no solo necesita saber cuales son las claves que va a procesar, sino también desde cuantos nodos debe recibir las claves y valores que le falten, y hacia qué nodos enviar las claves que el mismo posea pero que no le corresponda procesar.

Para lograr esto, por cada nodo, se recorre el listado de claves que generó, y si una de las claves no le corresponde para la etapa reduce, se busca cuál es el nodo que la tiene asignada y se indica que dicha clave (junto con todos los valores) tiene que ser transferida a tal nodo. Posteriormente se utiliza esta información para calcular desde cuántos nodos cada uno debe recibir sus correspondientes claves.

Finalmente, el master le informa a cada nodo las claves que le asignó, desde cuantos nodos debe esperar recibir claves, y hacia qué nodos transferir claves. Utilizando esta información, los nodos slave empiezan a repartirse las claves entre ellos, dando inicio a la etapa reduce.

6.3.2.4 Manejo de Resultados y Finalización de Tareas

Una vez que los Slaves completan sus tareas, el cluster se desbloquea. En ese momento, el Master recopila y consolida los resultados, determinando si el job ha finalizado con éxito.

Los resultados finales de la ejecución se presentan en la parte inferior de la interfaz, en formato de tabla. La primera columna muestra la clave, mientras que la segunda muestra el valor final asociado. Esta tabla es ordenable tanto por clave como por valor, en orden ascendente o descendente, simplemente presionando el título de la columna correspondiente. En el caso de los Slaves, se mostrarán los resultados correspondientes a cada etapa.

Para evitar una sobrecarga de información en la pantalla, los resultados se presentan paginados, permitiendo seleccionar la cantidad de filas por página y la página a visualizar. Además, un botón ubicado en la esquina inferior izquierda permite descargar el contenido de la tabla en formato CSV.

Resultados

CLAVE	VALOR
Plaza	92
Otro	1228
Supermercado	316
Museo	197
Ferreteria	93

Filas por página: 5 1–5 de 8

Fig. 6.3.2.4.1: Visualización de los resultados de ejecución

A su vez, el sistema proporcionará un conjunto de estadísticas relacionadas con la ejecución del job. Los detalles sobre estas estadísticas, así como su cálculo, se explicarán en una sección posterior.

En caso de que el Master decida abandonar el clúster, todos los nodos serán notificados y expulsados del mismo, para posteriormente proceder con el cierre del clúster.

6.3.3 Funciones del Slave

Los Slaves, bajo la dirección del Master, realizan las siguientes funciones:

6.3.3.1 Ejecución de Tareas Asignadas

Los Slaves procesan los datos de entrada utilizando las funciones *map*, *combine*, y *reduce* según las instrucciones del Master. Durante la ejecución, reportan continuamente su estado al Master.

6.3.3.2 Monitoreo y Reporte de Estado

Los Slaves informan su estado de conexión al Master, siguiendo el sistema de colores para la conexión y los estados de ejecución previamente mencionados. Este reporte continuo permite al Master tener visibilidad sobre el progreso y la situación de cada nodo en todo momento.

Cuando un nodo abandona el cluster, el resto de nodos conectados son informados. Si se estaba ejecutando un job, la ejecución es cancelada, y los nodos pasan al estado inicial.

6.3.3.3 Manejo de Errores

Si ocurre un error durante la ejecución, los Slaves notificarán al Master el error en cuestión y la etapa en la cual ocurrió. Es decir, el Master podrá visualizar cuales nodos fallaron, cuales fueron los errores específicos, y sobre qué etapas.

Cuando alguno de los nodos falla durante el procesamiento, el job vuelve a su estado inicial, es decir, todos los nodos pasan al estado “Esperando confirmación”, y se desbloquea el cluster.

Luego, el Master deberá decidir cómo proceder, ya sea reasignando tareas o archivos de entrada, para posteriormente reintentar la ejecución del job.

6.3.3.4 Visualización de estadísticas

Una vez que el Slave finaliza correctamente su ejecución, en la parte inferior de la página podrá visualizar un conjunto de estadísticas asociadas al mismo. En una sección posterior se brindarán detalles sobre qué estadísticas se visualizan y cómo se calculan.

6.3.4 Subida de Archivos

La gestión de archivos es un componente fundamental en la operación de la aplicación, diseñado con la intención de emular el manejo como si se tratara con grandes volúmenes de datos, típico en entornos de Big Data, aunque la aplicación esté limitada a archivos de menor tamaño. Desde el inicio, se estableció como objetivo principal evitar la transferencia de los datos de los archivos a través de la red, una práctica común en sistemas de procesamiento masivo de datos, donde el

tamaño y el volumen de los archivos hacen que su transferencia sea impráctica o imposible.

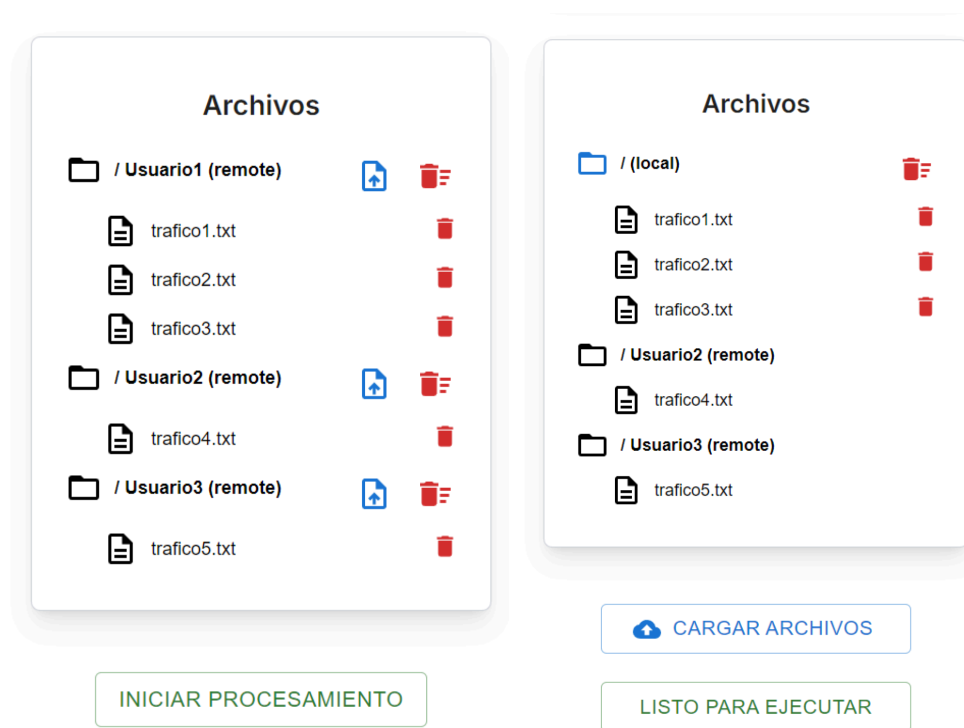


Fig. 6.3.4.1: Vista de subida de archivos del Master y Slave

Para cumplir con este objetivo, cada nodo Slave tiene la responsabilidad de cargar los archivos de trabajo de manera local, utilizando un botón específico en la interfaz de usuario, como se observa en la imagen de la derecha de la Fig. 6.3.4.1. Esta funcionalidad permite a cada nodo Slave subir los archivos directamente desde su dispositivo, sin que su contenido sea transmitido a través de la red. Una vez que los archivos han sido subidos localmente, el nodo Slave envía a través de la red únicamente los nombres de los archivos cargados, junto con una referencia al nodo que los ha cargado. Este enfoque asegura que cada nodo tenga una lista completa de todos los archivos disponibles, así como el conocimiento de qué nodo es responsable de cada archivo, sin necesidad de transferir los datos en sí. A su vez, cada nodo tiene la posibilidad de borrar todos los archivos seleccionados, o alguno en particular, utilizando los botones correspondientes. Una vez que el job comienza su ejecución, el Slave no podrá agregar ni borrar archivos.

Adicionalmente, y con fines prácticos, para facilitar la tarea en escenarios de enseñanza o demostraciones en clase, el sistema le permite opcionalmente al Master asignar archivos de entrada a cada nodo Slave en particular, tal como se observa en la imagen izquierda de la Fig. 6.3.4.1. En este caso, los archivos sí son transferidos a través de la red, hasta llegar al nodo Slave correspondiente. Esta opción se ha incorporado para mejorar la usabilidad de la aplicación en contextos

específicos, facilitando la preparación y ejecución de tareas sin necesidad de que cada nodo Slave cargue manualmente los archivos.

Este diseño no solo respeta los principios de eficiencia y escalabilidad que subyacen al job, sino que también ofrece flexibilidad para adaptarse a diferentes escenarios de uso, asegurando una experiencia de usuario óptima y una gestión de archivos efectiva dentro de la aplicación.

6.3.5 Circuito de Mensajes

El circuito de mensajes en la aplicación es esencial para la coordinación entre el nodo Master y los nodos Slaves, permitiendo una comunicación fluida y sincronizada en todas las etapas del proceso de ejecución de un job MapReduce. A continuación, se describe en detalle cómo se lleva a cabo este circuito de mensajes.

6.3.5.1 Inicio de Conexión y Señalización

El proceso de comunicación comienza cuando los nodos establecen una conexión inicial mediante WebSockets. Esta conexión permite el intercambio de información básica y la sincronización inicial entre los nodos, lo cual es fundamental para asegurar que todos los nodos estén en la misma página antes de iniciar cualquier operación de procesamiento. Posteriormente, se establece una conexión WebRTC, que se convierte en la línea troncal para toda la intercomunicación dentro de la aplicación. Esta conexión asegura que la transmisión de datos y mensajes entre los nodos sea rápida, segura y eficiente, minimizando las latencias y maximizando la fiabilidad del sistema.

6.3.5.2 Distribución de Tareas y Sincronización

Una vez establecidas las conexiones, el nodo Master tiene la responsabilidad de enviar las instrucciones necesarias para que los nodos Slaves ejecuten sus tareas asignadas. Estas instrucciones consisten en el código Python que define las funciones Map, Combine (cuando aplica) y Reduce, las cuales serán utilizadas por los Slaves durante el procesamiento. El Master también controla las órdenes de inicio o detención del procesamiento, asegurando que todas las etapas se ejecuten de manera coordinada.

Cabe destacar que, aunque cada nodo Slave es responsable de cargar localmente los archivos con los que va a trabajar, el Master tiene la capacidad, de manera opcional, de cargar y distribuir archivos de entrada a los Slaves a través de la red. Sin embargo, esta función es opcional y no reemplaza la responsabilidad de los Slaves de gestionar sus propios datos de entrada. La precisión en la distribución de las instrucciones y la correcta sincronización son cruciales para evitar errores que puedan comprometer la ejecución del trabajo.

6.3.5.3 Ejecución y Reporte de Resultados

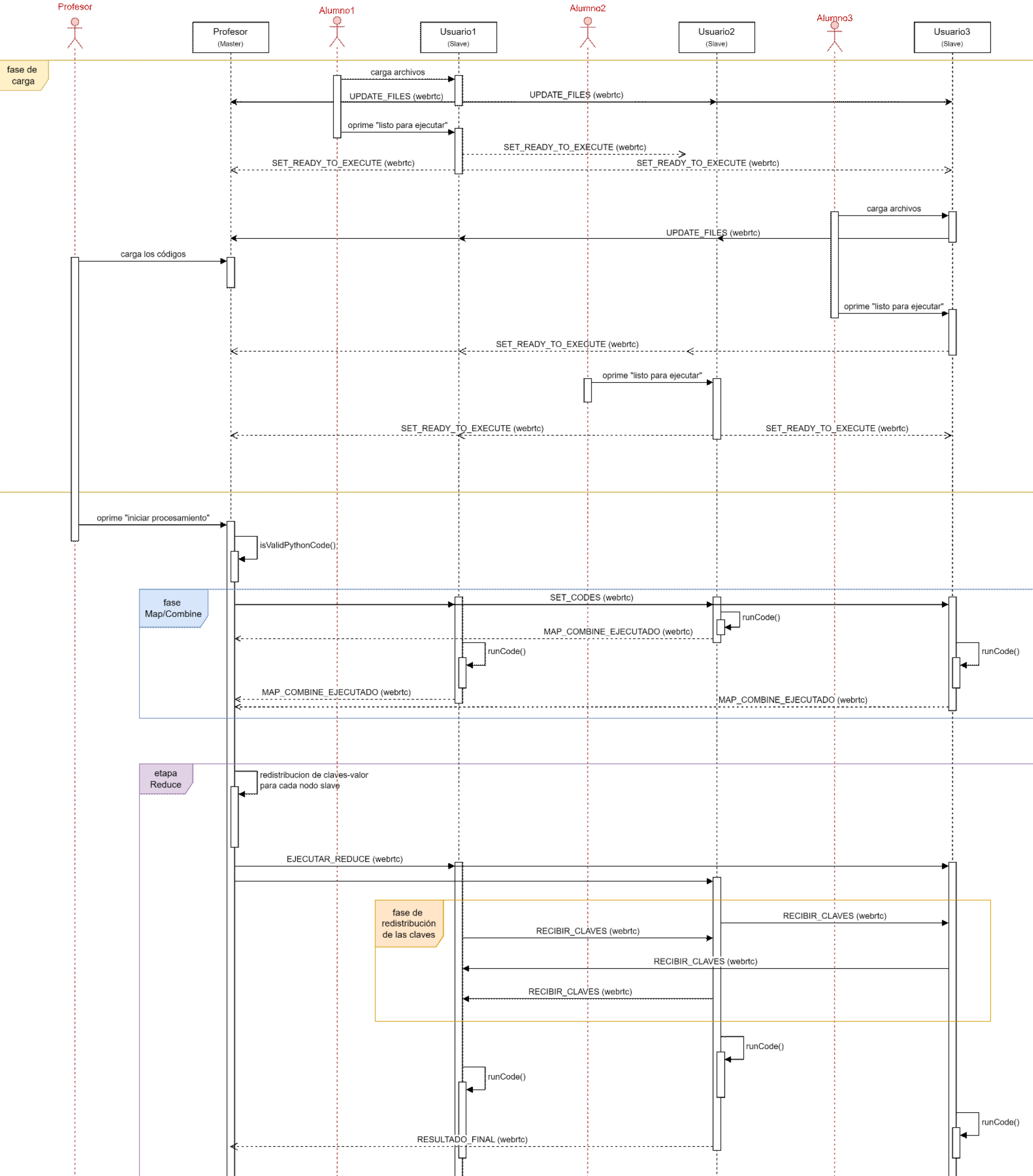
Durante la ejecución de las tareas asignadas, los nodos Slaves mantienen una comunicación continua con el nodo Master. A través de mensajes de estado, los Slaves informan al Master sobre su progreso en tiempo real, lo que permite al Master monitorear y controlar la ejecución de cada tarea. Esta retroalimentación es esencial para asegurar que todas las tareas se estén ejecutando correctamente y dentro de los parámetros esperados. Además, cuando finaliza la etapa Map, y se completó posteriormente el mecanismo de optimización Combine (en caso de que se haya especificado), los nodos Slaves envían los resultados intermedios al Master, lo cual facilita la redistribución de las claves y la preparación para la siguiente fase.

6.3.5.4 Finalización de Tareas y Desconexión

Al concluir la ejecución de las tareas asignadas, cada nodo Slave notifica al Master enviando no solo los resultados finales de su procesamiento, sino también todas las estadísticas recopiladas durante la ejecución. Estas estadísticas son cruciales para el análisis posterior y la optimización de futuras ejecuciones. Con toda la información en su poder, el Master puede optar por iniciar una nueva ejecución, ajustando parámetros como el número de nodos participantes, los archivos de entrada y el código a ejecutar. Alternativamente, el Master puede decidir dar por finalizada la sesión del clúster, procediendo a cerrar el circuito de mensajes y desconectar todos los nodos.

6.3.5.5 Sincronización para la Ejecución del Job MapReduce

El circuito de mensajes no solo facilita la comunicación básica entre los nodos, sino que también es fundamental para la sincronización durante la ejecución de un job MapReduce. Como se ilustra en la Fig. 6.3.5.5.1, la secuencia de mensajes comienza con la carga de los datos de entrada, tanto por parte de los nodos Slaves como del Master, quien es responsable de cargar el código Python que se utilizará en las diferentes etapas del procesamiento. Durante esta fase de carga, cada nodo es notificado sobre qué archivos subió cada uno, permitiendo que todos tengan una vista completa de los datos que se van a procesar.


Profesor
Alumno1
Alumno2
Alumno3
Profesor
(Master)
Usuario1
(Slave)
Usuario2
(Slave)
Usuario3
(Slave)
fase de carga
carga archivos
UPDATE_FILES (webrtc)
UPDATE_FILES (webrtc)
oprime "listo para ejecutar"
SET_READY_TO_EXECUTE (webrtc)
SET_READY_TO_EXECUTE (webrtc)
SET_READY_TO_EXECUTE (webrtc)
carga archivos
UPDATE_FILES (webrtc)
carga los códigos
oprime "listo para ejecutar"
SET_READY_TO_EXECUTE (webrtc)
oprime "listo para ejecutar"
SET_READY_TO_EXECUTE (webrtc)
SET_READY_TO_EXECUTE (webrtc)
oprime "iniciar procesamiento"
isValidPythonCode()
fase Map/Combine
SET_CODES (webrtc)
runCode()
MAP_COMBINE_EJECUTADO (webrtc)
runCode()
runCode()
MAP_COMBINE_EJECUTADO (webrtc)
MAP_COMBINE_EJECUTADO (webrtc)
etapa Reduce
redistribucion de claves-valor
para cada nodo slave
EJECUTAR_REDUCE (webrtc)
fase de redistribución de las claves
RECIBIR_CLAVES (webrtc)
RECIBIR_CLAVES (webrtc)
RECIBIR_CLAVES (webrtc)
RECIBIR_CLAVES (webrtc)
runCode()
runCode()
runCode()
RESULTADO_FINAL (webrtc)

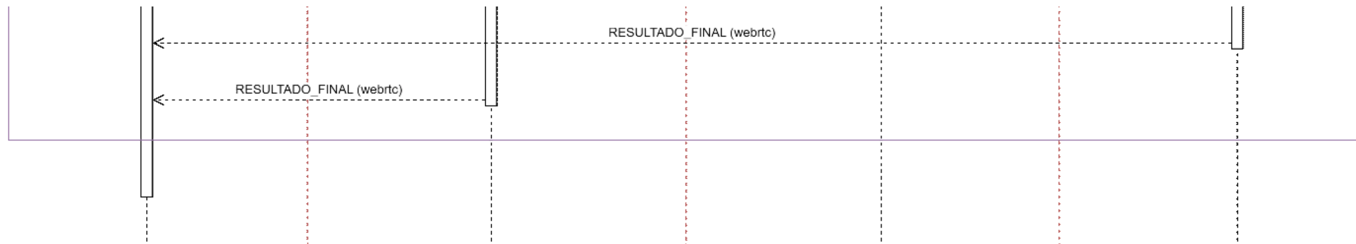


Fig. 6.3.5.5.1: Diagrama de secuencia de la ejecución de un job Mapreduce

Una vez completada la carga de datos, el Master da la señal para iniciar la ejecución del job MapReduce, comenzando con la fase Map. Al finalizar dicha etapa, en caso de haber sido especificado el mecanismo de optimización Combine, el mismo es ejecutado a continuación. Los nodos Slaves, al finalizar su parte de la ejecución, informan al Master sobre las claves intermedias generadas, lo que permite al Master redistribuirlas de manera adecuada. Esta redistribución es clave para la correcta ejecución de la fase Reduce, en la cual los nodos Slaves intercambian las claves y valores según corresponda, asegurándose de que cada nodo trabaje con la información que le ha sido asignada.

Al finalizar la fase Reduce, cada nodo Slave envía una notificación al Master indicando la finalización de su tarea, junto con los datos finales y las estadísticas obtenidas. Esta información es recopilada y procesada por el Master, quien la presenta en la interfaz de una manera legible y práctica, facilitando el análisis y la interpretación de los resultados obtenidos.

En resumen, el circuito de mensajes es la columna vertebral de la comunicación en la aplicación, asegurando que todas las etapas del proceso de ejecución del job MapReduce se realicen de manera coordinada, eficiente y efectiva. Este mecanismo no solo garantiza la correcta distribución y ejecución de las tareas, sino que también facilita la recolección y presentación de resultados, proporcionando una experiencia de usuario optimizada y coherente con los objetivos de la aplicación.

6.3.6 Gestión de Sesiones

La gestión de sesiones en la aplicación es un componente vital que asegura la correcta coordinación y seguimiento de los nodos durante la ejecución de los jobs MapReduce. Este proceso es manejado tanto por el backend como por el frontend, cada uno asumiendo responsabilidades específicas que garantizan un funcionamiento cohesivo y sincronizado de la aplicación.

6.3.6.1 Gestión de Sesiones en el Backend

El backend es responsable de almacenar y administrar toda la información relacionada con los clústers. Esta incluye la identificación y estado de los nodos conectados, la identificación del Master y el control del estado del clúster, verificando

cuando se encuentra bloqueado y cuando no, debido al inicio de la ejecución de un job. Este enfoque centralizado permite mantener una visión coherente y actualizada del entorno de ejecución, asegurando que la aplicación pueda coordinar efectivamente las tareas distribuidas entre los nodos. Además, el backend administra los datos necesarios para la gestión del clúster, facilitando la asignación de roles y la supervisión del estado del clúster.

6.3.6.2 Gestión de Sesiones en el Frontend

El frontend complementa esta gestión al centrarse en la administración de las conexiones peer-to-peer entre los nodos conectados a un clúster específico. Además de almacenar información clave del usuario, como el nombre, identificador único, rol (Master o Slave) y el identificador del clúster, el frontend se encarga de rastrear el estado de cada nodo durante la ejecución de un job MapReduce. Esto incluye el manejo de los datos y claves que están siendo procesados, los archivos que se analizan, y el código que se ejecuta sobre dichos archivos. Adicionalmente, el frontend supervisa estadísticas de rendimiento, resultados parciales y finales, y gestiona la detección y tratamiento de errores. Esta capa de gestión asegura que el frontend no solo facilite la interacción del usuario, sino que también contribuya a la estabilidad y eficiencia general del sistema.

6.3.6.3 Coordinación Integral

La interacción entre el backend y el frontend en la gestión de sesiones garantiza que todos los nodos operen de manera sincronizada, permitiendo una ejecución eficiente y sincrónica de los procesos MapReduce. El diseño integral de la gestión de sesiones refuerza la capacidad de la aplicación para manejar trabajos distribuidos con precisión y fiabilidad, optimizando el rendimiento y la estabilidad del sistema en su conjunto.

6.3.6.4 Persistencia y Reconexión

Independientemente del rol asumido en el clúster, si el usuario refresca la pestaña del navegador mientras está conectado a la misma, el sistema maneja la reconexión automática del mismo en el clúster. Durante este proceso, se restablecen todas las conexiones peer-to-peer hacia los demás nodos del clúster. Sin embargo, debido a la naturaleza de la reconexión, todos los parámetros de la sesión se reinician.

Esto implica que cualquier archivo previamente cargado debe ser subido de nuevo, y si el usuario había presionado el botón "Listo para ejecutar", deberá realizar esta acción nuevamente. Este comportamiento también afecta a los demás nodos del clúster, que captan la reconexión y ajustan su estado de acuerdo con la nueva sesión, regresando a un estado semi-inicial si ya habían ejecutado el job o estaban listos para hacerlo. Esto garantiza la coherencia en el estado del clúster y minimiza posibles desincronizaciones entre los nodos tras un evento de reconexión.

6.3.6.5 Desconexiones Involuntarias y Manejo de Estados

Un aspecto importante a destacar es la gestión de desconexiones involuntarias. Si un usuario cierra accidentalmente la pestaña del navegador sin haber seleccionado explícitamente la opción "Cerrar clúster" o "Abandonar clúster", el estado del nodo en el clúster cambia a "desconectado". Sin embargo, el nodo no es eliminado del clúster, lo que permite que el usuario pueda volver a ingresar al mismo clúster simplemente reabriendo la pestaña cerrada. Es crucial señalar que, en la implementación actual, no existe un mecanismo programado para eliminar automáticamente un nodo que no se ha reconectado en un período determinado.

Esto significa que un nodo permanecerá en estado "desconectado" indefinidamente hasta que el usuario decida reconectarse, que sea expulsado o que el clúster sea cerrado manualmente por el Master. Este diseño asegura que los usuarios puedan recuperar su sesión sin pérdida de progreso, pero también plantea desafíos en términos de gestión eficiente de recursos y limpieza de sesiones inactivas.

6.3.7 Estadísticas

Una vez finalizada la ejecución del job, en la parte inferior de la interfaz, el sistema proporcionará estadísticas detalladas sobre dicha ejecución. En el caso de los Slaves visualizarán estadísticas propias del nodo. Por su parte, el Master visualizará las estadísticas generales del cluster, agrupando las estadísticas de todos los nodos que participaron en la ejecución del job. A continuación se describen las estadísticas generadas.

6.3.7.1 Tiempo de ejecución

Para cada etapa se calcula el tiempo de procesamiento, sin contar las posibles esperas por transferencias de datos, medido en milisegundos.

En el caso de los Slaves, simplemente se muestra el tiempo empleado para cada etapa.

Por su parte, el Master visualizará el tiempo máximo, mínimo y promedio para cada una de las etapas. Para lograr esto, a medida que los nodos finalizan cada etapa, le comunican sus tiempos de ejecución al master. Luego, una vez que finaliza la ejecución del job, el Master dispone de los datos necesarios para calcular el máximo, mínimo y promedio.

A su vez, tal como se observa en la Fig. 6.3.7.3.1, el sistema proporciona un diagrama circular donde el Master puede visualizar en forma gráfica el tiempo promedio de ejecución de cada etapa. Cada Slave visualizará los datos propios que corresponden a sus respectivos nodos.

6.3.7.2 Uso de recursos

Para la etapa *map* se lleva un control del tamaño total de las entradas procesadas, sumando el tamaño de cada uno de los archivos de entrada procesados, y de la salida generada, teniendo en cuenta el tamaño de las claves y valores generados.

Para la función *combine* simplemente se informa el tamaño de la salida, ya que el tamaño de la entrada procesada es el tamaño de la salida de la etapa map.

Finalmente, para la etapa *reduce* se calcula el tamaño total de los datos recibidos, teniendo en cuenta las claves y valores que haya recibido por parte de otros nodos, y análogamente el tamaño total de los datos que ha enviado hacia otros nodos. A su vez, se calcula el tamaño total de las entradas procesadas, teniendo en cuenta el tamaño total de los datos recibidos y los datos propios del nodo, y el tamaño total de la salida generada.

6.3.7.3 Resultados de ejecución

Tanto para la etapa *map* como para la función *combine* el sistema informa la cantidad de claves únicas generadas y la cantidad total de valores escritos.

Por su parte, para la fase *reduce* se indica la cantidad de claves y valores recibidos de otros nodos, y la cantidad enviada hacia otros nodos.

Además, para cada función se informa la cantidad de invocaciones. Para el caso de la función *map*, la cantidad de invocaciones es igual a la cantidad total de líneas a procesar, lo cual depende de los archivos de entrada que tenga asignado el nodo. Para la función *combine*, la cantidad de invocaciones es igual a la cantidad de claves generadas en la etapa *map*. Por último, para la función *reduce*, la cantidad de invocaciones dependerá de la cantidad de claves que tenga asignada el nodo.

A su vez, el Master puede visualizar la cantidad total de nodos mappers, la cual depende de cuantos nodos tengan asignado al menos un archivo de entrada, y la cantidad total de nodos reducers, que depende de a cuántos nodos se le asignó al menos una clave para reducir.

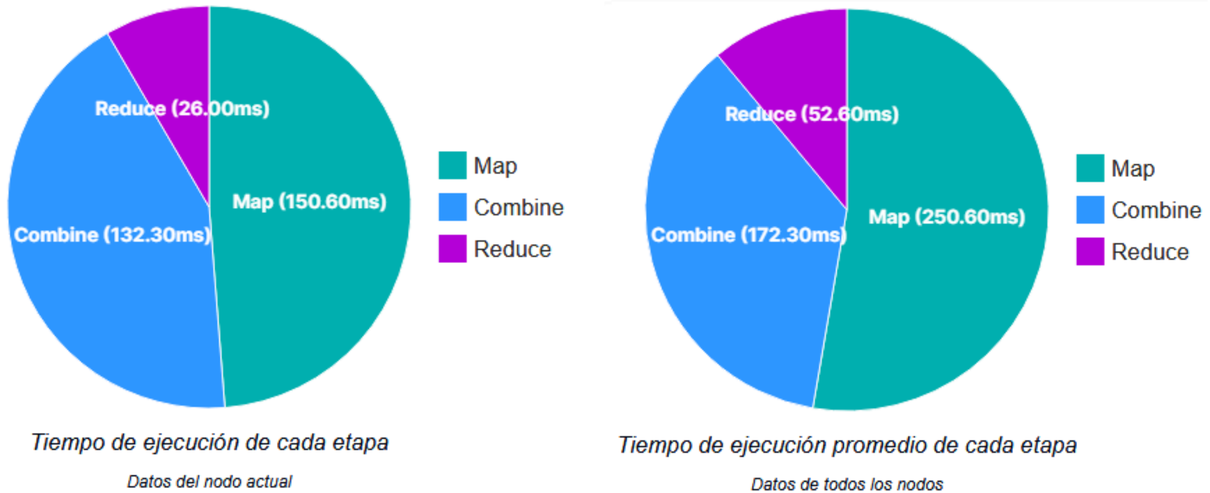


Fig. 6.3.7.3.1: Gráfico que muestra los tiempos de ejecución de cada etapa, desde la interfaz del Slave (izquierda) y Master (derecha)

6.4 Limitaciones actuales de la App

Esta sección explora las limitaciones actuales del sistema, en función de la arquitectura y las decisiones de diseño adoptadas durante su desarrollo. Aunque no se han implementado restricciones estrictas en cuanto a la cantidad de archivos o el número de nodos en el cluster, existen consideraciones prácticas que deben tomarse en cuenta al utilizar la aplicación en diferentes entornos.

6.4.1 Restricción en el Tamaño de Archivos

La única restricción programática que se ha implementado es el límite de tamaño para los archivos de entrada, que no deben exceder los 5 MB. Esta decisión se tomó para asegurar que los archivos se puedan procesar de manera eficiente dentro de los límites de recursos típicos de un entorno de pruebas, como un aula de clase o un laboratorio con recursos computacionales limitados.

Este límite está diseñado para prevenir la sobrecarga del sistema con archivos de gran tamaño que podrían afectar el rendimiento de los nodos. Aunque este límite puede ser modificado a través de una variable de entorno, la aplicación no está optimizada para manejar archivos significativamente más grandes, como los de 100 MB o 1 GB, lo que la hace más adecuada para escenarios de análisis de datos en pequeñas y medianas escalas.

6.4.2 Ausencia de Restricciones en la Cantidad de Archivos

Aunque no existe una limitación explícita en la cantidad de archivos que se pueden cargar y procesar, la capacidad práctica del sistema para manejar múltiples archivos

simultáneamente depende de los recursos del hardware y de la infraestructura de red disponible.

En entornos con recursos limitados, como un número reducido de nodos o conexiones de red con ancho de banda limitado, la falta de restricciones podría conducir a una degradación del rendimiento si se intenta procesar una gran cantidad de archivos de manera concurrente. Sin embargo, en entornos con mayor capacidad, el sistema debería ser capaz de manejar una cantidad considerable de archivos dentro de los límites del tamaño de archivo especificado.

6.4.3 Limitaciones de Conectividad y Redes

Aunque el sistema no establece un límite explícito en el número de nodos que pueden unirse al cluster, la conectividad y la capacidad de la red son factores críticos que influyen en el rendimiento de la aplicación.

Durante las pruebas iniciales, se observó que el sistema opera de manera óptima con un número moderado de nodos (hasta 15 nodos), especialmente en entornos donde la infraestructura de red no está diseñada para soportar altas cargas de tráfico de datos. Esto sugiere que, aunque el sistema puede escalar en número de nodos, es recomendable mantener el cluster en un tamaño controlado, adecuado a las condiciones de la red disponible. Esto se debe a que la cantidad de conexiones en una topología full mesh escala cuadráticamente con el número de nodos, lo que sugiere que, aunque el sistema puede escalar en número de nodos, es recomendable mantener el clúster en un tamaño controlado, adecuado a las condiciones de la red disponible.

6.4.4 Consideraciones sobre la Escalabilidad

El diseño actual del sistema no impone límites sobre la cantidad de tuplas, claves intermedias, o archivos que un nodo puede manejar. Sin embargo, la escalabilidad del sistema está intrínsecamente ligada a los recursos disponibles en el entorno de ejecución.

La ausencia de restricciones formales en la escalabilidad significa que la capacidad del sistema para manejar grandes volúmenes de datos dependerá de la infraestructura subyacente. En entornos con recursos limitados, un gran número de tuplas o claves intermedias podría llevar a una disminución del rendimiento, debido a una sobresaturación del uso de memoria del dispositivo, lo que sugiere la necesidad de optimizar el uso de recursos o introducir mecanismos de gestión de carga en el futuro.

6.4.5 Conclusiones y Recomendaciones

Aunque la aplicación no presenta restricciones programáticas significativas más allá del tamaño máximo de los archivos (lo cual es configurable a través de una variable de entorno), las pruebas y las consideraciones teóricas sugieren que su rendimiento óptimo depende del entorno en el que se utilice. Para los entornos a los cuales apuntó este desarrollo, que son aquellos de baja a media capacidad, como aulas o laboratorios, el sistema se adapta bien a las condiciones actuales.

Sin embargo, si se desea utilizar este sistema en escenarios más exigentes, sería recomendable:

- Implementar límites configurables en cuanto a la cantidad de archivos y nodos en el clúster.
- Mejorar la optimización de la red para garantizar un rendimiento constante en clústers más grandes.
- Considerar la introducción de herramientas de monitorización para identificar y gestionar posibles cuellos de botella en tiempo real.

Estas medidas permitirían extender la aplicabilidad del sistema a un rango más amplio de entornos y casos de uso, asegurando un rendimiento fiable incluso bajo condiciones más exigentes.

Capítulo 7: Pruebas funcionales de la aplicación

Este capítulo describe los problemas utilizados como ejemplo para probar y validar el sistema desarrollado. Los problemas seleccionados representan operaciones comunes en análisis de datos y fueron diseñados para evaluar diferentes aspectos del sistema, incluyendo su capacidad para realizar proyecciones, conteos, joins, clustering y agregaciones.

El dataset que se utilizó como input contiene el seguimiento de vehículos virtuales que circulan por una ciudad ficticia con 100 calles y 100 avenidas. Cada registro en el dataset representa el momento en que un vehículo llega a una nueva esquina, incluyendo la calle, avenida, el tiempo en segundos que tardó en llegar desde la esquina anterior, y el destino final si es el final del viaje. A continuación se muestra un fragmento del dataset *tráfico*.

id_vehiculo	calle	avenida	tiempo	destino
60	94	17	4983	
60	94	16	5019	
60	94	15	5055	
60	94	12	5163	
60	94	11	5199	
60	94	11	5199	Supermercado
…	…	…	…	…

Tabla 7.1: Representación gráfica en formato tabla del dataset de trafico

Además, se utilizó un segundo dataset que asocia cada vehículo con el nombre de su dueño. A continuación se muestra un fragmento del dataset *vehículo*.

nombre_dueño	id_vehiculo
Nombre_782	1
Nombre_893	10
Nombre_1013	100
Nombre_1149	101
Nombre_1258	102
Nombre_1454	104
…	…

Tabla 7.2: Representación gráfica en formato tabla del dataset vehiculo

En ambos casos el dataset se encuentra dividido en 15 archivos *txt*, cada uno con la misma cantidad de líneas, donde los valores de cada fila se encuentran separados por tab, para diferenciar cada columna.

Para cada problema estudiado, se realizaron cuatro ejecuciones del job bajo diferentes condiciones: dos con un clúster con 5 nodos slaves y dos con un clúster con 15 nodos slaves, considerando tanto una distribución pareja de los datos (asignando la misma cantidad de archivos de entrada a cada nodo), como una distribución desigual.

Las ejecuciones realizadas con 5 nodos fueron hechas sobre una misma computadora, en 5 pestañas del navegador distintas. Por otro lado, para la combinación de 15 nodos, se optó por ejecutar tanto la instancia Master como la de 12 Slaves en una misma computadora en 13 pestañas del navegador distintas, y los restantes 3 nodos Slaves se corrieron desde un mismo celular en 3 pestañas del navegador distintas.

Para las distribuciones desiguales de los datos, en el caso de 5 nodos, se asignaron conjuntos de ocho, cuatro, dos y un archivo de entrada, dejando un nodo sin datos asignados. Para el problema de join, se asignaron conjuntos de doce, seis (en dos nodos), cinco, y un archivo de entrada. Para las distribuciones desiguales de los datos en el caso de 15 nodos, se asignaron conjuntos de cinco, cuatro, tres, y un archivo de entrada (en tres nodos), dejando nueve nodos sin datos asignados. Para el problema de join, se asignaron conjuntos de seis, cinco (en dos nodos), cuatro, tres (en dos nodos), y un archivo de entrada (en tres nodos).

Para asegurar que los resultados generados por el sistema fueran correctos, se compararon con las salidas producidas por MRE, utilizando los mismos datos de entrada.

A continuación se describen los problemas utilizados para realizar las pruebas. Tanto el código para resolver cada uno, como el dataset, se encuentra disponible en el repositorio.

7.1 Problema de Proyección

Este problema se enfocó en obtener todas las esquinas distintas por las que pasó al menos un vehículo, lo que se asemeja a una operación SELECT DISTINCT en SQL:

```sql
SELECT DISTINCT calle, avenida
FROM trafico
```

Esta prueba demostró la eficiencia del sistema en operaciones de proyección y eliminación de duplicados.

Etapa Map	Función Combine	Etapa Reduce
Cantidad de nodos mappers: **5** Cantidad de invocaciones: **168723** Cantidad de archivos de entrada: **15** Tamaño total de las entradas procesadas: **2851196 bytes** Cantidad de claves únicas generadas: **9925** Cantidad de valores escritos: **168723** Tamaño total de la salida generada: **1776165 bytes** Tiempo promedio de ejecución: **190.00ms** Tiempo máximo de ejecución: **214.00ms** Tiempo mínimo de ejecución: **162.00ms**	Función no ejecutada	Cantidad de nodos reducers: **5** Cantidad de invocaciones: **9925** Cantidad de claves recibidas de otros nodos: **34053** Cantidad de valores recibidos de otros nodos: **133255** Tamaño total de los datos recibidos: **1241340 bytes** Cantidad de claves únicas enviadas a otros nodos: **34053** Cantidad de valores enviados a otros nodos: **133255** Tamaño total de los datos enviados: **1241340 bytes** Tamaño total de las entradas procesadas: **1010815 bytes** Tamaño total de la salida generada: **236670 bytes** Tiempo promedio de ejecución: **87.80ms** Tiempo máximo de ejecución: **94.00ms** Tiempo mínimo de ejecución: **85.00ms**

Tabla 7.1.1: Ejecución ***Problema de Proyección*** *con 5 nodos y datos distribuidos equitativamente*

Etapa Map	Función Combine	Etapa Reduce
Cantidad de nodos mappers: **4** Cantidad de invocaciones: **168723** Cantidad de archivos de entrada: **15** Tamaño total de las entradas procesadas: **2851196 bytes** Cantidad de claves únicas generadas: **9925** Cantidad de valores escritos: **168723** Tamaño total de la salida generada: **1586989 bytes** Tiempo promedio de ejecución: **296.50ms** Tiempo máximo de ejecución: **563.00ms** Tiempo mínimo de ejecución: **111.00ms**	Función no ejecutada	Cantidad de nodos reducers: **5** Cantidad de invocaciones: **9925** Cantidad de claves recibidas de otros nodos: **24706** Cantidad de valores recibidos de otros nodos: **144618** Tamaño total de los datos recibidos: **1140515 bytes** Cantidad de claves únicas enviadas a otros nodos: **24706** Cantidad de valores enviados a otros nodos: **144618** Tamaño total de los datos enviados: **1140515 bytes** Tamaño total de las entradas procesadas: **1010815 bytes** Tamaño total de la salida generada: **236670 bytes** Tiempo promedio de ejecución: **132.60ms** Tiempo máximo de ejecución: **160.00ms** Tiempo mínimo de ejecución: **105.00ms**

Tabla 7.1.2: Ejecución ***Problema de Proyección*** *con 5 nodos y datos con una distribución despareja*

Etapa Map	Función Combine	Etapa Reduce
Cantidad de nodos mappers: **15** Cantidad de invocaciones: **168723** Cantidad de archivos de entrada: **15** Tamaño total de las entradas procesadas: **2851196 bytes** Cantidad de claves únicas generadas: **9925** Cantidad de valores escritos: **168723** Tamaño total de la salida generada: **2553141 bytes** Tiempo promedio de ejecución: **210.49ms** Tiempo máximo de ejecución: **315.10ms** Tiempo mínimo de ejecución: **42.80ms**	Función no ejecutada	Cantidad de nodos reducers: **15** Cantidad de invocaciones: **9925** Cantidad de claves recibidas de otros nodos: **80179** Cantidad de valores recibidos de otros nodos: **156925** Tamaño total de los datos recibidos: **2140666 bytes** Cantidad de claves únicas enviadas a otros nodos: **80179** Cantidad de valores enviados a otros nodos: **156925** Tamaño total de los datos enviados: **2140666 bytes** Tamaño total de las entradas procesadas: **1010825 bytes** Tamaño total de la salida generada: **236670 bytes** Tiempo promedio de ejecución: **72.08ms** Tiempo máximo de ejecución: **195.10ms** Tiempo mínimo de ejecución: **25.00ms**

*Tabla 7.1.3: Ejecución **Problema de Proyección** con 15 nodos y datos distribuidos equitativamente*

Etapa Map	Función Combine	Etapa Reduce
Cantidad de nodos mappers: **6** Cantidad de invocaciones: **168723** Cantidad de archivos de entrada: **15** Tamaño total de las entradas procesadas: **2851196 bytes** Cantidad de claves únicas generadas: **9925** Cantidad de valores escritos: **168723** Tamaño total de la salida generada: **1798292 bytes** Tiempo promedio de ejecución: **258.50ms** Tiempo máximo de ejecución: **683.00ms** Tiempo mínimo de ejecución: **55.00ms**	Función no ejecutada	Cantidad de nodos reducers: **15** Cantidad de invocaciones: **9925** Cantidad de claves recibidas de otros nodos: **41177** Cantidad de valores recibidos de otros nodos: **162673** Tamaño total de los datos recibidos: **1509313 bytes** Cantidad de claves únicas enviadas a otros nodos: **41177** Cantidad de valores enviados a otros nodos: **162673** Tamaño total de los datos enviados: **1509313 bytes** Tamaño total de las entradas procesadas: **1010825 bytes** Tamaño total de la salida generada: **236670 bytes** Tiempo promedio de ejecución: **65.37ms** Tiempo máximo de ejecución: **194.70ms** Tiempo mínimo de ejecución: **34.00ms**

*Tabla 7.1.4: Ejecución **Problema de Proyección** con 15 nodos y datos con una distribución despareja*

7.2 Problema de Contador

Este problema buscaba contar la cantidad de viajes a cada destino distinto, excluyendo los registros sin destino. La operación es similar a un WordCount con un filtro, y se puede expresar en SQL como:

```sql
SELECT count(*)
FROM trafico
WHERE destino <> ""
GROUP BY destino
```

De esta forma, pudimos verificar la capacidad del sistema para realizar conteos selectivos.

Etapa Map	Función Combine	Etapa Reduce
Cantidad de nodos mappers: **5** Cantidad de invocaciones: **168723** Cantidad de archivos de entrada: **15** Tamaño total de las entradas procesadas: **2851196 bytes** Cantidad de claves únicas generadas: **8** Cantidad de valores escritos: **2429** Tamaño total de la salida generada: **7822 bytes** Tiempo promedio de ejecución: **81.80ms** Tiempo máximo de ejecución: **106.00ms** Tiempo mínimo de ejecución: **66.00ms**	Cantidad de invocaciones: **40** Cantidad de claves únicas generadas: **8** Cantidad de valores escritos: **40** Tamaño total de la salida generada: **700 bytes** Tiempo promedio de ejecución: **0.40ms** Tiempo máximo de ejecución: **1.00ms** Tiempo mínimo de ejecución: **0ms**	Cantidad de nodos reducers: **4** Cantidad de invocaciones: **8** Cantidad de claves recibidas de otros nodos: **32** Cantidad de valores recibidos de otros nodos: **32** Tamaño total de los datos recibidos: **512 bytes** Cantidad de claves únicas enviadas a otros nodos: **32** Cantidad de valores enviados a otros nodos: **32** Tamaño total de los datos enviados: **512 bytes** Tamaño total de las entradas procesadas: **230 bytes** Tamaño total de la salida generada: **148 bytes** Tiempo promedio de ejecución: **0.50ms** Tiempo máximo de ejecución: **1.00ms** Tiempo mínimo de ejecución: **0ms**

*Tabla 7.2.1: Ejecución **Problema de Contador** con 5 nodos y datos distribuidos equitativamente*

Etapa Map	Función Combine	Etapa Reduce
Cantidad de nodos mappers: **4** Cantidad de invocaciones: **168723** Cantidad de archivos de entrada: **15** Tamaño total de las entradas	Cantidad de invocaciones: **32** Cantidad de claves únicas generadas: **8** Cantidad de valores escritos: **32** Tamaño total de la salida generada: **560**	Cantidad de nodos reducers: **4** Cantidad de invocaciones: **8** Cantidad de claves recibidas de otros nodos: **26**

procesadas: **2851196 bytes** Cantidad de claves únicas generadas: **8** Cantidad de valores escritos: **2429** Tamaño total de la salida generada: **7717 bytes** Tiempo promedio de ejecución: **98.00ms** Tiempo máximo de ejecución: **193.00ms** Tiempo mínimo de ejecución: **27.00ms**	**bytes** Tiempo promedio de ejecución: **0ms** Tiempo máximo de ejecución: **0ms** Tiempo mínimo de ejecución: **0ms**	Cantidad de valores recibidos de otros nodos: **26** Tamaño total de los datos recibidos: **410 bytes** Cantidad de claves únicas enviadas a otros nodos: **26** Cantidad de valores enviados a otros nodos: **26** Tamaño total de los datos enviados: **410 bytes** Tamaño total de las entradas procesadas: **203 bytes** Tamaño total de la salida generada: **148 bytes** Tiempo promedio de ejecución: **0ms** Tiempo máximo de ejecución: **0ms** Tiempo mínimo de ejecución: **0ms**

Tabla 7.2.2: Ejecución ***Problema de Contador*** *con 5 nodos y datos con una distribución despareja*

Etapa Map	Función Combine	Etapa Reduce
Cantidad de nodos mappers: **15** Cantidad de invocaciones: **168723** Cantidad de archivos de entrada: **15** Tamaño total de las entradas procesadas: **2851196 bytes** Cantidad de claves únicas generadas: **8** Cantidad de valores escritos: **2429** Tamaño total de la salida generada: **8892 bytes** Tiempo promedio de ejecución: **39.66ms** Tiempo máximo de ejecución: **126.20ms** Tiempo mínimo de ejecución: **20.00ms**	Cantidad de invocaciones: **120** Cantidad de claves únicas generadas: **8** Cantidad de valores escritos: **120** Tamaño total de la salida generada: **2039 bytes** Tiempo promedio de ejecución: **0.69ms** Tiempo máximo de ejecución: **9.00ms** Tiempo mínimo de ejecución: **0ms**	Cantidad de nodos reducers: **8** Cantidad de invocaciones: **8** Cantidad de claves recibidas de otros nodos: **112** Cantidad de valores recibidos de otros nodos: **112** Tamaño total de los datos recibidos: **1790 bytes** Cantidad de claves únicas enviadas a otros nodos: **112** Cantidad de valores enviados a otros nodos: **112** Tamaño total de los datos enviados: **1790 bytes** Tamaño total de las entradas procesadas: **421 bytes** Tamaño total de la salida generada: **148 bytes** Tiempo promedio de ejecución: **0.59ms** Tiempo máximo de ejecución: **2.10ms** Tiempo mínimo de ejecución: **0ms**

Tabla 7.2.3: Ejecución **Problema de Contador** *con 15 nodos y datos distribuidos equitativamente*

Etapa Map	Función Combine	Etapa Reduce
Cantidad de nodos mappers: **6** Cantidad de invocaciones: **168723** Cantidad de archivos de entrada: **15** Tamaño total de las entradas	Cantidad de invocaciones: **48** Cantidad de claves únicas generadas: **8** Cantidad de valores escritos: **48** Tamaño total de la salida generada: **828**	Cantidad de nodos reducers: **8** Cantidad de invocaciones: **8** Cantidad de claves recibidas de otros nodos: **46**

procesadas: **2851196 bytes** Cantidad de claves únicas generadas: **8** Cantidad de valores escritos: **2429** Tamaño total de la salida generada: **7929 bytes** Tiempo promedio de ejecución: **142.67ms** Tiempo máximo de ejecución: **294.00ms** Tiempo mínimo de ejecución: **18.00ms**	**bytes** Tiempo promedio de ejecución: **0.50ms** Tiempo máximo de ejecución: **2.00ms** Tiempo mínimo de ejecución: **0ms**	Cantidad de valores recibidos de otros nodos: **46** Tamaño total de los datos recibidos: **751 bytes** Cantidad de claves únicas enviadas a otros nodos: **46** Cantidad de valores enviados a otros nodos: **46** Tamaño total de los datos enviados: **751** bytes Tamaño total de las entradas procesadas: **245 bytes** Tamaño total de la salida generada: **148 bytes** Tiempo promedio de ejecución: **0.65ms** Tiempo máximo de ejecución: **1.00ms** Tiempo mínimo de ejecución: **0ms**

Tabla 7.2.4: Ejecución **Problema de Contador** *con 15 nodos y datos con una distribución despareja*

7.3 Problema de Agregación

En este problema, se buscó obtener la cantidad de esquinas visitadas, el tiempo promedio, la calle mínima, y la avenida máxima para cada vehículo. Este problema es análogo a una operación GROUP BY en SQL, y se expresa de la siguiente manera:

```sql
SELECT count(*), avg(tiempo), min(calle), max(avenida)
FROM trafico
GROUP BY id_vehiculo
```

De esta manera corroboramos que el sistema fue capaz de procesar los datos de manera eficiente, realizando la agregación de los registros correspondientes a cada vehículo. Los resultados obtenidos mostraron un correcto agrupamiento y cálculo de los valores solicitados, validando la funcionalidad de reducción del sistema.

Etapa Map	Función Combine	Etapa Reduce
Cantidad de nodos mappers: **5** Cantidad de invocaciones: **168723** Cantidad de archivos de entrada: **15** Tamaño total de las entradas procesadas: **2851196 bytes** Cantidad de claves únicas generadas: **843** Cantidad de valores escritos: **168723** Tamaño total de la salida generada: **3212283 bytes** Tiempo promedio de ejecución:	Cantidad de invocaciones: **1861** Cantidad de claves únicas generadas: **843** Cantidad de valores escritos: **1861** Tamaño total de la salida generada: **57404 bytes** Tiempo promedio de ejecución: **14.20ms** Tiempo máximo de ejecución: **17.00ms** Tiempo mínimo de ejecución: **12.00ms**	Cantidad de nodos reducers: 5 Cantidad de invocaciones: **843** Cantidad de claves recibidas de otros nodos: **1493** Cantidad de valores recibidos de otros nodos: **1493** Tamaño total de los datos recibidos: **38582 bytes** Cantidad de claves únicas enviadas a otros nodos: **1493** Cantidad de valores enviados a otros

292.40ms Tiempo máximo de ejecución: **308.00ms** Tiempo mínimo de ejecución: **271.00ms**		nodos: **1493** Tamaño total de los datos enviados: **38582 bytes** Tamaño total de las entradas procesadas: **40092 bytes** Tamaño total de la salida generada: **35426 bytes** Tiempo promedio de ejecución: **2.40ms** Tiempo máximo de ejecución: **4.00ms** Tiempo mínimo de ejecución: **1.00ms**

Tabla 7.3.1: Ejecución ***Problema de Agregación*** *con 5 nodos y datos distribuidos equitativamente*

Etapa Map	Función Combine	Etapa Reduce
Cantidad de nodos mappers: **4** Cantidad de invocaciones: **168723** Cantidad de archivos de entrada: **15** Tamaño total de las entradas procesadas: **2851196 bytes** Cantidad de claves únicas generadas: **843** Cantidad de valores escritos: **168723** Tamaño total de la salida generada: **3209718 bytes** Tiempo promedio de ejecución: **329.25ms** Tiempo máximo de ejecución: **675.00ms** Tiempo mínimo de ejecución: **89.00ms**	Cantidad de invocaciones: **1571** Cantidad de claves únicas generadas: **843** Cantidad de valores escritos: **1571** Tamaño total de la salida generada: **48638 bytes** Tiempo promedio de ejecución: **14.50ms** Tiempo máximo de ejecución: **29.00ms** Tiempo mínimo de ejecución: **3.00ms**	Cantidad de nodos reducers: **5** Cantidad de invocaciones: **843** Cantidad de claves recibidas de otros nodos: **1259** Cantidad de valores recibidos de otros nodos: **1259** Tamaño total de los datos recibidos: **32660 bytes** Cantidad de claves únicas enviadas a otros nodos: **1259** Cantidad de valores enviados a otros nodos: **1259** Tamaño total de los datos enviados: **32660 bytes** Tamaño total de las entradas procesadas: **35051 bytes** Tamaño total de la salida generada: **35426 bytes** Tiempo promedio de ejecución: **3.80ms** Tiempo máximo de ejecución: **6.00ms** Tiempo mínimo de ejecución: **3.00ms**

Tabla 7.3.2: Ejecución ***Problema de Agregación*** *con 5 nodos y datos con una distribución despareja*

Etapa Map	Función Combine	Etapa Reduce
Cantidad de nodos mappers: **15** Cantidad de invocaciones: **168723** Cantidad de archivos de entrada: **15** Tamaño total de las entradas procesadas: **2851196 bytes** Cantidad de claves únicas generadas: **843** Cantidad de valores escritos: **168723** Tamaño total de la salida generada: **3215523 bytes** Tiempo promedio de ejecución:	Cantidad de invocaciones: **2227** Cantidad de claves únicas generadas: **843** Cantidad de valores escritos: **2227** Tamaño total de la salida generada: **68366 bytes** Tiempo promedio de ejecución: **6.45ms** Tiempo máximo de ejecución: **19.50ms** Tiempo mínimo de ejecución: **3.60ms**	Cantidad de nodos reducers: **15** Cantidad de invocaciones: **843** Cantidad de claves recibidas de otros nodos: **2078** Cantidad de valores recibidos de otros nodos: **2078** Tamaño total de los datos recibidos: **53605 bytes** Cantidad de claves únicas enviadas a otros nodos: **2078** Cantidad de valores enviados a otros

201.53ms Tiempo máximo de ejecución: **302.00ms** Tiempo mínimo de ejecución: **69.50ms**		nodos: **2078** Tamaño total de los datos enviados: **53605 bytes** Tamaño total de las entradas procesadas: **46360 bytes** Tamaño total de la salida generada: **35426 bytes** Tiempo promedio de ejecución: **1.77ms** Tiempo máximo de ejecución: **3.00ms** Tiempo mínimo de ejecución: **1.00ms**

*Tabla 7.3.3: Ejecución **Problema de Agregación** con 15 nodos y datos distribuidos equitativamente*

Etapa Map	Función Combine	Etapa Reduce
Cantidad de nodos mappers: **6** Cantidad de invocaciones: **168723** Cantidad de archivos de entrada: **15** Tamaño total de las entradas procesadas: **2851196 bytes** Cantidad de claves únicas generadas: **843** Cantidad de valores escritos: **168723** Tamaño total de la salida generada: **3211800 bytes** Tiempo promedio de ejecución: **311.50ms** Tiempo máximo de ejecución: **753.00ms** Tiempo mínimo de ejecución: **136.00ms**	Cantidad de invocaciones: **1806** Cantidad de claves únicas generadas: **843** Cantidad de valores escritos: **1806** Tamaño total de la salida generada: **55798 bytes** Tiempo promedio de ejecución: **17.67ms** Tiempo máximo de ejecución: **49.00ms** Tiempo mínimo de ejecución: **5.00ms**	Cantidad de nodos reducers: **15** Cantidad de invocaciones: **843** Cantidad de claves recibidas de otros nodos: **1685** Cantidad de valores recibidos de otros nodos: **1685** Tamaño total de los datos recibidos: **43692 bytes** Cantidad de claves únicas enviadas a otros nodos: **1685** Cantidad de valores enviados a otros nodos: **1685** Tamaño total de los datos enviados: **43692 bytes** Tamaño total de las entradas procesadas: **39199 bytes** Tamaño total de la salida generada: **35426 bytes** Tiempo promedio de ejecución: **1.84ms** Tiempo máximo de ejecución: **5.10ms** Tiempo mínimo de ejecución: **0ms**

*Tabla 7.3.4: Ejecución **Problema de Agregación** con 15 nodos y datos con una distribución despareja*

7.4 Problema de Join

En este problema, se ejecutó un join entre los datasets de tráfico y vehículos para obtener los destinos visitados por cada dueño de vehículo. El join es de tipo uno a uno, ya que cada dueño tiene un solo vehículo. La operación SQL equivalente es:

```sql
SELECT vehiculo.nombre_dueño, trafico.destino
FROM trafico INNER JOIN vehiculo ON trafico.id_vehiculo = vehiculo.id
```

A partir de esta prueba pudimos validar la capacidad del sistema para integrar datos de diferentes fuentes de manera eficiente, mediante joins uno a uno.

Etapa Map	Función Combine	Etapa Reduce
Cantidad de nodos mappers: **5** Cantidad de invocaciones: **169566** Cantidad de archivos de entrada: **30** Tamaño total de las entradas procesadas: **2866386 bytes** Cantidad de claves únicas generadas: **843** Cantidad de valores escritos: **3272** Tamaño total de la salida generada: **81818 bytes** Tiempo promedio de ejecución: **161.80ms** Tiempo máximo de ejecución: **174.00ms** Tiempo mínimo de ejecución: **133.00ms**	Función no ejecutada	Cantidad de nodos reducers: **5** Cantidad de invocaciones: **843** Cantidad de claves recibidas de otros nodos: **1944** Cantidad de valores recibidos de otros nodos: **2776** Tamaño total de los datos recibidos: **62331 bytes** Cantidad de claves únicas enviadas a otros nodos: **1944** Cantidad de valores enviados a otros nodos: **2776** Tamaño total de los datos enviados: **62331 bytes** Tamaño total de las entradas procesadas: **61239 bytes** Tamaño total de la salida generada: **39916 bytes** Tiempo promedio de ejecución: **4.20ms** Tiempo máximo de ejecución: **6.00ms** Tiempo mínimo de ejecución: **3.00ms**

*Tabla 7.4.1: Ejecución **Problema de Join** con 5 nodos y datos distribuidos equitativamente*

Etapa Map	Función Combine	Etapa Reduce
Cantidad de nodos mappers: **5** Cantidad de invocaciones: **169566** Cantidad de archivos de entrada: **30** Tamaño total de las entradas procesadas: **2866386 bytes** Cantidad de claves únicas generadas: **843** Cantidad de valores escritos: **3272** Tamaño total de la salida generada: **80115 bytes** Tiempo promedio de ejecución: **125.40ms** Tiempo máximo de ejecución: **352.00ms** Tiempo mínimo de ejecución: **3.00ms**	Función no ejecutada	Cantidad de nodos reducers: **5** Cantidad de invocaciones: **843** Cantidad de claves recibidas de otros nodos: **1691** Cantidad de valores recibidos de otros nodos: **2509** Tamaño total de los datos recibidos: **54552 bytes** Cantidad de claves únicas enviadas a otros nodos: **1691** Cantidad de valores enviados a otros nodos: **2509** Tamaño total de los datos enviados: **54552 bytes** Tamaño total de las entradas procesadas: **61239 bytes** Tamaño total de la salida generada: **39916 bytes** Tiempo promedio de ejecución: **3.00ms** Tiempo máximo de ejecución: **3.00ms** Tiempo mínimo de ejecución: **3.00ms**

*Tabla 7.4.2: Ejecución **Problema de Join** con 5 nodos y datos con una distribución despareja*

Etapa Map	Función Combine	Etapa Reduce
Cantidad de nodos mappers: **15** Cantidad de invocaciones: **169566** Cantidad de archivos de entrada: **30** Tamaño total de las entradas procesadas: **2866386 bytes** Cantidad de claves únicas generadas: **843** Cantidad de valores escritos: **3272** Tamaño total de la salida generada: **86842 bytes** Tiempo promedio de ejecución: **84.18ms** Tiempo máximo de ejecución: **244.90ms** Tiempo mínimo de ejecución: **34.90ms**	Función no ejecutada	Cantidad de nodos reducers: **15** Cantidad de invocaciones: **843** Cantidad de claves recibidas de otros nodos: **2709** Cantidad de valores recibidos de otros nodos: **3060** Tamaño total de los datos recibidos: **72562 bytes** Cantidad de claves únicas enviadas a otros nodos: **2709** Cantidad de valores enviados a otros nodos: **3060** Tamaño total de los datos enviados: **72562 bytes** Tamaño total de las entradas procesadas: **61249 bytes** Tamaño total de la salida generada: **39916 bytes** Tiempo promedio de ejecución: **2.19ms** Tiempo máximo de ejecución: **7.70ms** Tiempo mínimo de ejecución: **1.00ms**

*Tabla 7.4.3: Ejecución **Problema de Join** con 15 nodos y datos distribuidos equitativamente*

Etapa Map	Función Combine	Etapa Reduce
Cantidad de nodos mappers: **10** Cantidad de invocaciones: **169566** Cantidad de archivos de entrada: **30** Tamaño total de las entradas procesadas: **2866386 bytes** Cantidad de claves únicas generadas: **843** Cantidad de valores escritos: **3272** Tamaño total de la salida generada: **84637 bytes** Tiempo promedio de ejecución: **97.20ms** Tiempo máximo de ejecución: **362.00ms** Tiempo mínimo de ejecución: **1.00ms**	Función no ejecutada	Cantidad de nodos reducers: **15** Cantidad de invocaciones: **843** Cantidad de claves recibidas de otros nodos: **2530** Cantidad de valores recibidos de otros nodos: **3108** Tamaño total de los datos recibidos: **72133 bytes** Cantidad de claves únicas enviadas a otros nodos: **2530** Cantidad de valores enviados a otros nodos: **3108** Tamaño total de los datos enviados: **72133 bytes** Tamaño total de las entradas procesadas: **61249 bytes** Tamaño total de la salida generada: **39916 bytes** Tiempo promedio de ejecución: **21.55ms** Tiempo máximo de ejecución: **290.30ms** Tiempo mínimo de ejecución: **0ms**

*Tabla 7.4.4: Ejecución **Problema de Join** con 15 nodos y datos con una distribución despareja*

7.5 Problema de Clustering (k-means)

Este problema consistió en ejecutar una iteración del algoritmo k-means para agrupar vehículos en clusters según sus rutas. El k-means es un algoritmo iterativo que recalcula los centroides de los clústers hasta que converge. Dado que el algoritmo iterativo consiste en la ejecución del mismo job múltiples veces hasta alcanzar el estado de convergencia, sólo se simuló una iteración completa del algoritmo, involucrando la ejecución del job y la actualización de los centroides.

Etapa Map	Función Combine	Etapa Reduce
Cantidad de nodos mappers: **5** Cantidad de invocaciones: **168723** Cantidad de archivos de entrada: **15** Tamaño total de las entradas procesadas: **2851196 bytes** Cantidad de claves únicas generadas: **5** Cantidad de valores escritos: **168723** Tamaño total de la salida generada: **2181534 bytes** Tiempo promedio de ejecución: **702.40ms** Tiempo máximo de ejecución: **738.00ms** Tiempo mínimo de ejecución: **680.00ms**	Cantidad de invocaciones: **25** Cantidad de claves únicas generadas: **5** Cantidad de valores escritos: **25** Tamaño total de la salida generada: **775 bytes** Tiempo promedio de ejecución: **18.20ms** Tiempo máximo de ejecución: **22.00ms** Tiempo mínimo de ejecución: **13.00ms**	Cantidad de nodos reducers: **5** Cantidad de invocaciones: **5** Cantidad de claves recibidas de otros nodos: **20** Cantidad de valores recibidos de otros nodos: **20** Tamaño total de los datos recibidos: **560 bytes** Cantidad de claves únicas enviadas a otros nodos: **20** Cantidad de valores enviados a otros nodos: **20** Tamaño total de los datos enviados: **560 bytes** Tamaño total de las entradas procesadas: **560 bytes** Tamaño total de la salida generada: **257 bytes** Tiempo promedio de ejecución: **0.40ms** Tiempo máximo de ejecución: **1.00ms** Tiempo mínimo de ejecución: **0ms**

*Tabla 7.5.1: Ejecución **Problema de Clustering (k-means)** con 5 nodos y datos distribuidos equitativamente*

Etapa Map	Función Combine	Etapa Reduce
Cantidad de nodos mappers: **4** Cantidad de invocaciones: **168723** Cantidad de archivos de entrada: **15** Tamaño total de las entradas procesadas: **2851196 bytes** Cantidad de claves únicas generadas: **5** Cantidad de valores escritos: **168723** Tamaño total de la salida generada: **2181501 bytes**	Cantidad de invocaciones: **20** Cantidad de claves únicas generadas: **5** Cantidad de valores escritos: **20** Tamaño total de la salida generada: **624 bytes** Tiempo promedio de ejecución: **23.00ms** Tiempo máximo de ejecución: **58.00ms** Tiempo mínimo de ejecución: **6.00ms**	Cantidad de nodos reducers: **5** Cantidad de invocaciones: **5** Cantidad de claves recibidas de otros nodos: **16** Cantidad de valores recibidos de otros nodos: **16** Tamaño total de los datos recibidos: **450 bytes** Cantidad de claves únicas enviadas a

Tiempo promedio de ejecución: **741.75ms** Tiempo máximo de ejecución: **1.53s** Tiempo mínimo de ejecución: **216.00ms**		otros nodos: **16** Cantidad de valores enviados a otros nodos: **16** Tamaño total de los datos enviados: **450 bytes** Tamaño total de las entradas procesadas: **457 bytes** Tamaño total de la salida generada: **257 bytes** Tiempo promedio de ejecución: **0.20ms** Tiempo máximo de ejecución: **1.00ms** Tiempo mínimo de ejecución: **0ms**

*Tabla 7.5.2: Ejecución **Problema de Clustering (k-means)** con 5 nodos y datos con una distribución despareja*

Etapa Map	Función Combine	Etapa Reduce
Cantidad de nodos mappers: **15** Cantidad de invocaciones: **168723** Cantidad de archivos de entrada: **15** Tamaño total de las entradas procesadas: **2851196 bytes** Cantidad de claves únicas generadas: **5** Cantidad de valores escritos: **168723** Tamaño total de la salida generada: **2181884 bytes** Tiempo promedio de ejecución: **560.65ms** Tiempo máximo de ejecución: **757.00ms** Tiempo mínimo de ejecución: **339.20ms**	Cantidad de invocaciones: **75** Cantidad de claves únicas generadas: **5** Cantidad de valores escritos: **75** Tamaño total de la salida generada: **2247 bytes** Tiempo promedio de ejecución: **15.21ms** Tiempo máximo de ejecución: **116.00ms** Tiempo mínimo de ejecución: **5.20ms**	Cantidad de nodos reducers: **5** Cantidad de invocaciones: **5** Cantidad de claves recibidas de otros nodos: **70** Cantidad de valores recibidos de otros nodos: **70** Tamaño total de los datos recibidos: **1886 bytes** Cantidad de claves únicas enviadas a otros nodos: **70** Cantidad de valores enviados a otros nodos: **70** Tamaño total de los datos enviados: **1886 bytes** Tamaño total de las entradas procesadas: **1532 bytes** Tamaño total de la salida generada: **257 bytes** Tiempo promedio de ejecución: **1.02ms** Tiempo máximo de ejecución: **2.00ms** Tiempo mínimo de ejecución: **0ms**

*Tabla 7.5.3: Ejecución **Problema de Clustering (k-means)** con 15 nodos y datos distribuidos equitativamente*

Etapa Map	Función Combine	Etapa Reduce
Cantidad de nodos mappers: **6** Cantidad de invocaciones: **168723** Cantidad de archivos de entrada: **15** Tamaño total de las entradas procesadas: **2851196 bytes** Cantidad de claves únicas generadas: **5** Cantidad de valores escritos: **168723** Tamaño total de la salida generada: **2181569 bytes**	Cantidad de invocaciones: **30** Cantidad de claves únicas generadas: **5** Cantidad de valores escritos: **30** Tamaño total de la salida generada: **918 bytes** Tiempo promedio de ejecución: **16.00ms** Tiempo máximo de ejecución: **29.00ms** Tiempo mínimo de ejecución: **7.00ms**	Cantidad de nodos reducers: **5** Cantidad de invocaciones: **5** Cantidad de claves recibidas de otros nodos: **30** Cantidad de valores recibidos de otros nodos: **30** Tamaño total de los datos recibidos: **828 bytes** Cantidad de claves únicas enviadas a

Tiempo promedio de ejecución: **722.33ms** Tiempo máximo de ejecución: **1.23s** Tiempo mínimo de ejecución: **327.00ms**		otros nodos: **30** Cantidad de valores enviados a otros nodos: **30** Tamaño total de los datos enviados: **828 bytes** Tamaño total de las entradas procesadas: **653 bytes** Tamaño total de la salida generada: **257 bytes** Tiempo promedio de ejecución: **0.46ms** Tiempo máximo de ejecución: **1.00ms** Tiempo mínimo de ejecución: **0ms**

*Tabla 7.5.4: Ejecución **Problema de Clustering (k-means)** con 15 nodos y datos con una distribución despareja*

7.6 Conclusiones

Los resultados obtenidos confirman que el sistema es competente en realizar una amplia gama de operaciones comunes en el procesamiento de datos distribuidos.

El análisis de rendimiento sobre las distintas configuraciones probadas sobre los problemas descritos, demuestra que, en general, utilizar un mayor número de nodos en combinación con una distribución uniforme de los datos tiende a ofrecer un rendimiento superior. No se ha visto una diferencia a favor de alguna otra configuración en ninguna de las ejecuciones de los diferentes problemas. Esto se debe a que la paralelización lograda con más nodos permite distribuir la carga de trabajo de manera más efectiva, reduciendo los tiempos de procesamiento en las diferentes etapas.

Particularmente, la etapa de reducción se beneficia significativamente de un mayor número de nodos, dado que el trabajo se divide en más partes, lo que minimiza los tiempos de ejecución. Sin embargo, es importante señalar que una distribución desigual de los datos puede contrarrestar estas mejoras, generando cargas desbalanceadas entre los nodos y aumentando los tiempos de ejecución, especialmente en la etapa de mapeo, que es más sensible a cómo se distribuyen los datos inicialmente.

Por lo tanto, para optimizar el rendimiento en entornos de procesamiento distribuido, es recomendable optar por configuraciones con una mayor cantidad de nodos, siempre que se mantenga una distribución de datos lo más uniforme posible. Esta combinación no solo mejora la eficiencia global del sistema, sino que también minimiza la variabilidad en los tiempos de ejecución, garantizando un procesamiento más consistente y predecible.

Capítulo 8: Problemas y Soluciones en el Desarrollo

En el desarrollo de la aplicación se presentaron varios desafíos técnicos que requirieron soluciones específicas para garantizar el correcto funcionamiento del sistema. Estos problemas surgieron tanto en la fase de implementación como en la de despliegue, y no sólo dificultaron la implementación de ciertas funcionalidades, sino que también requirieron un análisis exhaustivo para encontrar soluciones efectivas que mantuvieran la integridad del sistema, ofreciendo cada uno de ellos un aprendizaje valioso sobre la complejidad de la interconexión de nodos en una arquitectura distribuida. En este capítulo, se abordarán los problemas más significativos que surgieron durante el desarrollo, sus causas y las soluciones que se implementaron para superarlos.

8.1 NAT Simétrico y Asimétrico

Uno de los primeros y más críticos obstáculos que enfrentamos surgió al desplegar la aplicación en servicios de la nube. Estos servicios alojaban tanto el frontend como el backend de la aplicación.

Fig. 8.1.1: Un mundo sin NAT ni firewalls

Fuente: (Dutton, 2013)

Frente al desconocimiento inicial de lo que se explicará a continuación, habiamos planteado un esquema tal como se muestra en la Fig. 8.1.1.1, pero durante las primeras pruebas, observamos que las conexiones entre los nodos, conocidos como *peers*, no se establecían correctamente, o directamente no lo hacían, lo que impedía la comunicación efectiva entre ellos. Este problema se identificó como resultado de la forma en que las conexiones son manejadas por los *Network Address Translation* (NAT)[8], específicamente en su modalidad simétrica y asimétrica. Es así como nos encontramos en un escenario con una capa de dificultad más, tal como se observa en la Fig. 8.1.1.2.

[8] Proceso de traducción de direcciones IP y puerto cuando el paquete pasa a través de un router o firewall.

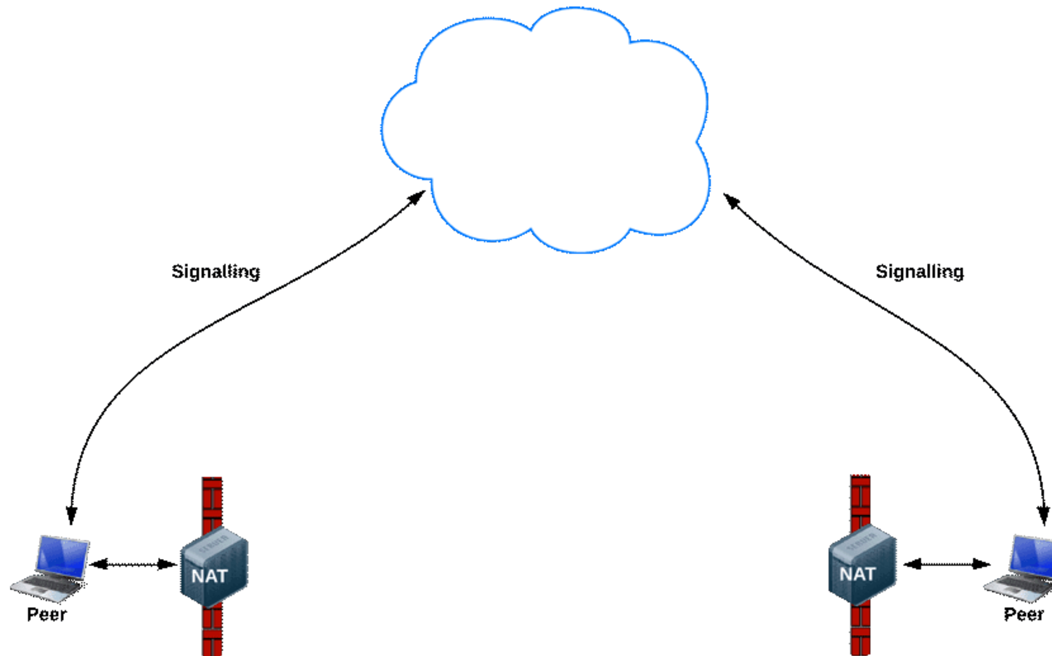


Fig. 8.1.2: El mundo real

Fuente: (Dutton, 2013)

El NAT es una técnica que permite a varios dispositivos en una red privada compartir una única dirección IP pública para comunicarse con redes externas. Sin embargo, en un NAT simétrico, cada solicitud saliente desde la misma dirección IP interna pero hacia diferentes destinos externos recibe un puerto público distinto. Este comportamiento complica la conexión directa entre nodos, ya que las direcciones IP y puertos no coinciden cuando los nodos están detrás de diferentes NATs, lo que resultó en intentos de conexión fallidos, e imposibilitó la conexión.

En contraste, en un NAT asimétrico, los puertos públicos asignados a una conexión se mantienen consistentes para todas las comunicaciones del nodo, facilitando la comunicación entre *peers* bajo ciertas condiciones. No obstante, en nuestro despliegue en la nube, donde los *peers* estaban distribuidos geográficamente y bajo diferentes configuraciones de NAT, la imposibilidad de establecer conexiones directas entre los nodos se debía a las discrepancias en las traducciones de IP y puertos, lo que requería una solución más avanzada para superar estas limitaciones.

Una aclaración importante es que, cuando se opera en un entorno local, se debe estar conectado a la red Wi-Fi y configurarla como una red privada en sistemas operativos como Windows, para que sea detectada como segura y permita las conexiones. Además, en redes públicas, es posible que se necesite desactivar el firewall para habilitar las conexiones, aunque esta práctica no es recomendable por motivos de seguridad.

8.2 Servidores STUN y TURN

Para abordar el problema de conectividad causado por los NATs simétricos, consideramos la implementación de servidores STUN (Session Traversal Utilities for NAT) y TURN (Traversal Using Relays around NAT). Los servidores STUN permiten a los nodos descubrir su dirección IP pública y el puerto que se utiliza para el tráfico saliente, proporcionando la información necesaria para intentar establecer una conexión directa con otros nodos.

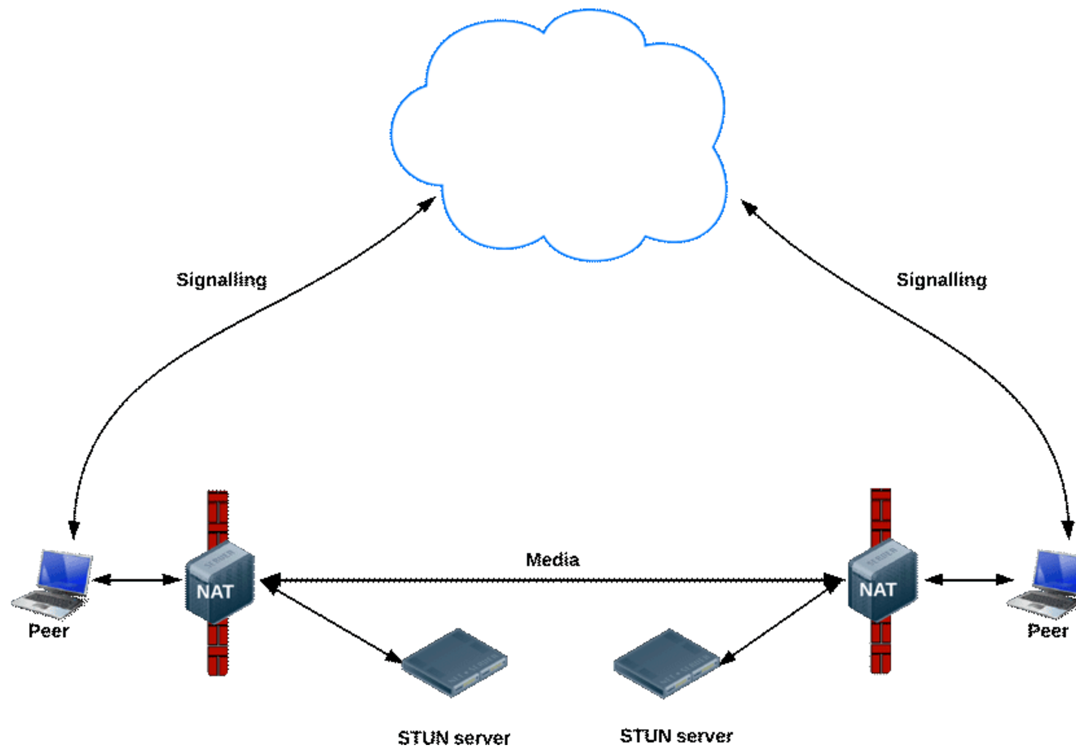


Fig. 8.2.1: Uso de servidores STUN para obtener direcciones IP:puertos públicas

Fuente: (Dutton, 2013)

Sin embargo, en entornos donde el NAT simétrico está presente, esta información por sí sola no es suficiente para garantizar la conectividad.

Aquí es donde entra en juego el servidor TURN, que actúa como intermediario, reenviando el tráfico entre los nodos cuando el NAT simétrico impide una conexión directa.

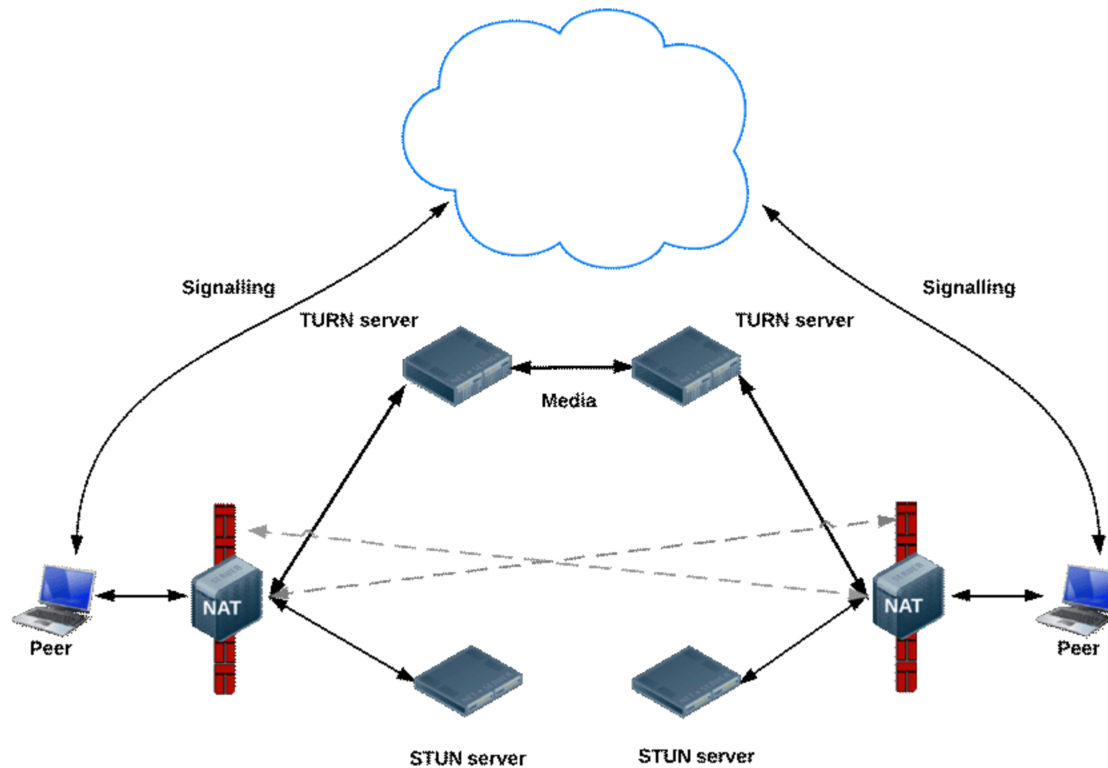


Fig. 8.2.2: El Monty completo: STUN, TURN y señalización

Fuente: (Dutton, 2013)

Aunque la implementación de servidores STUN y TURN podría haber resuelto el problema de conectividad, introducía una nueva complejidad en la arquitectura de la aplicación. El despliegue de estos servidores no solo requería recursos adicionales, sino también la administración continua de un servidor dedicado, lo que aumentaba significativamente la carga de infraestructura y el mantenimiento. Además de esto, se sumaba la variable de un servidor (TURN) el cual tenía que encargarse de distribuir todo el tráfico de la aplicación a través de la red, perdiendo el foco inicial, el cual era conseguir que las conexiones sean peer-to-peer evitando una sobrecarga sobre un solo servidor.

Dado este panorama, se tomó la decisión de limitar el alcance de la aplicación a un entorno de red local (LAN). Al operar en una red local, todos los nodos estarían bajo la misma subred, eliminando así la necesidad de servidores STUN y TURN, ya que el problema de NAT desaparece en este contexto. Las conexiones directas entre nodos se establecen sin necesidad de intermediarios, simplificando la infraestructura y eliminando la complejidad adicional. Sin embargo, esta solución también introdujo una limitación importante: la aplicación sólo podría ejecutarse en escenarios donde todos los nodos estuvieran conectados físicamente a la misma red local, restringiendo su aplicabilidad a entornos controlados como laboratorios o redes privadas.

Además, dado que la aplicación se ejecutará en una red local, se optó por *dockerizar* toda la arquitectura para simplificar el despliegue en cualquier dispositivo dentro de esa red. Con Docker, se consiguió encapsular cada componente en un contenedor, facilitando así el despliegue y la administración de toda la aplicación en diferentes dispositivos. Esta solución no solo reduce la complejidad de la configuración, sino que también asegura un entorno de ejecución coherente y controlado para todos los nodos.

8.3 Implementación de HTTPS en Red Local

Durante el desarrollo del sistema, surgió la necesidad de configurar un entorno seguro utilizando HTTPS, a pesar de encontrarnos dentro de una red local. Esto se debió principalmente, a la integración de la librería **react-py**, la cual se apoya en Pyodide para ejecutar Python en el navegador. Esta herramienta depende de tecnologías como los *Service Workers*[9] y funciones criptográficas como *Crypto.randomUUID()*[10], las cuales desde su especificación, remarcan la necesidad de un entorno seguro para funcionar correctamente.

Dado que la aplicación debía operar en una red local, era imperativo que la arquitectura estuviera configurada sobre HTTPS para cumplir con estos requisitos de seguridad. El uso de HTTPS (Hypertext Transfer Protocol Secure) en una red local presentaba una serie de desafíos específicos, principalmente relacionados con la generación y gestión de certificados SSL/TLS. Mientras que en entornos de producción pública el uso de HTTPS es esencial para garantizar la seguridad de las comunicaciones, su implementación en redes locales requiere un enfoque diferente debido a la ausencia de un dominio público y la necesidad de garantizar la confianza en los certificados generados.

Para superar estos obstáculos, se optó por utilizar un generador de certificados automatizado dentro del entorno dockerizado. Este generador, parte del servicio **cert-generator**, se encargó de crear certificados SSL válidos para la red local y configurar automáticamente a Nginx para utilizar estos certificados. Esta solución no solo facilitó la implementación de HTTPS en el entorno local, sino que también mejoró la seguridad y redujo la complejidad de la gestión de certificados en la red.

---

[9] Un service worker es un worker manejado por eventos registrado para una fuente y una ruta. Consiste en un fichero JavaScript que controla la página web (o el sitio) con el que está asociado, interceptando y modificando la navegación y las peticiones de recursos, y cacheando los recursos de manera muy granular para ofrecer un control completo sobre cómo la aplicación debe comportarse en ciertas situaciones, por ejemplo, cuando la red no está disponible. https://developer.mozilla.org/es/docs/Web/API/Service_Worker_API

[10] El método randomUUID() de la interfaz Crypto se utiliza para generar un UUID v4 utilizando un generador de números aleatorios criptográficamente seguro. https://developer.mozilla.org/en-US/docs/Web/API/Crypto/randomUUID

El proceso de generación de los certificados SSL se realiza mediante un script que automatiza la creación de claves y certificados autofirmados utilizando OpenSSL. El código del cert-generator se detalla a continuación:

```
#!/bin/sh

# Obtener los nombres de los archivos de las variables de entorno
SERVER_KEY_NAME=${SERVER_KEY_NAME:-"server.key"}
SERVER_CERT_NAME=${SERVER_CERT_NAME:-"server.crt"}

printf $SERVER_CERT_NAME

printf "Generando certificados SSL autofirmados...\n"

# Verificar si /certs es un directorio
if [ no -d /certs ]; then
  # Si /certs no es un directorio, intentar crearlo
  mkdir -p /certs
fi

openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout
/certs/$SERVER_KEY_NAME -out /certs/$SERVER_CERT_NAME -subj "/CN=localhost"

if [ -f /certs/$SERVER_KEY_NAME ] && [ -f /certs/$SERVER_CERT_NAME ]; then
  echo "¡Certificados SSL autofirmados generados y movidos correctamente!"
else
  echo "Error al generar los certificados SSL autofirmados"
  exit 1
fi
```

El script crea un par de claves RSA de 2048 bits y un certificado X.509 válido por 365 días, todo ello sin la intervención manual del usuario. La clave privada y el certificado se almacenan en un directorio específico dentro del contenedor Docker. Con esta configuración, todos los servicios de la aplicación pueden utilizar estos certificados para establecer conexiones HTTPS, garantizando la compatibilidad con las funciones críticas de react-py y Pyodide.

Sin embargo, es importante resaltar que, aunque se implementaron medidas para garantizar la seguridad en la red local, la aplicación de HTTPS en este contexto sigue presentando limitaciones en comparación con un entorno de producción público. La principal diferencia radica en la necesidad de aceptar manualmente la confianza en los certificados sobre cada dispositivo nuevo que se conecte a la aplicación.

En resumen, la implementación de HTTPS fue una solución indispensable para cumplir con los requisitos de seguridad de la aplicación, permitiendo el uso

adecuado de las tecnologías necesarias para su correcto funcionamiento dentro de una red local.

8.4 Particionado de Mensajes por la Red WebRTC en Chunks

A medida que avanzamos en el desarrollo de la aplicación, comenzamos a probar la transmisión de archivos más realistas, que podrían ser utilizados en un entorno académico, tales como archivos de texto con cientos o incluso miles de líneas. Fue en este punto cuando nos encontramos con un nuevo desafío. El problema principal se debía a que los archivos superaban el límite de tamaño de los mensajes que WebRTC podía manejar en una sola transmisión, lo que resultaba en fallos o bloqueos durante la transmisión. La librería **simple-peer**, que utilizábamos para manejar las conexiones peer-to-peer, no controlaba estos escenarios, lo que nos obligó a implementar una solución personalizada para particionar los mensajes en fragmentos más pequeños, conocidos como chunks.

Tras investigar, en la documentación y sobre las mejores prácticas, determinamos que un tamaño ideal para los chunks era de 16 KB. Esta elección se basó en el equilibrio entre la eficiencia de transmisión y la minimización de la latencia. Con esta configuración, desarrollamos un sistema para dividir los archivos en estos chunks y transmitirlos secuencialmente a través de WebRTC.

8.4.1 Implementación de la Transmisión de Chunks

El proceso de envío de un archivo se inicia dividiendo el archivo en fragmentos de 16 KB, cada uno con un encabezado que incluye metadatos como el identificador único del archivo (UUID), el índice del chunk y el número total de chunks. Estos fragmentos se envían de manera secuencial al peer destinatario. A medida que los chunks llegan al destinatario, éste los almacena temporalmente y los reensambla en su orden correcto utilizando la información contenida en los encabezados. Una vez que se han recibido todos los chunks, el archivo es reconstruido y está listo para ser utilizado.

El código de envío de archivos fue diseñado para manejar estos fragmentos de manera eficiente, asegurando que cada chunk se envíe correctamente. Además, implementamos un sistema de reensamblaje en el lado del receptor que se encarga de verificar que todos los chunks hayan llegado y estén en el orden correcto antes de reconstruir el archivo completo.

8.4.2 Desafíos y Soluciones

Uno de los principales desafíos en esta implementación fue asegurar que los chunks se recibieran en el orden correcto y sin pérdidas, para evitar causar corrupción de datos. Para abordar este problema, añadimos un mecanismo de control de errores que verifica la integridad de los datos antes de reensamblarlos. Si algún chunk no llega o se corrompe, el archivo o mensaje es descartado.

Además, como WebRTC es una tecnología asíncrona, donde los paquetes pueden llegar fuera de orden o con diferentes latencias, implementamos buffers para gestionar la recepción de chunks de manera fluida, minimizando los impactos de la variabilidad en la red.

En resumen, la solución de particionado de mensajes en chunks permitió a la aplicación manejar la transmisión de archivos grandes de manera eficiente y robusta, resolviendo el problema inicial de los fallos en la transmisión. Aunque la implementación añadió un nivel de complejidad al desarrollo, resultó esencial para cumplir con los requisitos del proyecto y garantizar que la aplicación funcionara correctamente en un entorno de comunicación en tiempo real, incluso con archivos de gran tamaño.

En el siguiente capítulo, se presentarán las conclusiones generales del proyecto, resumiendo los principales hallazgos y aprendizajes obtenidos durante su desarrollo. Además, se explorarán posibles mejoras que podrían implementarse en el futuro y se sugerirán líneas de investigación que podrían expandir o perfeccionar el trabajo realizado, con el objetivo de seguir mejorando la aplicación en términos de eficiencia, escalabilidad y usabilidad.

Capítulo 9: Conclusiones y Trabajo Futuro

El desarrollo de este sistema ha representado un desafío significativo que involucró la integración de múltiples tecnologías y la aplicación de conocimientos adquiridos durante la carrera de Licenciatura en Sistemas. A lo largo del proceso, no solo se logró construir una solución funcional, sino que también se adquirieron nuevas habilidades y se consolidaron conceptos clave en áreas como arquitectura de software, redes, programación distribuida y tecnologías emergentes. Este capítulo ofrece una reflexión sobre los logros obtenidos, las dificultades superadas, y cómo la experiencia contribuyó a nuestro crecimiento profesional. Además, se presentan posibles mejoras y áreas de investigación futura que pueden expandir las capacidades de la aplicación y abrir nuevas oportunidades para su evolución.

9.1 Resumen de los Hallazgos

A lo largo del desarrollo de esta tesina, se ha logrado alcanzar un conocimiento profundo sobre múltiples aspectos clave en la implementación de sistemas distribuidos y aplicaciones web modernas. El proyecto presentado no solo permitió aplicar los conocimientos adquiridos durante la carrera de Licenciatura en Sistemas, sino también explorar y dominar nuevas tecnologías que se encuentran en la vanguardia del desarrollo de software.

Una tarea fundamental en la etapa inicial fue la investigación de las herramientas adecuadas para llevar a cabo esta aplicación. Esta investigación se vió facilitada por la amplia experiencia adquirida durante la carrera, donde se aprendieron y utilizaron diversas tecnologías que fueron fundamentales para concretar el proyecto. A lo largo del desarrollo, se siguieron las mejores prácticas aprendidas tanto en programación como en la administración de tareas y la gestión del trabajo en equipo, siempre preparados para adaptarse a cambios constantes.

Uno de los principales logros, fue la implementación exitosa de un sistema que permite la distribución de tareas mediante el paradigma MapReduce, un concepto central en el procesamiento de grandes volúmenes de datos. A través de esta arquitectura, se pudo dividir un problema complejo en tareas más pequeñas y manejables, distribuyendo eficientemente la carga de trabajo entre diferentes nodos en un clúster. Este enfoque se complementó con la incorporación de WebRTC, una tecnología web que habilita la comunicación directa entre pares, lo cual fue fundamental para asegurar la transferencia de datos en tiempo real dentro de una red.

Otro aspecto crucial del proyecto fue la implementación y gestión de un sistema con completa sincronización entre los nodos del clúster, permitiendo de esta manera la correcta ejecución de las tareas distribuidas. La sincronización entre el nodo Master

y los nodos Slaves fue esencial para garantizar que el flujo de trabajo se llevara a cabo de manera ordenada y eficiente. Mediante un circuito de mensajería a través de WebRTC, el nodo Master dirige la carga de trabajo hacia a los nodos Slaves, y reparte los datos generados en cada etapa del proceso. Este mecanismo de comunicación y sincronización requirió un análisis exhaustivo de las herramientas actuales empleadas en la industria y el ámbito académico, permitiendo definir un circuito de mensajes que asegura la ejecución coordinada de los jobs MapReduce, en donde cada etapa opera en sincronía con los demás nodos del clúster.

Uno de los hallazgos relevantes fue la complejidad inherente a la gestión de conexiones en entornos con diferentes configuraciones de NAT (Network Address Translation). Durante la fase de despliegue en la nube, se encontraron problemas con la conectividad entre pares debido a la presencia de NAT simétricos y asimétricos, lo que limitó inicialmente la funcionalidad de la aplicación en redes más amplias. Este desafío llevó a la investigación y comprensión de tecnologías, como los servidores STUN y TURN, cuyo propósito es facilitar la conexión en tales redes. Sin embargo, se optó por limitar la aplicación a un entorno de red local, llevando a cabo un proceso de dockerización de la arquitectura para simplificar el despliegue y mejorar la estabilidad.

Otro aspecto importante fue la necesidad de operar la aplicación sobre HTTPS en la red local. Esta decisión, fue motivada principalmente por la implementación de la librería react-py, la cual utiliza funciones que requieren un entorno seguro para su ejecución, como lo son por ejemplo, el método randomUUID() de la clase Crypto, y utilidades de la plataforma web como los Service Workers. La generación de certificados SSL autofirmados localmente fue un avance crucial, permitiendo asegurar las comunicaciones y cumpliendo con los requisitos técnicos de la aplicación.

Por otra parte, la experiencia adquirida en la manipulación de datos y el manejo de grandes archivos en la red WebRTC fue otro punto a destacar. Luego de pruebas e investigación, se identificó que los archivos de gran tamaño superaban el límite de transferencia única permitida por las herramientas utilizadas para el envío de datos y archivos por la red. Este problema fue resuelto mediante la división de los mensajes o archivos a ser transferidos, en fragmentos más pequeños de 16 KB cada uno, lo que permitió su envío eficiente a través de la red. La solución requirió un control manual del particionado de los mensajes, evidenciando la importancia de ajustar las herramientas existentes a las necesidades específicas del proyecto.

En resumen, la tesina ha sido un ejercicio integral que abarcó desde la comprensión teórica de los paradigmas de procesamiento distribuido hasta la implementación práctica de una solución completa y funcional. La experiencia adquirida no solo ha reforzado los conocimientos previamente adquiridos, sino que también ha abierto nuevas áreas de aprendizaje, especialmente en el campo de las comunicaciones en

tiempo real y la seguridad en aplicaciones web. Los hallazgos obtenidos durante el desarrollo de este proyecto sientan una base sólida para futuras investigaciones y mejoras en la aplicación, las cuales se abordarán en detalle en la siguiente sección.

9.2 Posibles Mejoras y Futuras Investigaciones

Este proyecto ha sentado las bases para una aplicación funcional y efectiva, pero existen numerosas oportunidades para mejorar su robustez, extender sus capacidades y explorar nuevas áreas de desarrollo. A continuación, se presentan algunas sugerencias para futuros estudiantes, investigadores o entusiastas de la tecnología que deseen continuar y expandir este trabajo. Estas sugerencias no solo buscan añadir nuevas características, sino también solucionar limitaciones actuales y explorar el potencial de la aplicación en diferentes contextos.

9.2.1 Implementación de las Etapas Shuffle y Sort

Las etapas *shuffle* y *sort* son cruciales en la ejecución de algoritmos MapReduce, ya que determinan cómo se distribuyen y ordenan los datos entre los nodos del clúster. En la versión actual del sistema, estas etapas no están implementadas. El desafío principal radica en permitir que el nodo master pueda definir y personalizar el código tanto para la etapa *shuffle* como para la etapa *sort*, especificando la lógica de distribución y ordenamiento de las claves y valores.

Esta capacidad añadida daría a los usuarios un control granular sobre la manipulación de datos, lo que sería particularmente útil en escenarios donde las operaciones *shuffle* y *sort* requieren una lógica específica para optimizar el rendimiento o resolver problemas complejos. Además, la correcta implementación y sincronización de estas etapas entre los nodos *slave*, quienes almacenan las salidas de la etapa *map* y/o la función de optimización *combine*, sería esencial para asegurar una ejecución eficiente y coherente del proceso MapReduce.

9.2.2 Soluciones Iterativas

Actualmente, aunque es suficiente para resolver muchos problemas comunes, el sistema no permite la ejecución iterativa de un mismo job MapReduce. Implementar esta capacidad permitiría desarrollar soluciones iterativas que realicen cálculos más avanzados, para resolver problemas donde los resultados de una iteración se utilizan como entrada para la siguiente.

De esta forma, la posibilidad de implementar soluciones iterativas dentro del sistema, ampliaría significativamente su ámbito de aplicación, permitiendo su uso en algoritmos de aprendizaje automático, análisis de grafos y otras áreas que requieren múltiples ciclos de procesamiento de datos. Esto no solo aumenta la versatilidad del

sistema, sino que también lo alinea con prácticas y necesidades actuales en la industria y la investigación académica.

9.2.3 Configuración de Funciones Map para Diferentes Conjuntos de Datos

Una de las limitaciones de la versión actual del sistema es la incapacidad de asignar diferentes funciones *map* a conjuntos de datos específicos. Esta característica sería especialmente útil en escenarios donde un solo job requiere manejar múltiples tipos de datos que deben ser procesados de maneras distintas.

Implementar esta funcionalidad permitiría abordar una gama más amplia de problemas MapReduce, otorgando mayor libertad en el manejo y manipulación de los datos. Además, esto facilitaría la ejecución de soluciones con grafos acíclicos dirigidos (DAGs), donde la complejidad de las dependencias entre tareas requiere un control detallado sobre la asignación de funciones para procesar los datos.

9.2.4 Administración de Parámetros y Variables Globales

En sistemas distribuidos, la capacidad de manejar parámetros o variables globales accesibles por todos los nodos del clúster es vital para coordinar operaciones y compartir estados. Actualmente, el sistema no cuenta con un mecanismo para gestionar estas variables, lo que limita su capacidad para ejecutar trabajos más complejos que requieren un contexto compartido entre nodos.

Incorporar esta funcionalidad permitiría a los usuarios definir y utilizar variables globales que podrían ser accedidas y modificadas durante la ejecución de un job MapReduce. Esto sería particularmente útil para implementar comportamientos más sofisticados, como la incorporación de estadísticas de ejecución en tiempo real, la coordinación de tareas entre nodos y el compartir constantes necesarias por todos los nodos.

9.2.5 Integración de IntelliSense en el Editor de Código

Una de las mejoras clave para optimizar la experiencia del usuario es la integración de un sistema de autocompletado, similar a IntelliSense, en el editor de código de la aplicación. Este sistema proporcionaría sugerencias contextuales mientras los usuarios escriben código, lo que no solo aceleraría el proceso de desarrollo, sino que también reduciría errores de sintaxis y lógica.

La incorporación de una herramienta como la mencionada, mejoraría la accesibilidad y usabilidad de la aplicación, haciendo que sea más amigable para usuarios menos experimentados. Al ofrecer asistencia en tiempo real, también fomentaría el

aprendizaje y la adopción de buenas prácticas de programación, lo que contribuiría a la creación de soluciones más robustas, eficientes y más rápidas de hacer.

9.2.6 Implementación de la Aplicación para Funcionamiento Global

Actualmente, el sistema está diseñado para operar en una red LAN local, lo que limita su alcance y aplicabilidad. Sin embargo, permitir que la aplicación funcione a través de internet ampliaría drásticamente su utilidad, haciéndola accesible desde cualquier parte del mundo. Para lograr esto, es necesario resolver los problemas asociados con la configuración de servidores STUN y TURN, que son esenciales para facilitar la conectividad entre nodos a través de redes NAT y firewalls.

Implementar esta funcionalidad permitiría a la aplicación trascender los entornos locales y ser utilizada en escenarios más amplios, como proyectos distribuidos globalmente o ambientes de investigación colaborativa. Esto también le abriría las puertas para darse a conocer más rápidamente en el entorno del desarrollo del software, posibilitando su mejora y crecimiento.

9.2.7 Mejora del Poder Computacional de los Nodos

Finalmente, otra de las limitaciones actuales del sistema que podemos mencionar, y que sería interesante su profundización, es su dependencia a una única instancia de Python por nodo, lo que restringe el uso de los recursos computacionales disponibles. Aumentar el poder computacional de los nodos mediante la ejecución de múltiples hilos, con varias instancias de ejecución de Python por nodo, permitiría mejorar significativamente el rendimiento del sistema.

Esta mejora sería especialmente relevante para trabajos que requieren un procesamiento intensivo, donde la distribución efectiva de la carga de trabajo entre los recursos disponibles es clave para lograr tiempos de respuesta más rápidos y un uso más eficiente de la infraestructura accesible. Además, permitiría manejar volúmenes de datos más grandes y ejecutar tareas más complejas sin degradar el rendimiento del sistema en el que se ejecuta.

9.2.8 Optimizar el algoritmo de distribución de claves

El sistema actual no implementa un algoritmo de distribución de claves que sea óptimo, ya que el nodo Master simplemente recopila todas las claves generadas y las distribuye equitativamente entre los nodos Slave sin considerar las que ya posee cada uno. Este enfoque, aunque sencillo, puede generar ineficiencias significativas, especialmente cuando los datos se distribuyen de manera no uniforme, lo que aumenta los costos de transferencia y procesamiento.

Una mejora sustancial sería implementar un algoritmo de distribución de claves que tenga en cuenta la localización de las claves en los nodos. Este algoritmo analizaría las claves que ya se encuentran almacenadas en cada nodo y asignaría nuevas claves de manera que minimice la necesidad de transferencias de datos entre nodos. Por ejemplo, si un nodo Slave ya posee una parte significativa del total de una determinada clave, el algoritmo debería asignarle tal clave a ese nodo, reduciendo así el tráfico de red y optimizando el tiempo de procesamiento.

Además de reducir los costos de transferencia, este enfoque puede mejorar la escalabilidad del sistema, permitiendo un manejo más eficiente de grandes volúmenes de datos a medida que la red crece.

9.2.9 Conclusión

Con estas mejoras y futuras líneas de investigación, el sistema podría evolucionar hacia un sistema más adaptable, potente y accesible, capaz de enfrentar desafíos más complejos y expandir su aplicabilidad en diferentes ámbitos, tanto académicos como en la industria.

Referencias bibliográficas

Ahmed, A. (2024, March 24). *Big Data Analytics in the Entertainment Industry: Audience Behavior Analysis, Content Recommendation, and Revenue Maximization*. ResearchGate. Retrieved August 27, 2024, from https://www.researchgate.net/publication/379154685_Big_Data_Analytics_in_the_Entertainment_Industry_Audience_Behavior_Analysis_Content_Recommendation_and_Revenue_Maximization

Bappalige, S. P. (2014, Agosto 26). *An introduction to Apache Hadoop for big data*. Opensource.com. Retrieved June 25, 2024, from https://opensource.com/life/14/8/intro-apache-hadoop-big-data

Bazán, P. (2017). *Aplicaciones, servicios y procesos distribuidos : una visión para la construcción del software* (1ra ed.). Edulp. https://sedici.unlp.edu.ar/bitstream/handle/10915/62354/Documento_completo.pdf-PDFA.pdf?sequence=1

Brython Documentation. (n.d.). *Brython Documentation*. Brython. Retrieved July 18, 2024, from https://brython.info/index.html

Coulouris, G. F., Dollimore, J., Kindberg, T., & Blair, G. (2012). *Distributed Systems: Concepts and Design*. Addison-Wesley. https://www.amazon.com/Distributed-Systems-Concepts-Design-5th/dp/0132143011

Dean, J., & Ghemawat, S. (2004). *MapReduce: Simplified Data Processing on Large Clusters*. Google Research. Retrieved June 24, 2024, from https://static.googleusercontent.com/media/research.google.com/es//archive/mapreduce-osdi04.pdf

Doug Laney, D. (2001). *3D Data Management: Controlling Data Volume, Velocity and Variety*. META Group.

Dutton, S. (2013, November 4). *Build the backend services needed for a WebRTC app | Articles*. web.dev. Retrieved August 18, 2024, from https://web.dev/articles/webrtc-infrastructure

Fernandez, O. (2022, Diciembre 29). *¿Qué es Hadoop MapReduce? Introducción*. Aprender BIG DATA. Retrieved Junio 18, 2024, from https://aprenderbigdata.com/hadoop-mapreduce/

Fette, I., & Melnikov, A. (2011, Diciembre). *RFC 6455 - The WebSocket Protocol*. IETF Datatracker. Retrieved July 2, 2024, from https://datatracker.ietf.org/doc/html/rfc6455

George, L. (2011). *HBase: The Definitive Guide*. O'Reilly Media, Incorporated. https://www.oreilly.com/library/view/hbase-the-definitive/9781449314682/

Introducción: Clústeres. (n.d.). IBM. Retrieved June 18, 2024, from https://www.ibm.com/docs/es/was-zos/9.0.5?topic=servers-introduction-clusters

Loreto, S., & Romano, S. P. (2014). *Real-time Communication with WebRTC*. O'Reilly.

Mashruwala, A. (2024, June 9). *(PDF) Fraud Detection and Prevention in Financial Services Using Big Data Analytics*. ResearchGate. Retrieved August 27, 2024, from https://www.researchgate.net/publication/381263670_Fraud_Detection_and_Prevention_in_Financial_Services_Using_Big_Data_Analytics

Narkhede, N., Shapira, G., & Palino, T. (2017). *Kafka: The Definitive Guide : Real-time Data and Stream Processing at Scale*. O'Reilly Media. https://www.oreilly.com/library/view/kafka-the-definitive/9781491936153/

Olston, C., Reed, B., Srivastava, U., Kumar, R., & Tomkins, A. (2008). Pig Latin: A Not-So-Foreign Language for Data Processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data* (pp. 1099-1110).

Petroc, T. (2023, Noviembre 16). Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2025. *Statista*. https://www.statista.com/statistics/871513/worldwide-data-created/

Ristevski, B., & Chen, M. (2018, 05 10). Big Data Analytics in Medicine and Healthcare. *Journal of Integrative Bioinformatics*. https://doi.org/10.1515/jib-2017-0030

Sadalage, P. J., & Fowler, M. (2012). *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley.

Skulpt Documentation. (2015). *Skulpt Documentation*. Welcome! - Skulpt.org Homepage. Retrieved July 18, 2024, from https://skulpt.org/

Steen, M. R. v. (2017). *Distributed Systems*. Amazon Fulfillment Poland Sp. z o.o. https://www.distributed-systems.net/index.php/books/ds3/

The Pyodide development team. (2021, agosto). *pyodide/pyodide*. Pyodide — Version 0.26.1. Retrieved July 18, 2024, from https://pyodide.org/en/stable/

Thusoo, A., Sarma, J. S., Jain, N., Shao, Z., Chakka, P., Zhang, N., Antony, S., Liu, H., & Murthy, R. (2010). Hive – A Petabyte Scale Data Warehouse Using

Hadoop. In IEEE (Ed.), *2010 IEEE 26th international conference on data engineering (ICDE 2010)* (pp. 996-1005).

Transcrypt Documentation. (2014). *Transcrypt Documentation*. Transcrypt - Python in the browser - Lean, fast, open! Retrieved July 18, 2024, from https://www.transcrypt.org/home

Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., & Seth, S. (2013). Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing* (pp. 1-16).

WebSockets - Referencia de la API Web | MDN. (2023, February 21). MDN Web Docs. Retrieved June 18, 2024, from https://developer.mozilla.org/es/docs/Web/API/WebSockets_API

White, T. (2015). *Hadoop: The Definitive Guide*. O'Reilly.

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M. J., Shenker, S., & Stoica, I. (2012). Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *9th USENIX symposium on networked systems design and implementation (NSDI 12)* (pp. 15-28). https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf

Anexos

Anexo 1: Instalación y ejecución local de la aplicación

En este anexo se detalla el proceso de instalación y ejecución local de la aplicación, proporcionando una guía paso a paso que permite configurar correctamente el entorno de desarrollo y asegurar el funcionamiento de todos los componentes del sistema. La finalidad es garantizar que los usuarios o desarrolladores interesados en desplegar la aplicación puedan seguir las indicaciones de manera clara y precisa. Este documento también aborda la configuración de las variables de entorno necesarias para el despliegue exitoso de la aplicación, ofreciendo detalles técnicos sobre la personalización y adaptación del entorno de trabajo según las necesidades específicas.

El siguiente instructivo está dirigido a aquellos con conocimientos básicos en el manejo de Docker y Docker Compose, y proporciona la información necesaria para poner en marcha la aplicación en un entorno local.

Link del repositorio:

https://github.com/DavidScoffield/domex

1. **Pre-requisitos**

   Antes de proceder con la instalación y ejecución local de la aplicación, es necesario contar con las siguientes herramientas instaladas:

   - **Docker**: Herramienta de contenedorización que facilita la creación y gestión de contenedores. Es esencial para el despliegue de la aplicación en un entorno controlado y aislado. Guía de instalación[11].

   - **Docker Compose**: Utilidad para definir y manejar aplicaciones de Docker compuestas por múltiples contenedores, lo cual es crucial para la correcta orquestación de los servicios que componen la aplicación. Guía de instalación[12].

   - **Git (opcional)**: Sistema de control de versiones utilizado para clonar el repositorio de la aplicación. Alternativamente, se puede optar por descargar el código fuente en formato comprimido directamente desde el repositorio de GitHub. Instalación[13].

[11] Guía oficial Docker: https://docs.docker.com/get-docker/
[12] Guía oficial Docker Compose: https://docs.docker.com/compose/install/
[13] Instalación: https://git-scm.com/

2. Guía de Instalación

1. **Obtener el código fuente**:

   - **Opción 1: Clonar el repositorio con Git**

     Si se cuenta con Git, se puede clonar el repositorio con el siguiente comando:

     ```
     git clone https://github.com/DavidScoffield/domex
     ```

   - **Opción 2: Descargar el código en formato comprimido desde GitHub**

     Alternativamente, se puede descargar el código en formato comprimido desde GitHub. Para ello, acceda al repositorio, seleccione la opción de descargar el archivo ZIP y descomprímalo en el directorio deseado.

2. **Navegar al directorio del proyecto**:

   Una vez descargado o clonado el repositorio, diríjase al directorio raíz del proyecto:

   ```
   cd domex
   ```

3. **Configurar las variables de entorno**: Es imprescindible configurar las variables de entorno requeridas por la aplicación. Para ello, consulte la sección de **Configuración** más adelante en este anexo.

4. **Construir y desplegar los contenedores de Docker**: Para levantar los servicios de la aplicación, ejecute el siguiente comando:

   ```
   docker-compose up -d
   ```

5. **Acceder a la aplicación**: Una vez que los contenedores estén en funcionamiento, se puede acceder a la aplicación a través de un navegador web. Ingrese a la siguiente dirección:

   ```
   https://<PRIVATE_IP>:<EXTERNAL_HTTPS_PORT>
   ```

Reemplazando *<PRIVATE_IP>* y *<EXTERNAL_HTTP_PORT>* con los valores correspondientes configurados en su entorno.

6. **Detener y eliminar los contenedores de Docker**: Si desea detener la aplicación y eliminar los contenedores, ejecute:

```
docker-compose down
```

Nota: Si ha realizado cambios en el archivo ***'.env'*** o en el código fuente de la aplicación, es necesario reconstruir los contenedores utilizando el siguiente comando:

```
docker-compose up -d --build
```

3. Configuración

La aplicación permite la configuración de varios parámetros a través de variables de entorno definidas en el archivo ***'.env'***. A continuación, se muestra un ejemplo del contenido de dicho archivo y se detallan los parámetros más relevantes:

```
# .env
PRIVATE_IP=192.168.0.32

# Internal
FRONT_INTERNAL_PORT=3000
BACK_INTERNAL_PORT=5000
# External
EXTERNAL_HTTP_PORT=80
EXTERNAL_HTTPS_PORT=443

# App
NEXT_PUBLIC_ICESERVER=local # local, metered
NEXT_PUBLIC_SERVER_URL=https://${PRIVATE_IP}:${EXTERNAL_HTTPS
_PORT}/backend

# Backend
CLUSTER_IDS_LENGTH=4
```

Explicación de las variables

PRIVATE_IP

Este parámetro es obligatorio y debe ser configurado correctamente. Representa la dirección IP privada de la máquina host, la cual iniciará la aplicación, necesaria para la correcta operación de los servicios de la aplicación.

Para obtener la dirección IP privada, se puede ejecutar el siguiente comando en la raíz del proyecto (requiere ***npm***):

```
cd frontend/ && npm run getIPWifi && cd ..
```

En caso de que el comando anterior no funcione, también puede obtener la IP privada utilizando las siguientes herramientas según su sistema operativo:

- **Windows**: `ipconfig`
- **Linux**: `ifconfig`
- **MacOS**: `ifconfig`

FRONT_INTERNAL_PORT y BACK_INTERNAL_PORT

No es necesario modificar estos valores. Definen los puertos internos de los contenedores Docker donde se ejecutan la aplicación web y el servidor backend, respectivamente.

EXTERNAL_HTTP_PORT

No es necesario modificar este valor. Es el puerto externo del contenedor donde se expone la aplicación web. La aplicación redirige automáticamente al puerto HTTPS definido en *EXTERNAL_HTTPS_PORT*.

EXTERNAL_HTTPS_PORT

Es el puerto externo del contenedor donde se expone la aplicación con soporte HTTPS. Se debe acceder a la aplicación a través de este puerto.

NEXT_PUBLIC_ICESERVER

Este parámetro configura el servidor de señalización WebRTC. Los valores disponibles son ***"local"*** (por defecto) y ***"metered"***, siendo este último el que utiliza un servidor de señalización externo. Este último es un servidor con un límite de uso, por lo que para el uso de esta tesina, el parámetro a utilizar es ***"local".***

NEXT_PUBLIC_SERVER_URL

No se recomienda modificar este valor. Define la URL del servidor backend con el que se conecta la aplicación web.

CLUSTER_IDS_LENGTH

Define la longitud de los identificadores únicos de los clústers. El valor predeterminado es ***"5"***.

Este anexo proporciona las bases para una instalación exitosa de la aplicación en un entorno local, con la flexibilidad suficiente para adaptarse a diferentes configuraciones de red y requisitos del usuario.

Anexo 2: Dockerización de la aplicación

Este anexo ofrece una descripción detallada de la dockerización de la aplicación, abordando cada componente de la arquitectura y cómo se configuran mediante Dockerfiles y el archivo `docker-compose.yml`. La dockerización facilita el empaquetado, despliegue y gestión de aplicaciones en contenedores aislados, garantizando consistencia en diferentes entornos.

1. Configuración General de Docker Compose

El archivo docker-compose.yml define cuatro servicios principales: cert-generator, next-app, backend, y nginx. Estos servicios están configurados para integrarse en una red común denominada domex, lo que permite la comunicación fluida entre ellos.

Archivo `docker-compose.yml`:

```
version: '3'

services:
  cert-generator:
    build:
      context: ./certs
      dockerfile: certs.Dockerfile
    volumes:
      - ./certs/ssl:/certs
      - ./certs/scripts/generate_cert.sh:/generate_cert.sh
    environment:
      SERVER_KEY_NAME: server.key
      SERVER_CERT_NAME: server.crt
    command: sh -c "./generate_cert.sh"
    networks:
      - domex

  next-app:
    build:
      context: ./frontend
      dockerfile: prod.Dockerfile
      args:
        NEXT_PUBLIC_ICESERVER: ${NEXT_PUBLIC_ICESERVER:-local}
        NEXT_PUBLIC_SERVER_URL:
${NEXT_PUBLIC_SERVER_URL:-https://192.168.2.5/backend
        NODE_TLS_REJECT_UNAUTHORIZED: '0'
    restart: always
    environment:
      PORT: ${FRONT_INTERNAL_PORT:-3000}
```

```yaml
    networks:
      - domex

  backend:
    build:
      context: ./backend
      dockerfile: prod.Dockerfile
    restart: always
    environment:
      NODE_ENV: production
      PORT: ${BACK_INTERNAL_PORT:-5000}
      HOST: 0.0.0.0
      CLUSTER_IDS_LENGTH: ${CLUSTER_IDS_LENGTH:-5}
    networks:
      - domex

  nginx:
    image: nginx:alpine
    restart: always
    ports:
      - ${EXTERNAL_HTTP_PORT:-80}:80
      - ${EXTERNAL_HTTPS_PORT:-443}:443
    volumes:
      - ./nginx/templates:/etc/nginx/templates
      - ./certs/ssl:/etc/nginx/certs:ro
    environment:
      NGINX_ENVSUBST_OUTPUT_DIR: '/etc/nginx'
      FRONT_INTERNAL_PORT: ${FRONT_INTERNAL_PORT:-3000}
      BACK_INTERNAL_PORT: ${BACK_INTERNAL_PORT:-5000}
      EXTERNAL_HTTP_PORT: ${EXTERNAL_HTTP_PORT:-80}
      EXTERNAL_HTTPS_PORT: ${EXTERNAL_HTTPS_PORT:-443}
    networks:
      - domex
    depends_on:
      - cert-generator
      - next-app
      - backend

networks:
  domex:
```

2. Descripción de Servicios

a. Servicio “*cert-generator*”

El servicio *cert-generator* es responsable de la generación de certificados SSL autofirmados necesarios para establecer conexiones seguras entre los

servicios. Este servicio utiliza el Dockerfile *certs.Dockerfile* y un script de generación de certificados.

- **Archivo `certs.Dockerfile`**

```
FROM alpine:3.19

RUN apk --no-cache add openssl
```

Este Dockerfile utiliza una imagen base de Alpine Linux e instala `openssl`, una herramienta esencial para la generación de certificados SSL.

- **Script `generate_cert.sh`**

El script *generate_cert.sh* se encarga de crear los certificados SSL necesarios para la seguridad de la comunicación. Aquí se detalla su funcionamiento:

```
#!/bin/sh

# Obtener los nombres de los archivos de las variables de entorno
SERVER_KEY_NAME=${SERVER_KEY_NAME:-"server.key"}
SERVER_CERT_NAME=${SERVER_CERT_NAME:-"server.crt"}

echo "Generando certificados SSL autofirmados..."

# Verificar si /certs es un directorio
if [ ! -d /certs ]; then
  # Si /certs no es un directorio, intentar crearlo
  mkdir -p /certs
fi

# Generar el certificado SSL autofirmado
openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout
/certs/$SERVER_KEY_NAME -out /certs/$SERVER_CERT_NAME -subj
"/CN=localhost"

if [ -f /certs/$SERVER_KEY_NAME ] && [ -f
/certs/$SERVER_CERT_NAME ]; then
  echo "¡Certificados SSL autofirmados generados y movidos
correctamente!"
else
  echo "Error al generar los certificados SSL autofirmados"
  exit 1
fi
```

Explicación del script

- **Definición de Nombres de Archivos:** Obtiene los nombres de los archivos de los certificados desde las variables de entorno *SERVER_KEY_NAME* y *SERVER_CERT_NAME*, asignando valores predeterminados si no se proporcionan.
- **Creación del Directorio:** Verifica si el directorio */certs* existe; si no es así, lo crea.
- **Generación de Certificados:** Usa *openssl* para generar un certificado SSL autofirmado. El comando especifica un certificado válido por 365 días y una nueva clave RSA de 2048 bits.
- **Verificación y Mensajes:** Verifica si los certificados se han creado correctamente y proporciona mensajes de éxito o error.

b. Servicio *"next-app"*

El servicio *next-app* se encarga de ejecutar la aplicación frontend basada en Next.js. Esta aplicación se construye utilizando un Dockerfile específico para producción, configurado para optimizar el rendimiento y la seguridad en un entorno de contenedores.

Archivo '*Dockerfile*' para '*next-app*' ('frontend/prod.Dockerfile')

```
FROM node:18-alpine AS base

# Step 1. Rebuild the source code only when needed
FROM base AS builder

WORKDIR /app

# Install dependencies based on the preferred package manager
COPY package.json yarn.lock* package-lock.json* pnpm-lock.yaml* ./
# Omit --production flag for TypeScript devDependencies
RUN \
  if [ -f yarn.lock ]; then yarn --frozen-lockfile; \
  elif [ -f package-lock.json ]; then npm ci; \
  elif [ -f pnpm-lock.yaml ]; then corepack enable pnpm && pnpm i; \
  # Allow install without lockfile, so example works even without Node.js
installed locally
  else echo "Warning: Lockfile not found. It is recommended to commit
lockfiles to version control." && yarn install; \
  fi

COPY src ./src
```

```
COPY public ./public
COPY next.config.js .
COPY tsconfig.json .
COPY .eslintrc.json .
COPY .prettierrc.json .
COPY tailwind.config.ts postcss.config.js ./

# Environment variables must be present at build time
# https://github.com/vercel/next.js/discussions/14030
ARG NEXT_PUBLIC_ICESERVER
ENV NEXT_PUBLIC_ICESERVER=${NEXT_PUBLIC_ICESERVER}
ARG NEXT_PUBLIC_SERVER_URL
ENV NEXT_PUBLIC_SERVER_URL=${NEXT_PUBLIC_SERVER_URL}
# Set NODE_TLS_REJECT_UNAUTHORIZED to 0 to allow self-signed certificates
in production (only for local network testing, not recommended for
production use)
ARG NODE_TLS_REJECT_UNAUTHORIZED
ENV NODE_TLS_REJECT_UNAUTHORIZED=${NODE_TLS_REJECT_UNAUTHORIZED}

# Next.js collects completely anonymous telemetry data about general
usage. Learn more here: https://nextjs.org/telemetry
# Uncomment the following line to disable telemetry at build time
# ENV NEXT_TELEMETRY_DISABLED 1

# Build Next.js based on the preferred package manager
RUN \
  if [ -f yarn.lock ]; then yarn build; \
  elif [ -f package-lock.json ]; then npm run build; \
  elif [ -f pnpm-lock.yaml ]; then pnpm build; \
  else npm run build; \
  fi

# Note: It is not necessary to add an intermediate step that does a full
copy of `node_modules` here

# Step 2. Production image, copy all the files and run next
FROM base AS runner

WORKDIR /app

# Don't run production as root
RUN addgroup --system --gid 1001 nodejs
RUN adduser --system --uid 1001 nextjs
USER nextjs


COPY --from=builder /app/public ./public

# Automatically leverage output traces to reduce image size
# https://nextjs.org/docs/advanced-features/output-file-tracing
```

```
COPY --from=builder --chown=nextjs:nodejs /app/.next/standalone ./
COPY --from=builder --chown=nextjs:nodejs /app/.next/static
./.next/static

# Environment variables must be redefined at run time
ARG NEXT_PUBLIC_ICESERVER
ENV NEXT_PUBLIC_ICESERVER=${NEXT_PUBLIC_ICESERVER}
ARG NEXT_PUBLIC_SERVER_URL
ENV NEXT_PUBLIC_SERVER_URL=${NEXT_PUBLIC_SERVER_URL}
# Set NODE_TLS_REJECT_UNAUTHORIZED to 0 to allow self-signed certificates
in production (only for local network testing, not recommended for
production use)
ARG NODE_TLS_REJECT_UNAUTHORIZED
ENV NODE_TLS_REJECT_UNAUTHORIZED=${NODE_TLS_REJECT_UNAUTHORIZED}

# Uncomment the following line to disable telemetry at run time
ENV NEXT_TELEMETRY_DISABLED 1

# Note: Don't expose ports here, Compose will handle that for us

CMD ["node", "server.js"]
```

Explicación del Dockerfile

1. **Base Stage**

   - Utiliza la imagen *'node:18-alpine'* como base, que es una versión ligera y segura de Node.js sobre Alpine Linux.

2. **Builder Stage**

   - **Directorio de Trabajo:** Establece el directorio de trabajo en '*/app*'.

   - **Instalación de Dependencias:** Copia los archivos de configuración de paquetes ('package.json', 'yarn.lock', 'package-lock.json', 'pnpm-lock.yaml') y ejecuta el gestor de paquetes correspondiente ('yarn', 'npm', 'pnpm') para instalar las dependencias.

   - **Copiado del Código Fuente:** Copia el código fuente de la aplicación y los archivos de configuración relevantes.

   - **Variables de Entorno:** Configura las variables de entorno necesarias para la construcción de la aplicación ('*NEXT_PUBLIC_ICESERVER*', '*NEXT_PUBLIC_SERVER_URL*', '*NODE_TLS_REJECT_UNAUTHORIZED*').

- **Construcción de la Aplicación:** Ejecuta el comando de construcción ('yarn build', 'npm run build', 'pnpm build') para preparar la aplicación para producción.

3. **Runner Stage**

- **Configuración del Usuario:** Crea un usuario y un grupo no root ('nextjs' y 'nodejs') para ejecutar la aplicación, mejorando la seguridad.

- **Copiado de Archivos de Construcción:** Copia los archivos de construcción desde la etapa anterior ('public', '.next/standalone', '.next/static'), asignando los permisos adecuados al usuario 'nextjs'.

- **Variables de Entorno:** Reconfigura las variables de entorno necesarias para la ejecución de la aplicación.

- **Deshabilitación de Telemetría:** Desactiva la telemetría de Next.js ('*NEXT_TELEMETRY_DISABLED 1*').

- **Inicio de la Aplicación:** El comando '*CMD ["node", "server.js"]*' inicia la aplicación Next.js.

Configuración en *'docker-compose.yml'*

```
next-app:
    build:
      context: ./frontend
      dockerfile: prod.Dockerfile
      args:
        NEXT_PUBLIC_ICESERVER: ${NEXT_PUBLIC_ICESERVER:-local}
        NEXT_PUBLIC_SERVER_URL:
${NEXT_PUBLIC_SERVER_URL:-https://192.168.2.5/backend}
        # Set NODE_TLS_REJECT_UNAUTHORIZED to 0 to allow self-signed
certificates in production (only for local network testing, not
recommended for production use)
        NODE_TLS_REJECT_UNAUTHORIZED: '0'
    restart: always
    environment:
      PORT: ${FRONT_INTERNAL_PORT:-3000}
    networks:
      - domex
```

Explicación de la Configuración en *'docker-compose.yml'*

- **Build Context:** Define el contexto de construcción y el Dockerfile ('prod.Dockerfile') que se utilizará para construir la imagen de 'next-app'.
- **Argumentos de Construcción:** Pasa variables de entorno al Dockerfile (*'NEXT_PUBLIC_ICESERVER'*, *'NEXT_PUBLIC_SERVER_URL'*, *'NODE_TLS_REJECT_UNAUTHORIZED'*).
- **Restart Policy:** Configura el contenedor para reiniciarse automáticamente en caso de fallo (*'restart: always'*).
- **Environment Variables:** Establece el puerto interno en el que la aplicación Next.js escuchará (*'PORT'*).
- **Red:** Conecta el contenedor a la red *'domex'* para permitir la comunicación con otros servicios.

c. Servicio *"backend"*

El servicio 'backend' es responsable de ejecutar la aplicación del lado del servidor. Este servicio está diseñado para ejecutarse en un entorno de producción con Node.js y utiliza un Dockerfile específico para construir y ejecutar la aplicación de manera eficiente.

Archivo *'Dockerfile'* para *'backend'* ('backend/prod.Dockerfile'):

```
FROM node:18.17-alpine AS base

# Step 1. Rebuild the source code only when needed
FROM base AS builder

WORKDIR /app

# Install dependencies based on the preferred package manager
COPY package.json yarn.lock* package-lock.json* pnpm-lock.yaml* ./
# Omit --production flag for TypeScript devDependencies
RUN \
  if [ -f yarn.lock ]; then yarn --frozen-lockfile; \
  elif [ -f package-lock.json ]; then npm ci; \
  elif [ -f pnpm-lock.yaml ]; then corepack enable pnpm && pnpm i; \
  # Allow install without lockfile, so example works even without Node.js
installed locally
  else echo "Warning: Lockfile not found. It is recommended to commit
```

```
lockfiles to version control." && yarn install; \
  fi

# Copy the source code
COPY src ./src

# Copy the configuration files
COPY tsconfig.json ./
COPY .prettierrc.json .
COPY .eslintrc.cjs .

# Build the application
RUN npm run compile

# Step 2. Production image, copy all the files and run the application
FROM base AS runner

WORKDIR /app

# Don't run production as root
RUN addgroup --system --gid 1001 nodejs
RUN adduser --system --uid 1001 express
# Give the user the permission to write to the /app directory
RUN chown -R express:nodejs .
USER express

#  Copy the files from the builder
COPY --from=builder /app/dist ./dist
COPY --from=builder /app/package.json ./
COPY --from=builder /app/node_modules ./node_modules

# Start the application
CMD ["npm", "start"]
```

Explicación del Dockerfile

1. **Base Stage**

   - **Imagen Base:** Utiliza *'node:18.17-alpine'*, una versión ligera de Node.js basada en Alpine Linux, que proporciona un entorno eficiente para la ejecución de la aplicación.

2. **Builder Stage**

   - **Directorio de Trabajo:** Establece el directorio de trabajo en '*/app'* para todas las operaciones subsiguientes.
   - **Instalación de Dependencias:** Copia los archivos de configuración de paquetes ('package.json', 'yarn.lock',

'package-lock.json', 'pnpm-lock.yaml') y ejecuta el gestor de paquetes adecuado ('yarn', 'npm', 'pnpm') para instalar las dependencias del proyecto. La elección del gestor se basa en los archivos presentes.

- **Copiado del Código Fuente y Configuración:** Copia el código fuente de la aplicación y los archivos de configuración necesarios ('tsconfig.json', '.prettierrc.json', '.eslintrc.cjs').
- **Construcción de la Aplicación:** Ejecuta el comando *'npm run compile'* para compilar el código TypeScript en JavaScript, generando los archivos de distribución necesarios.

3. **Runner Stage**

- **Configuración del Usuario:** Crea un usuario y grupo no root ('express' y 'nodejs') para mejorar la seguridad. Le da permisos de escritura al ‘express’ sobre el directorio ‘/app’.
- **Copiado de Archivos de Construcción:** Copia los archivos necesarios desde la etapa de construcción, builder stage, ('dist', 'package.json', 'node_modules') al contenedor de ejecución.
- **Inicio de la Aplicación:** El comando *'CMD ["npm", "start"]'* se utiliza para iniciar la aplicación en modo producción.

Configuración en *'docker-compose.yml'*

```
backend:
    build:
      context: ./backend
      dockerfile: prod.Dockerfile
    restart: always
    environment:
      NODE_ENV: production
      PORT: ${BACK_INTERNAL_PORT:-5000}
      HOST: 0.0.0.0
      CLUSTER_IDS_LENGTH: ${CLUSTER_IDS_LENGTH:-5}
    networks:
      - domex
```

Explicación de la Configuración en *'docker-compose.yml'*

- **Build Context:** Define el contexto de construcción ('./backend') y el Dockerfile ('prod.Dockerfile') que se utilizará para construir la imagen del servicio 'backend'.
- **Restart Policy:** Configura el contenedor para reiniciarse automáticamente en caso de fallo (*'restart: always'*), lo que garantiza la disponibilidad continua del servicio.
- **Variables de Entorno:**
    - *'NODE_ENV':* Establece el entorno de ejecución como 'production', optimizando el rendimiento y la seguridad.
    - *'PORT':* Define el puerto interno en el que el backend escuchará ('5000' por defecto).
    - *'HOST':* Establece el host en '0.0.0.0' para que el servicio esté disponible en todas las interfaces de red del contenedor.
    - *'CLUSTER_IDS_LENGTH':* Configura la longitud de los identificadores del cluster, utilizando el valor '5' por defecto si no se especifica otro.
- **Red:** Conecta el contenedor a la red *'domex'*, permitiendo la comunicación con otros servicios dentro de la misma red Docker.

d. Servicio *"nginx"*

El servicio *'nginx'* actúa como un servidor proxy inverso. Está configurado para manejar tanto el tráfico HTTP como HTTPS, proporcionando una capa adicional de seguridad y optimización al enrutar las solicitudes a los servicios backend adecuados.

Configuración en *'docker-compose.yml'*

```
nginx:
   image: nginx:alpine
   restart: always
   ports:
     - ${EXTERNAL_HTTP_PORT:-80}:80
     - ${EXTERNAL_HTTPS_PORT:-443}:443
   volumes:
     - ./nginx/templates:/etc/nginx/templates
```

```
      - ./certs/ssl:/etc/nginx/certs:ro
    environment:
      NGINX_ENVSUBST_OUTPUT_DIR: '/etc/nginx'
      FRONT_INTERNAL_PORT: ${FRONT_INTERNAL_PORT:-3000}
      BACK_INTERNAL_PORT: ${BACK_INTERNAL_PORT:-5000}
      EXTERNAL_HTTP_PORT: ${EXTERNAL_HTTP_PORT:-80}
      EXTERNAL_HTTPS_PORT: ${EXTERNAL_HTTPS_PORT:-443}
    networks:
      - domex
    depends_on:
      - cert-generator
      - next-app
      - backend
```

Explicación de la Configuración en *'docker-compose.yml'*

- **Imagen:** Utiliza la imagen oficial *'nginx:alpine'*, una versión ligera de NGINX basada en Alpine Linux, que es eficiente en términos de tamaño y rendimiento.
- **Restart Policy:** Configura el contenedor para reiniciarse automáticamente en caso de fallo (*'restart: always'*), asegurando la alta disponibilidad del servidor NGINX.
- **Ports:** Expone los puertos HTTP (**80**) y HTTPS (**443**) del contenedor al host, mapeando estos puertos a los valores especificados en las variables de entorno (*'EXTERNAL_HTTP_PORT'* y *'EXTERNAL_HTTPS_PORT'*).
- **Volumes:**
  - *'./nginx/templates:/etc/nginx/templates'*: Monta el directorio de plantillas NGINX desde el host al contenedor, permitiendo la personalización de la configuración de NGINX.
  - *'./certs/ssl:/etc/nginx/certs:ro'*: Monta el directorio que contiene los certificados SSL necesarios para la terminación HTTPS en el contenedor, en modo solo lectura (*'ro'*).
- **Environment Variables:**
  - *'NGINX_ENVSUBST_OUTPUT_DIR'*: Define el directorio de salida para la sustitución de variables en los archivos de configuración de NGINX.

  - *'FRONT_INTERNAL_PORT'*: El puerto interno en el que la aplicación frontend (Next.js) escucha dentro del contenedor ('3000' por defecto).
  - *'BACK_INTERNAL_PORT'*: El puerto interno en el que la aplicación backend escucha dentro del contenedor ('5000' por defecto).
  - *'EXTERNAL_HTTP_PORT'*: El puerto externo para tráfico HTTP ('80' por defecto).
  - *'EXTERNAL_HTTPS_PORT'*: El puerto externo para tráfico HTTPS ('443' por defecto).
- **Networks:** Conecta el contenedor a la red *'domex'*, permitiendo la comunicación con otros servicios como *'cert-generator'*, *'next-app'*, y *'backend'*.
- **Depends On:** Asegura que los servicios *'cert-generator'*, *'next-app'*, y *'backend'* se inicien antes de *'nginx'*, garantizando que todos los servicios necesarios estén disponibles cuando NGINX arranque.

Configuración del Dockerfile de NGINX:

En este caso, el servicio 'nginx' utiliza la imagen oficial de NGINX, por lo que no se proporciona un Dockerfile personalizado. La configuración se maneja a través de archivos de configuración montados en el contenedor.

Sin embargo, el contenedor de NGINX es configurado a traves de un archivo de configuración principal que se encuentra en *'./nginx/templates'*. En el **Anexo 3: Configuración del Servidor NGINX** se entrará más en detalle de las configuraciones específicas del mismo.

Anexo 3: Configuración del Servidor NGINX

El archivo *'nginx.conf'* que se encuentra en *'./nginx/templates/nginx.conf'* define las configuraciones esenciales del servidor NGINX utilizado en la arquitectura de la aplicación. A continuación, se detalla cada sección clave del archivo:

1. Bloque 'events { }'

```
events {
}
```

Este bloque está vacío en la configuración actual, pero típicamente se utiliza para configurar parámetros relacionados con la gestión de eventos en NGINX, como el número de conexiones máximas por worker.

2. Bloque 'http { }'

El bloque *'http'* contiene toda la configuración necesaria para manejar solicitudes HTTP y HTTPS.

```
http {
  upstream next-app {
    server next-app:$FRONT_INTERNAL_PORT;
  }

  upstream backend {
    server backend:$BACK_INTERNAL_PORT;
  }
```

Dentro de este bloque, se definen dos upstreams: *'next-app'* y *'backend'*. Estos upstreams especifican cómo NGINX enruta el tráfico a los servicios *'next-app'* y *'backend'*, respectivamente, a través de los puertos internos configurados en las variables de entorno ('*$FRONT_INTERNAL_PORT*' y '*$BACK_INTERNAL_PORT*').

3. Bloque 'server { }' para Redirección de HTTP a HTTPS

El primer bloque *'server'* dentro del bloque *'http'* configura NGINX para redirigir todo el tráfico HTTP (puerto 80) a HTTPS (puerto 443):

```
server {
  listen 80;
  server_name localhost;
  root /srv/public;
  return 301 https://$host:$EXTERNAL_HTTPS_PORT$request_uri;
}
```

- **listen 80:** Instruye a NGINX para escuchar en el puerto 80.
- **server_name localhost:** Define el nombre del servidor como *'localhost'*.
- **root /srv/public:** Especifica la raíz del servidor, que apunta al directorio *'/srv/public'*.
- **return 301:** Redirige todas las solicitudes HTTP a su equivalente HTTPS usando un código de estado *'301 Moved Permanently'*.

4. Bloque 'server { }' para Manejo de Solicitudes HTTPS

El segundo bloque *'server'* dentro del bloque *'http'* configura NGINX para manejar las solicitudes HTTPS:

```
server {
  listen 443 ssl;
  server_name localhost;
  root /srv/public;
  server_tokens off;

  ssl_certificate /etc/nginx/certs/server.crt;
  ssl_certificate_key /etc/nginx/certs/server.key;
```

- **listen 443 ssl:** Configura NGINX para escuchar en el puerto 443 y utilizar SSL.
- **server_tokens off:** Desactiva la visualización de la versión de NGINX en las respuestas HTTP, aumentando la seguridad.
- **ssl_certificate** y **ssl_certificate_key:** Especifican las rutas de los archivos de certificado y clave privada que se utilizarán para las conexiones SSL.

Ruta Principal '/'

```
location / {
  try_files $uri $uri/ @next-app;
}
```

Este bloque intenta servir los archivos estáticos desde la raíz del servidor. Si el archivo no existe, la solicitud se envía al bloque *'@next-app'*.

Ruta para la Aplicación Next.js

```
location @next-app {
  proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
  proxy_set_header X-Forwarded-Proto https;
  proxy_set_header X-Forwarded-Ssl on;
  proxy_set_header Host $http_host;
  proxy_redirect off;
  proxy_pass http://next-app;
  proxy_cookie_path / "/; HTTPOnly; Secure";
}
```

Este bloque redirige las solicitudes a la aplicación Next.js, pasando las cabeceras adecuadas y asegurando que las cookies se marquen como seguras y HTTPOnly.

Ruta para el Backend *'/backend'*

```
location /backend {
  rewrite ^/backend/(.*) /$1 break;
  proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
  proxy_set_header X-Forwarded-Proto https;
  proxy_set_header X-Forwarded-Ssl on;
  proxy_set_header Host $http_host;
  proxy_redirect off;
  proxy_pass http://backend;
  proxy_cookie_path / "/; HTTPOnly; Secure";

  proxy_http_version 1.1;
  proxy_set_header Upgrade $http_upgrade;
  proxy_set_header Connection "upgrade";
}
```

Este bloque gestiona las solicitudes que comienzan con *'/backend'*, eliminando el prefijo y redirigiéndolas al servicio backend. Además, permite la comunicación WebSocket al manejar las cabeceras 'Upgrade' y 'Connection'.

Anexo 4: Código del módulo de Python MapReduceJob

```
import os, json, sys, time, traceback
from functools import wraps
from typing import Literal

def log_error(phase_name: str, error: Exception):
    """Guarda el error en un archivo JSON"""

    error_details = ''.join(traceback.format_exception(type(error), error,
error.__traceback__))
    with open('/stderr.json', 'w') as errors_file:
        json.dump({f"{phase_name}Code": f"[{phase_name}] -> {error_details}"},
errors_file)

def read_code(file_path: str, instance: 'MapReduceJob'):
    """Ejecuta el código de un archivo y retorna si está vacío"""

    with open(file_path) as file:
        code = file.read()
        exec(code, {'write': instance.write}, instance.__dict__)
        return bool(code.strip())

def load_json(file_path: str):
    """Retorna el contenido y tamaño de un archivo JSON"""

    with open(file_path) as file:
        data = json.load(file)

    return data, os.path.getsize(file_path)

def write_keys(dict_to_write: dict, file_name: str):
    """Escribe un diccionario en un archivo JSON, convirtiendo las claves de
tipo tupla a strings, y retorna el tamaño del archivo"""

    with open(file_name, 'w') as file:
        json.dump({str(k) if isinstance(k, tuple) else k: v for k, v in
dict_to_write.items()}, file)

    return os.path.getsize(file_name)

def save_statistics(statistics: dict):
    """Guarda las estadísticas en un archivo JSON"""

    with open('/sizes.json', 'w') as statistics_file:
        json.dump(statistics, statistics_file)

def clean_up():
```

```
    """Elimina archivos temporales"""

    if os.path.exists('/stderr.json'):
        os.remove("/stderr.json")

def safe_execute(phase):
    @wraps(phase)
    def wrapper(self: 'MapReduceJob'):
        try:
            # se obtiene el nombre de la etapa
            phase_name : Literal['map', 'combine', 'reduce'] = phase.__name__

            # se lee el código de la etapa
            self.execute_phase = read_code(f'/{phase_name}_code.py', self)

            # se setea la referencia a los resultados de la etapa actual
            self.current_results = getattr(self, f'{phase_name}_results')

            # se obtiene el tiempo inicial
            start_time = time.perf_counter_ns()

            # se ejecuta la etapa
            result = phase(self)

            elapsed_time = time.perf_counter_ns() - start_time

            output_size = write_keys(self.current_results,
f'/{phase_name}_results.json')

            if self.invocations:
                # se imprime un mensaje de éxito
                print(f"{phase_name.upper()} EJECUTADO SATISFACTORIAMENTE")
                input_size = self.input_size
            else:
                elapsed_time = 0
                input_size = 0
                output_size = 0

            # se guardan las estadísticas de la etapa
            self.statistics.update(
                {
                    # tiempo de ejecución
                    f'{phase_name}Time': elapsed_time,
                    # cantidad de invocaciones
                    f'{phase_name}Count': self.invocations,
                    # tamaño de la entrada
                    f'{phase_name}Input': input_size,
                    # se escriben los resultados de la etapa y se obtiene el
tamaño
                    f'{phase_name}Output': output_size
```

```
                }
            )

            return result
        except Exception as e:
            log_error(phase_name, e)
            sys.exit(1)
    return wrapper

class MapReduceJob:

    MAP_INPUT_FILE = '/input.txt'
    REDUCE_KEYS_FILE = '/reduce_keys.json'

    map_results = {}
    combine_results = {}
    reduce_results = {}
    statistics = {}

    current_results = None
    input_size = 0
    invocations = 0
    execute_phase = True

    def write(self, key, value):
        self.current_results.setdefault(key, []).append(value)

    @safe_execute
    def map(self):
        """Ejecuta la etapa map"""

        with open(self.MAP_INPUT_FILE) as input_file:
            # se leen las lineas del archivo de entrada
            input_lines = input_file.readlines()

        # se aplica "fmap" a cada linea del archivo de entrada
        for line in input_lines:
            # se borra el salto de línea al final de cada linea
            self.fmap(line[:-1])

        # se obtiene el tamaño del archivo de entrada
        self.input_size = os.path.getsize(self.MAP_INPUT_FILE)
        # se obtiene la cantidad de lineas del archivo de entrada
        self.invocations = len(input_lines)

    @safe_execute
    def combine(self):
        """Ejecuta la etapa combine"""

        # si el código combine fue especificado
```

```
        if self.execute_phase:
            # se aplica "fcomb" a cada key y sus valores
            for key, values in self.map_results.items():
                self.fcomb(key, values)
            # se obtiene la cantidad de keys de la etapa map
            self.invocations = len(self.map_results)
        else:
            self.invocations = 0
            self.current_results = self.map_results

    @safe_execute
    def reduce(self):
        """Ejecuta la etapa reduce"""

        # se cargan las claves y valores a reducir
        reduce_keys, size = load_json(self.REDUCE_KEYS_FILE)

        # se aplica "fred" a cada clave y sus valores
        for key, values in reduce_keys.items():
            # se evalua la clave si es una tupla
            key = eval(key) if key.startswith("(") and key.endswith(")") else
key
            self.fred(key, values)

        # se obtiene el tamaño del archivo de claves a reducir
        self.input_size = size
        # se obtiene la cantidad de claves de la etapa reduce
        self.invocations = len(reduce_keys)

    def execute(self):
        """Ejecuta el Job MapReduce"""

        if os.path.exists(self.REDUCE_KEYS_FILE):
            self.reduce()
        else:
            self.map()
            self.combine()

        save_statistics(self.statistics)
        clean_up()

MapReduceJob().execute()
```