Facultad de Ingeniería
Escuela de Ingeniería Informática

DESARROLLO DE UN ENTORNO WEB PARA LA SIMULACIÓN DE PLATAFORMAS DE PROCESAMIENTO DE STREAMS ORIENTADA A SERVICIOS

Por

Bastián Joel Toledo Salas

Trabajo realizado para optar al Título de
INGENIERO (CIVIL) EN INFORMÁTICA
Prof. Guía: Alonso Inostrosa Psijas
Noviembre 2024

Certifico que he leído este documento y que, en mi opinión, es adecuado en ámbito y calidad como trabajo para optar al título de Ingeniero Civil en Informática.

Alonso Inostrosa Psijas Profesor Guía

Certifico que he leído este documento y que, en mi opinión, es adecuado en ámbito y calidad como trabajo para optar al título de Ingeniero Civil en Informática.

Nombre Profesor Informante 1 Profesor Informante

Aprobado por la Escuela de Ingeniería en Informática, UNIVERSIDAD DE VALPARAÍSO.

Resumen

A continuación aspectos que debe considera para escribir un resumen comprensible y que cumpla su propósito.

- Una o dos oraciones que provean una introducción básica al área de trabajo, comprensibles para un interesado de cualquier disciplina.
- Dos o tres oraciones de antecedentes más detallados, comprensibles para personas de disciplinas relacionadas.
- Una oración que indica claramente el problema general que trata en este trabajo de título.
- Una oración que resume el resultado principal.
- Dos o tres oraciones que explique lo que el resultado principal aporta al estado del arte.
- Una o dos oraciones para poner los resultados en un contexto más general.
- (opcional) Dos o tres oraciones para proporcionar una perspectiva más amplia, fácilmente comprensible para una persona de cualquier disciplina.

Recuerde

- El resumen es una consolidación de su trabajo
- Debe ser conciso y describir el alcance de su trabajo, resumir sus resultados y presentar las principales conclusiones.
- Debe tener máximo 250 palabras.
- Debe ser escrito en pasado.

Índice general

Resumen III

1. Introducción 1
- 1.1. Principales Contribuciones . . . 2
- 1.2. Estructura del Documento . . . 2

2. Marco Conceptual y Estado del Arte 3
- 2.1. Marco Conceptual . . . 3
  - 2.1.1. Modelos de Datos en Sistemas de Procesamiento de Streams . . . 3
  - 2.1.2. Métodos de Procesamiento de datos . . . 4
  - 2.1.3. Procesamiento de Streams . . . 5
  - 2.1.4. Plataformas de Procesamiento de Streams . . . 5
  - 2.1.5. Clasificación de Plataformas de Procesamiento de Streams . . . 6
  - 2.1.6. Modelos de Simulación en Sistemas de Procesamiento de Streams . 7
- 2.2. Estado del Arte . . . 8
  - 2.2.1. Principales Plataformas de Procesamiento de Streams . . . 8
  - 2.2.2. Apache Samza . . . 9
  - 2.2.3. Apache Flink . . . 10
  - 2.2.4. Apache Storm . . . 13
  - 2.2.5. Apache Spark Streaming . . . 15
  - 2.2.6. Trabajos Relacionados . . . 19

3. Definición del Problema y Análisis de Requerimientos 22
- 3.1. Contexto del Problema . . . 22
- 3.2. Definición del Problema . . . 23
  - 3.2.1. Complejidad de Configuración y Uso de Aplicación . . . 23
  - 3.2.2. Gestión eficiente del flujo de datos y comunicación entre componentes . . . 24
  - 3.2.3. Portabilidad y Despliegue de Aplicación . . . 24
- 3.3. Solución Propuesta . . . 24

3.4. Objetivos 27
3.4.1. Objetivo General 27
3.4.2. Objetivos Específicos 27
3.5. Metodología 28
3.5.1. Metodología de Desarrollo 28
3.5.2. Planificación de Desarrollo basada en Scrum 30
3.6. Especificación de Requerimientos 31
3.6.1. Requerimientos Funcionales 31
3.6.2. Requerimientos No Funcionales 33
3.7. Funcionalidades del Sistema 35
3.7.1. Diagramas de Casos de Uso 35
3.7.2. Casos de Uso 35
3.7.3. Diagramas de Estado 36
3.7.4. Modelo Conceptual 37

4. Diseño de la solución **38**
4.1. Diseño Arquitectónico 38
4.1.1. Tecnologías utilizadas 38
4.1.2. Flujo de datos / Vista de alto nivel 41
4.2. Diseño Lógico 41
4.2.1. Diagrama de despliegue 41
4.2.2. Diagrama de componentes 42
4.2.3. Diagrama de paquetes 43
4.2.4. Diagrama de clases 43
4.3. Diseño de Datos 45
4.3.1. Diagrama Entidad Relación 45
4.3.2. Modelo Relacional 45
4.3.3. Diccionario de Datos 45
4.4. Diseño de Interfaz 45
4.4.1. Arquitectura de la Información 45
4.4.2. Prototipos de Interfaces Gráficas 45
4.5. Diseño de Pruebas 45
4.5.1. Pruebas Unitarias 45
4.5.2. Pruebas de Integración 45
4.5.3. Pruebas de Sistema 45
4.5.4. Pruebas con Usuarios 45
4.5.5. Pruebas de Aceptación 45
4.6. Diagramas Opcionales / Complementario 45

5. Experimentación **46**
5.1. Diseño de experimentos 46
5.1.1. Análisis Ordinario 46
5.2. Datasets y Casos de estudio 47
5.2.1. Caso de Estudio 47

6. Implementación **49**
6.1. Hardware utilizado 49
6.2. Software utilizado 50
6.2.1. Visual Studio Code 50
6.2.2. Docker y Docker Desktop 50
6.2.3. Node JS 51
6.2.4. React JS 51
6.2.5. RabbitMQ 51
6.2.6. MongoDB 52
6.2.7. Express JS 52
6.3. Lenguajes de programación 52
6.3.1. Javascript y JSX 52
6.3.2. Typescript 53
6.4. Estrategia de implementación 54

7. Resultados **55**
7.1. Implantación 55
7.1.1. Requerimientos 55
7.1.2. Preparación de Ambiente 55
7.1.3. Documentación 55
7.1.4. Manual de Usuario 55
7.1.5. Documentación de desarrollo 55

8. Conclusiones **56**
8.1. Conclusiones 56

Bibliografía **57**

Lista de Figuras

2.1. Clasificación de Plataformas de Procesamiento. Recuperado de [1] 6
2.2. Arquitectura y componentes de la plataforma Apache Samza. Recuperado de [1] 10
2.3. Arquitectura y componentes de la plataforma Apache Flink. Recuperado de [2] 12
2.4. Arquitectura de un sistema Flink. Recuperado de [3] 13
2.5. Arquitectura de Apache Storm. Recuperado de [1] 14
2.6. Topología base de una Aplicación Storm. Recuperado de [4] 15
2.7. Arquitectura de alto nivel de Apache Spark. Recuperado de [5] 17
2.8. Entidades clave para ejecutar una aplicación Apache Spark. Recuperado de [6] 19

3.1. Arquitectura del entorno web para la simulación de plataformas de procesamiento de streams 25
3.2. Arquitectura enfocada en micro servicios 26
3.3. Metodología Desarrollo Scrum. 29
3.4. Fases de una Metodología Scrum. 30
3.5. Diagrama General de Caso de Uso. 36
3.6. Diagrama de estado para usuarios. 37

4.1. Ejemplo Diagrama de Despliegue 41
4.2. Ejemplo Diagrama de Componentes 42
4.3. Ejemplo Diagrama de Paquetes 43
4.4. Ejemplo Diagrama de Clases 44

Índice de tablas

Capítulo 1

Introducción

El área de estudio de este trabajo de título se centra en el desarrollo de un entorno web para la simulación de plataformas de procesamiento de streams [7, 8, 9]. Este campo de investigación se sitúa en la intersección de la informática distribuida, la ciencia de datos y la ingeniería de software, y se enmarca en el contexto de la creciente demanda de soluciones tecnológicas que puedan manejar grandes volúmenes de datos en tiempo real, especialmente en áreas como la analítica de datos, la Internet de las cosas o IoT (*Internet of Things*) y la detección de anomalías [9].

Las plataformas de procesamiento de streams son sistemas diseñados para analizar, procesar y responder a flujos continuos de datos en tiempo real. Estas plataformas son fundamentales para todas aquellas aplicaciones que generan grandes cantidades de datos que deben ser procesados en tiempo real, como el monitoreo de sensores, el análisis de trafico de red [10], la detección de amenazas a la seguridad y la analítica de datos en tiempo real. Un ejemplo prominente de una plataforma de procesamiento de streams es *Apache Kafka* [11], que proporciona una infraestructura distribuida para el almacenamiento y procesamiento de datos de manera escalable [12]. Otra herramienta popular es *Apache Flink* [13], que ofrece capacidades avanzadas de procesamiento de streams y de procesamiento de datos por lotes en un solo motor unificado [14]. Por último, *Spark Streaming* [15], que forma parte del ecosistema de Apache Spark, permite el procesamiento de datos en tiempo real de forma escalable [5].

Las plataformas de procesamiento de streams enfrentan varios desafíos y problemas que deben abordarse para garantizar su eficacia y utilidad en entornos del mundo real. Dentro de los cuales podemos destacar la necesidad de procesar grandes cantidades de datos en cuestión de segundos, con latencias extremadamente bajas para proporcionar información a medida que los datos arriban [3]. Los componentes de la aplicación y la infraestructura deben asegurar un alto nivel de tolerancia a fallos. El grado de complejidad en la configuración y uso de estas aplicaciones, así como los obstáculos que la portabilidad y despliegue pueden representar tanto para usuarios experimentados como nuevos en esta área.

1.1. Principales Contribuciones

Para abordar los desafíos presentes en este trabajo de título se propone diseñar, desarrollar e implementar un entorno web centrado en el uso de una arquitectura de servicios y el sistema de contenedores Docker [16] como ejes principales. Esta iniciativa surge de la necesidad de abordar las complejidades inherentes a la creación de modelos concurrentes o paralelos de una plataforma de procesamiento de datos. Proporcionando una herramienta versátil, accesible para la experimentación, modelamiento, simulación y análisis de resultado. Abstrayendo y encapsulando estas complejidades, pudiendo simplificar el proceso de construcción y simulación de modelos de procesamiento de plataformas de streams para los usuarios.

Con el fin de alcanzar este objetivo, se aplica la metodología ágil Scrum [17]. Esta elección se basa en la premisa de facilitar la entrega progresiva de funcionalidades y proporcionar una mayor flexibilidad para ajustarse a los cambios en los requisitos del proyecto a lo largo del tiempo.

Respecto a las contribuciones esperadas se incluye el desarrollo de un entorno web para la simulación de plataformas de procesamiento de streams orientado a servicios. La evaluación de la efectividad y rendimiento del entorno desarrollado en comparación con otras herramientas similares. Documentación exhaustiva del proceso de desarrollo, incluyendo el diseño, implementación y pruebas del sistema. Identificación de áreas de mejora y recomendaciones para futuras investigaciones en este campo.

1.2. Estructura del Documento

El presente documento de tesis se estructura por capítulos y se organiza de la siguiente manera: En el Capítulo 2, se establece el marco conceptual para comprender el contexto de estudio. Se proporcionan definiciones de conceptos relevantes, junto con una revisión del estado del arte y una discusión sobre trabajos relacionados. En el Capítulo 3, se realiza un análisis detallado del contexto,los problemas actuales de las plataformas de procesamiento de streams, se propone una solución para estos desafíos y se establecen los objetivos del proyecto. En el Capítulo 4, se detalla el diseño de la solución propuesta en conjunto con los correspondientes diagramas arquitectónicos. En el Capítulo 5, se presenta el diseño de experimentos, análisis estadístico, descripción de Datasets y los casos de estudio pertinentes. En el Capítulo 6, se detallan el hardware y software utilizados, lenguajes de programación seleccionados y estrategias de implementación. En el Capítulo 7, se presenta el análisis de resultados, preparación del ambiente para la implantación, elaboración de la documentación y manual de usuario. Finalmente el Capítulo 8, aborda las conclusiones finales y los trabajos futuros derivados para la simulación de plataformas de procesamiento de streams.

Capítulo 2

Marco Conceptual y Estado del Arte

2.1. Marco Conceptual

Esta sección sirve como fundamento teórico imprescindible para adentrarse plenamente en la temática central abordada en el trabajo de título. Se presentan conceptos clave y definiciones relacionadas con el procesamiento y simulación de plataformas de procesamiento de streams. Estableciendo las bases para el análisis y diseño de la solución propuesta.

2.1.1. Modelos de Datos en Sistemas de Procesamiento de Streams

Modelo Relacional de Streaming

En los primeros sistemas de procesamiento de streams se implementó el modelo relacional de streaming, el cual se encuentra representado por plataformas como Borealis [18], Aurora [19], Stream [20], CDER [21], TelegraphCQ [22], Gigascope [23].

En un modelo relacional un stream es interpretado como la descripción de una relación cambiante sobre un esquema común. Además, se asume que los elementos de un stream poseen marcas de tiempo (*timestamps*) o números de secuencia, permitiendo definir ventanas sobre estos streams. Los streams de datos pueden ser generados por fuentes externas o ser producidos por consultas continuas. Los operadores generan streams de eventos que describen la vista cambiante calculada sobre el stream de entrada según la semántica relacional del operador, donde tanto la semántica como el esquema de relaciones son impuestos por el sistema.

Modelo de Streaming de Flujo de Datos

En la segunda generación de sistemas de procesamiento de streams se implementó el modelo de streaming de flujo de datos, el cual se encuentra representado por: Google

DataFlow Model [24], Apache Flink [14], Spark Streaming [7].

El modelo de streaming de flujo de datos se representa como un grafo de flujo de datos, es decir, un grafo dirigido G = (E,V) , donde los vértices en V representan operadores y las aristas en E denotan streams de datos [25]. Las aplicaciones pueden operar en uno de las 3 siguientes modos:

Tiempo de Evento o Aplicación: Tiempo cuando los eventos son generados por las fuentes de datos.

Tiempo de Asimilación o Ingestión: Tiempo cuando los eventos arriban al sistema.

Tiempo de procesamiento: Tiempo cuando los eventos son procesados en los sistemas de streaming.

En sistemas modernos de flujo de datos de streaming, se puede asimilar cualquier tipo de stream de entrada, independiente de si sus elementos representan adiciones, eliminaciones o reemplazos.

2.1.2. Métodos de Procesamiento de datos

El procesamiento de datos se divide en 3 enfoques principales: procesamiento por lotes, procesamiento por micro-lotes y procesamiento de streams.

Procesamiento por Lotes

el procesamiento lotes (*batch*) se define como el análisis de grandes conjuntos de datos estáticos, que se recopilan durante períodos de tiempo anteriores. Sin embargo, esta técnica tiene una gran latencia, con respuestas superiores a 30 segundos, mientras que varias aplicaciones requieren procesamiento en tiempo real, con respuestas en el orden de microsegundos [26]. A pesar de ello, esta técnica puede realizar un procesamiento casi en tiempo real mediante el procesamiento por micro-lotes.

Procesamiento Micro-Lotes

El procesamiento por micro-lotes (*micro-batch*) trata los streams como una secuencia de pequeños bloques de datos. Para intervalos de tiempo cortos, las entradas que son recibidas se agrupan en bloques de datos y se entregan al sistema por lotes para su procesamiento [3].

Procesamiento de Streams

El procesamiento de streams analiza secuencias masivas de datos ilimitados que se generan de forma continúa. Además, la latencia del procesamiento de streams es mejor que los micro-lotes, dado que los mensajes son procesados inmediatamente después de su llegada. Esto se traduce en un desempeño mejor en tiempo real pero presentando una tolerancia a fallos mas costosa en comparación, ya que debe realizarse por cada mensaje procesado [3].

2.1.3. Procesamiento de Streams

El procesamiento de streams (*Stream Processing*) se define como un método de computación distribuida capaz de soportar la recopilación y análisis de grandes volúmenes de datos de un flujo de datos (*data stream*), cuyo objetivo principal es ayudar en la toma de decisiones en tiempo real [27].

El procesamiento de streams abarca una serie de conceptos y principios fundamentales que son esenciales para comprender su funcionamiento y aplicaciones. A continuación se presentan algunos conceptos clave junto con sus definiciones:

- Flujo de datos (*Data Stream*): Conjunto de datos que se produce incrementalmente a lo largo del tiempo, en lugar de estar disponible en su totalidad antes de que comience su procesamiento [25]. Estos conjuntos de datos en tiempo real pueden ser potencialmente ilimitados.

- Consulta en tiempo real (*Streaming Query*): Captura eventos y produce resultados de manera continua, utilizando un único recorrido o un número limitado de recorridos sobre los datos [25].

- Procesamiento en Tiempo Real (*Real-Time Processing*): El procesamiento en tiempo real se refiere a la capacidad de analizar y actuar sobre los datos a medida que llegan, sin demoras significativas.

2.1.4. Plataformas de Procesamiento de Streams

Las plataformas de procesamiento de streams han surgido como herramientas fundamentales en el procesamiento de datos en tiempo real. Facilitan el análisis y procesamiento continuo de flujos de datos, siendo vitales en contextos donde la velocidad y la precisión son críticas. En este contexto, es pertinente considerar las contribuciones realizadas por los autores **Carbone, Paris, et. al.**, quienes destacan que estas plataformas se enfocan en ofrecer latencias extremadamente bajas, asegurando que los resultados del análisis estén disponibles en tiempo real o cerca de él. Además, los autores **Zaharia, M, et. al.** [7] subrayan

procesamiento de datos en tiempo real es esencial para una amplia gama de aplicaciones, tales como redes sociales, publicidad en línea, detección de fraudes y monitorización de la infraestructura.

Estas plataformas se distinguen de los sistemas tradicionales de procesamiento por lotes al permitir el procesamiento continuo de datos entrantes, en contraposición a esperar la acumulación de un conjunto completo de datos antes de comenzar el procesamiento. **Kreps, Jay** señala que el procesamiento de streams representa un cambio de paradigma en el tratamiento de los datos, considerándolos eventos que ocurren en el tiempo y procesándolos de manera continua a medida que llegan [28].

2.1.5. Clasificación de Plataformas de Procesamiento de Streams

La computación distribuida y paralela han emergido como un solución principal para el procesamiento de conjuntos de datos (*datasets*) muy grandes. Sin embargo, su complejidad y algunas de sus características pueden evitar su máxima utilización productiva por parte de los usuarios comunes. La partición y distribución de datos, la escalabilidad, el equilibrio de carga, la tolerancia a fallos y la alta disponibilidad se encuentran entre las principales preocupaciones [1]. Se han lanzado varios marcos para abstraer estas funciones y proporcionar a los usuarios soluciones de alto nivel. Estas plataformas suelen clasificarse según su enfoque de procesamiento de datos, es decir, procesamiento por lotes (*batch processing* 2.1.2), procesamiento de streams (*stream processing*)2.1.2 y procesamiento híbrido (*Hybrid*). Este ultimó combina diferentes técnicas de procesamiento de datos, incluido el procesamiento por lotes, micro-lotes 2.1.2 y procesamiento de streams en tiempo real. A continuación se presenta una figura que ilustra esta clasificación:

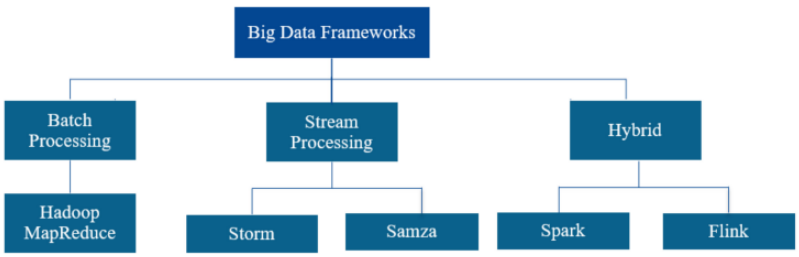

Figura 2.1: Clasificación de Plataformas de Procesamiento. Recuperado de [1]

2.1.6. Modelos de Simulación en Sistemas de Procesamiento de Streams

Simulación Basada en Eventos Complejos

- Evento: Se define como un suceso que tiene lugar en un sistema o dominio particular, siendo algo que ha ocurrido o se contempla que ha ocurrido en ese dominio. La palabra ”evento” también se utiliza para referirse a una entidad de programación que representa dicho suceso en un sistema informático. [29].
- Procesamiento de Eventos: Se define como la computación que realiza operaciones en eventos. Eventos comunes de procesamiento incluyen lectura, creación, transformación y eliminación de eventos [29].
- Stream de Eventos: Conjunto de eventos asociados. A menudo, es un conjunto ordenado temporal de forma total, es decir, hay un orden bien definido basado en marcas de tiempo (*timestamp*) para los eventos en los streams.
- Procesamiento de Eventos Complejos: El procesamiento de eventos complejos (CEP) es una tecnología basada en eventos que permite procesar y correlacionar streams de grandes datos, con el fin de detectar rápidamente eventos o situaciones significativas y responder a ellas de manera adecuada [30].

CEPSim: Modelado y simulación de sistemas de Procesamiento de Eventos Complejos en entornos de nube

Los autores **Wilson, A, et al.** nos presentan un enfoque para modelar y simular sistemas de Procesamiento de Eventos Complejos en entornos de nube [31]. La herramienta CEPSim permite a los investigadores y desarrolladores estudiar la escalabilidad y el rendimiento de los sistemas CEP y los efectos de aplicar diferentes estrategias de procesamiento de consultas en entornos de nube. Cabe recalcar que CEPSim se construyó sobre la herramienta CloudSim.

ECSSim: Un simulador para dispositivos de borde en la nube

En el trabajo de investigación realizado por **Amarasinghe, K, et al.** presentan el simulador ECSSim [32], este simulador se ha diseñado específicamente para sistemas de borde en la nube, también conocidos como *Edge-Cloud Systems*. Este tipo de sistemas se ubican en un punto intermedio entre la computación en la nube (*Cloud Computing*) y el borde de una red, están diseñados para ofrecer servicios de computación y almacenamiento más cercano a los usuarios finales.

El simulador ECSSim proporciona una plataforma para modelar y simular el comportamiento de los sistemas mencionados con anterioridad. Permitiendo a los investigadores y desarrolladores evaluar el rendimiento, la escalabilidad y otros aspectos claves de los sistemas de borde en la nube en entornos controlados.

ECSNet++: Un kit de herramientas de simulación para plataformas de procesamiento de streams en la nube.

En el trabajo de investigación realizado por **Amarasinghe, K, et al.** presentan un kit de herramientas de simulación llamado ECSNeT++ [33] para plataformas de procesamiento de streams. El simulador se implementa sobre el simulador de red OMNeT++. Este kit de herramientas está destinado a facilitar la simulación y evaluación del rendimiento de estas plataformas, lo que permite a los investigadores y desarrolladores explorar diferentes configuraciones y escenarios de uso en entornos controlados. Por otra parte, tanto ECSNeT++ como ECSSim están diseñados para funcionar en sistemas Edge-Cloud, razón por la cual no es posible utilizar redes basadas en conmutadores de uso general como el Fat-Tree.

Evaluación de rendimiento basada en simulación de eventos discretos de sistemas de procesamiento de streams

La investigación presentada en la Conferencia Internacional IEEE por los autores **Muthukrishnan, P. y Fadnis, K** [34], plantea un enfoque basado en la evaluación del rendimiento de los sistemas de procesamiento de streams utilizando la simulación de eventos discretos o DES (*Discrete Event Simulation*). Esto implica modelar el comportamiento del sistema de procesamiento de streams y simular su funcionamiento bajo diferentes condiciones y cargas de trabajo para evaluar su rendimiento y eficiencia. Además, permite comparar diferentes configuraciones, algoritmos o arquitecturas de procesamiento de streams mediante la simulación de eventos discretos. La implementación de este nuevo enfoque centrado en DES puede llegar a ofrecer información útil para diseñadores, desarrolladores y gestores de sistemas de procesamiento de streams interesados en comprender y mejorar el rendimiento de sus sistemas.

2.2. Estado del Arte

2.2.1. Principales Plataformas de Procesamiento de Streams

En esta subsección, exploramos algunas de las principales plataformas de procesamiento de streams disponibles en el mercado. Desde Apache Samza [35], con su enfoque en ofrecer una sistema confiable y fácil de usar para el procesamiento de datos en tiempo real, además de ofrecer una API intuitiva para desarrolladores basada en Apache Kafka

Streams, Apache Flink [13] se centra en ofrecer un sistema que combine tanto el procesamiento de datos en tiempo real como el procesamiento por lotes en una sola plataforma unificada , Apache Spark Streaming [15] con sus capacidades avanzadas de procesamiento y análisis, Apache Storm [36], una opción confiable para aplicaciones que requieren tolerancia a fallos y escalabilidad horizontal en entornos distribuidos. Ofreciendo un modelo de programación flexible que permite a los desarrolladores definir topologías de procesamiento mediante la composición de ”bolts” y ”spouts. A través de esta exploración, descubriremos cómo estas plataformas están transformando la forma en que las organizaciones gestionan y aprovechan sus datos en tiempo real, abriendo nuevas posibilidades para la innovación y la toma de decisiones ágil.

2.2.2. Apache Samza

Apache Samza [37] se basa en un diseño unificado para el procesamiento con estado de datos por lote y stream de datos en tiempo real de alto volumen. Samza esta diseñado para soportar altas cantidades de streams de datos que se deben procesar (*throughput*) en un período de tiempo, mientras provee una alta recuperación a fallos y fiabilidad. Para lograr cumplir con estos objetivos, Samza usa abstracciones claves, tales como la partición de streams, captura de registros de cambio y la gestión de estado local. Samza se basa en un modelo descentralizado donde no hay un maestro a nivel de sistema para coordinar actividades; en su lugar, cada trabajo tiene un coordinador liviano para administrarlo. Samza ha sido adoptado por grandes compañías, dentro de las que incluyen LinkedIn, VMWare, Uber, Netflix y TripAdvisor [35].

Arquitectura Samza

Apache Samza exhibe una arquitectura que consta de tres capas distintas: la capa de streaming, la capa de ejecución y la capa de procesamiento (ver Figura 2.2). A continuación, se ofrece una breve descripción de cada una de estas capas:

- Capa de Streaming (*Streaming Layer*): Responsable de proporcionar una fuente de datos reproducible, tales como Apache Kafka [11], AWS Kinesis [38] o Azure EventHub [39].
- Capa de Ejecución (*Execution Layer*): Responsable de la planificación y gestión de recursos.
- Capa de Procesamiento (*Processing Layer*): Responsable del procesamiento de datos y gestión del flujo.

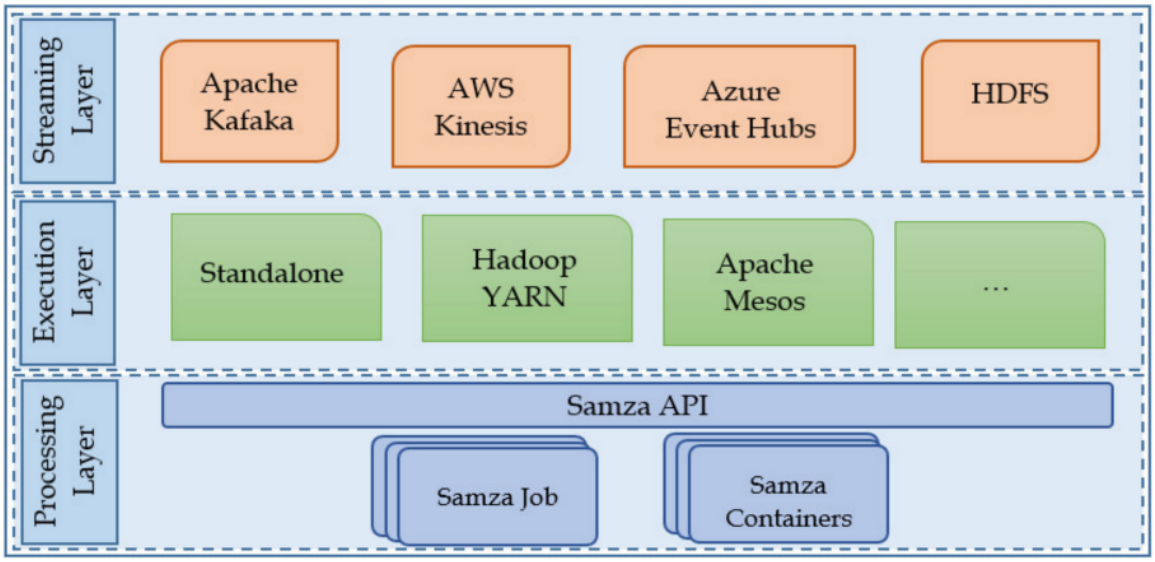

Figura 2.2: Arquitectura y componentes de la plataforma Apache Samza. Recuperado de [1]

En la arquitectura Samza los streaming de datos pueden ser suministrados por cualquier fuente de recursos existente. De igual manera, la gestión de clúster y planificación es posible a través de Apache YARN [40] o Mesos [41]. Además, Samza cuenta con soporte integrado para el streaming de datos de Apache Kafka y Apache YARN para la ejecución de trabajos.

El modelo de ejecución de un trabajo de Samza se basa en el concepto de publicación/suscripción de tareas. Esto implica escuchar un stream de datos de un tópico de Kafka, procesar el mensaje cuando llega y luego enviar su salida a otro stream. En cuanto a los streams de datos en Kafka, están compuestos por varias particiones basadas en pares de valores clave. Las tareas de Samza consumen estos streams de datos y pueden ejecutar múltiples tareas en paralelo para consumir todas las particiones de streams en paralelo. Estas tareas se ejecutan en contenedores YARN, los cuales se distribuyen uniformemente en múltiples nodos en el clúster por YARN. Después de la ejecución, la salida puede ser enviada a otro flujo para su procesamiento adicional [1].

2.2.3. Apache Flink

Apache Flink representa una plataforma de procesamiento híbrida que ofrece soporte tanto para streams como para procesamiento de lotes, desarrollada en Java y Scala [1]. Su arquitectura se centra en un motor de flujo de datos distribuidos que procesa de manera uniforme trabajos tanto por lotes como de transmisión, los cuales están compuestos por tareas interconectadas con estado [42].

Un programa en ejecución en Flink se define como un grafo acíclico dirigido (DAG) de operadores con estado conectados a través de flujos de datos. Flink proporciona dos APIs principales: la API DataSet, diseñada para procesamiento de conjuntos de datos finitos (a menudo asociado con el procesamiento por lotes), y la API DataStream, utilizada para el procesamiento de flujos de datos potencialmente ilimitados (referido al procesamiento de streams). Además, Flink ofrece un entorno de ejecución distribuido para clústers, así como uno local, lo que permite ejecutar y depurar programas en un entorno de desarrollo local, facilitando así el desarrollo. El motor distribuido de Flink adapta el plan de ejecución al entorno del clúster, lo que le permite ejecutar diferentes planes en función del entorno y la distribución de los datos.

Arquitectura Flink

Apache Flink proporciona un entorno de ejecución unificado donde todos los programas se ejecutan. Estos programas se organizan como grafos dirigidos (JobGraphs) de operaciones paralelizadas, que también pueden incluir iteraciones [43]. Un JobGraph consiste en nodos y aristas; hay dos tipos de nodos: aquellos con estado (operadores) y aquellos lógicos (resultados intermedios o IRs). A continuación, se detallan los componentes de la arquitectura de Flink, como se muestra en la Figura 2.3:

- Librerias e Interfaces: En adición a la interfaz clásica de Flink, la librería **FlinkML** ofrece una serie de algoritmos y análisis de datos para machine learning. *Gelly* permite el análisis de grafos, mientras que *Table API* permite especificaciones declarativas de consultas similares a SQL [2].

- Stream Builder y Common API: Encargados de traducir entre el entorno de ejecución y las interfaces (API), mediante la transformación de grafos dirigidos de operaciones lógicas en programas genéricos de streams de datos que se ejecutan en su propio entorno de ejecución. La optimización automática de programas de flujo de datos esta incluida en este proceso. El optimizador integrado por ejemplo elige el mejor algoritmo de unión para cada caso de uso respectivo, con el usuario especificando únicamente una operación de unión abstracta [2].

- Gestor de Clúster y Almacenamiento (*Cluster Management and Storage*): Flink es compatible con un número de gestores de clúster y soluciones de almacenamiento, tales como Apache Tez [44], Apache Kafka [45], Apache HDFS [46] y Apache Hadoop YARN [40].

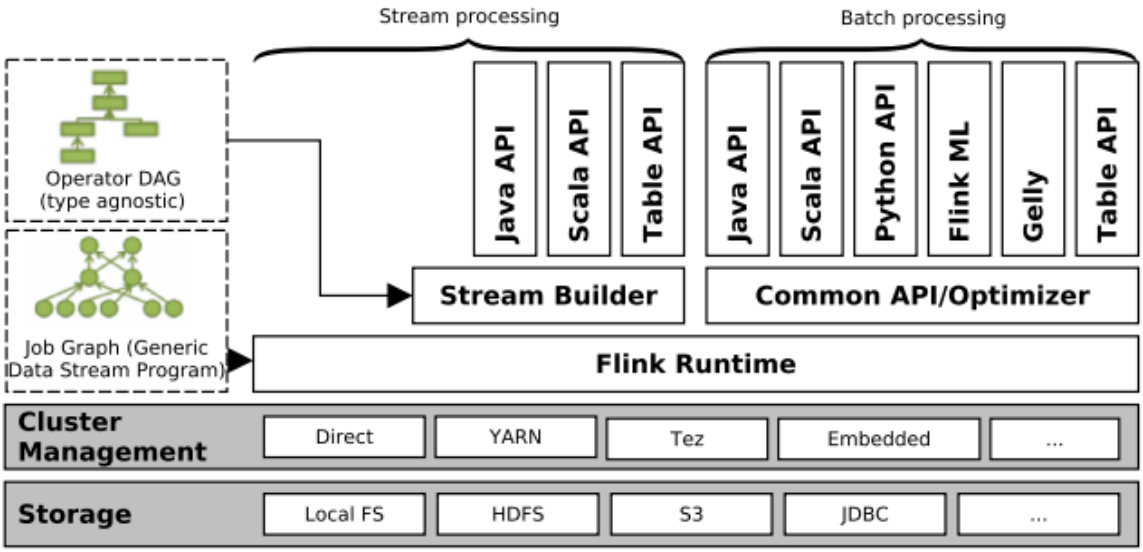

Figura 2.3: Arquitectura y componentes de la plataforma Apache Flink. Recuperado de [2]

Esta arquitectura propia de Flink posee similitudes a la propuesta por Storm, Flink utiliza un modelo maestro-trabajador. También se ha de diferenciar la arquitectura presentada con anterioridad a lo que es la arquitectura de un sistema Flink (ver Figura 2.4, esta última toma en cuenta los siguientes componentes:

- Gestor de Trabajos (*Job Manager*): Encargado de interactuar con las aplicaciones de los clientes y responsabilidades similares al nodo maestro de Storm. Recibe una serie de aplicaciones de clientes, organiza las tareas y las envía a los trabajadores. Además, se encarga de mantener el estado de todas las ejecuciones y los estados de cada trabajador.

- Gestor de Tareas (*Task Manager*): Ejecuta las tareas asignadas por el gestor de trabajos e intercambia información con otros trabajadores cuando es necesario. Cada gestor de tareas provee un número de ranuras de procesamiento a los clústers que son usados para ejecutar tareas en paralelo.

- Abstracción de Stream: Es llamada flujo de datos (DataStreams) y se define como secuencias de registros parcialmente ordenados. DataStreams pueden ser creados desde fuentes externas tales como colas de mensajes, socket streams, generadores personalizados o por la invocación de operaciones en otros DataStreams. DataStream posee soporte para una serie de operadores tales como *map, filtering, reduction* y pueden

ser paralelizados colocando instancias paralelas para ser ejecutadas en diferentes particiones de los respectivos streams, permitiendo la ejecución distribuida de transformaciones de streams.

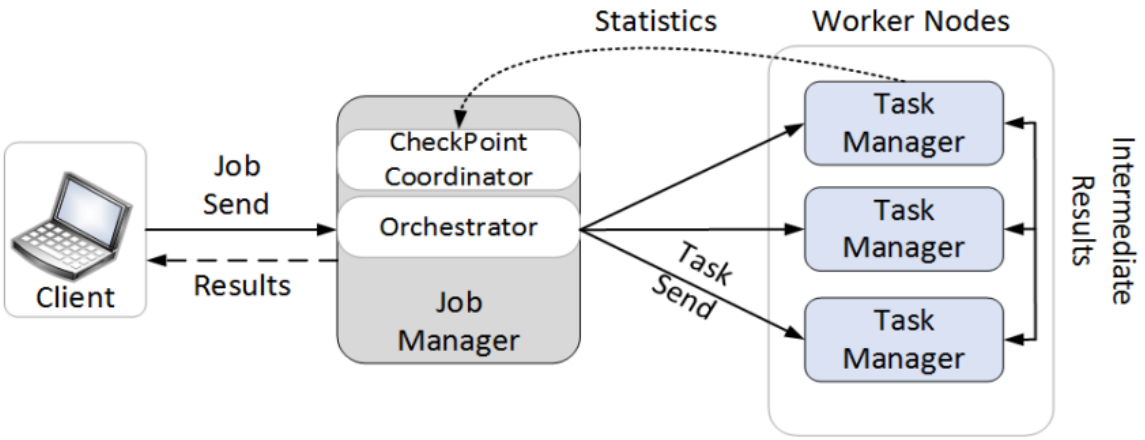

Figura 2.4: Arquitectura de un sistema Flink. Recuperado de [3]

2.2.4. Apache Storm

Apache Storm es una plataforma de procesamiento distribuido para streams de datos en tiempo real. Desde sus inicios, Storm ha sido ampliamente reconocido y adoptado por algunas de las grandes industrias, tales como Twitter, Yahoo!, Alibaba, Groupon, entre otras [47]. Está diseñado para procesar y analizar grandes cantidades de streams de datos ilimitados, los cuales pueden provenir de diversas fuentes y publicar actualizaciones en tiempo real en la interfaz de usuario u otros lugares sin almacenar datos reales. Storm es altamente escalable y proporciona baja latencia con una interfaz fácil de usar a través de la cual los desarrolladores pueden programar virtualmente en cualquier lenguaje de programación [48]. Para lograr esta independencia de lenguaje, utiliza la definición de Thrift para definir e implementar topologías [1].

Arquitectura Storm

Apache Storm adopta una arquitectura maestro-esclavo, con un único nodo maestro y varios nodos esclavos. En la implementación física de Storm, se distinguen tres componentes principales(Ver Figura 2.5):

- Nimbus: Nimbus es un demonio maestro (master daemon) que distribuye el trabajo entre todos los trabajadores disponibles. Sus principales responsabilidades incluyen el asignamiento de tareas a los nodos trabajadores, el seguimiento del progreso de las tareas, la reprogramación de las tareas para otros nodos trabajadores en caso de falla y la monitorización de las topologías que necesitan ser asignadas, realizando la asignación entre esas topologías y los supervisores cuando es necesario [1].
- Zookeeper: Zookeeper es utilizado a modo de coordinador entre Nimbus y los supervisores, ya que estos últimos no poseen estado Zookeeper se encarga de toda la gestión de estado [48].
- Supervisor: Los supervisores monitorean los procesos de cada topología e informan del estado a Nimbus utilizando el protocolo Heartbeat. Además, anuncian respecto a la disponibilidad de alguna ranura (*slot*) disponible que pueda asumir más trabajo.

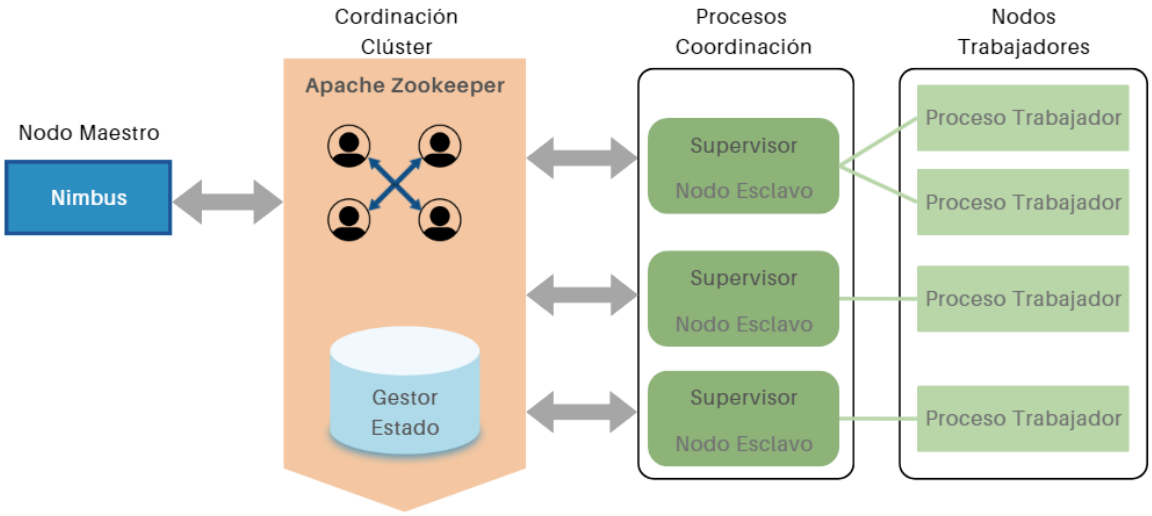

Figura 2.5: Arquitectura de Apache Storm. Recuperado de [1]

Modelo de procesamiento de datos y topologías

El modelo de procesamiento básico de datos de Storm consiste de 4 abstracciones: topología, spouts, bolts, stream. En Storm, un stream se define como una secuencia de tuplas ilimitadas, donde las tuplas son listas nombradas de valores, que pueden ser de cualquier tipo, incluyendo cadenas de texto, enteros, números de punto flotante, etc. La lógica de cualquier aplicación Storm en tiempo real se presenta en forma de una topología, compuesta de una red de bolts y spouts (Ver Figura 2.6). Los Spouts son fuentes de stream que

esencialmente se conectan con una fuente de datos como Kafka [45] o Kestrel. Estos reciben continuamente datos y los convierten en un flujo de tuplas para pasarlos a los bolts. Los bolts son las unidades de procesamiento de una aplicación de Storm que pueden realizar una variedad de tareas.

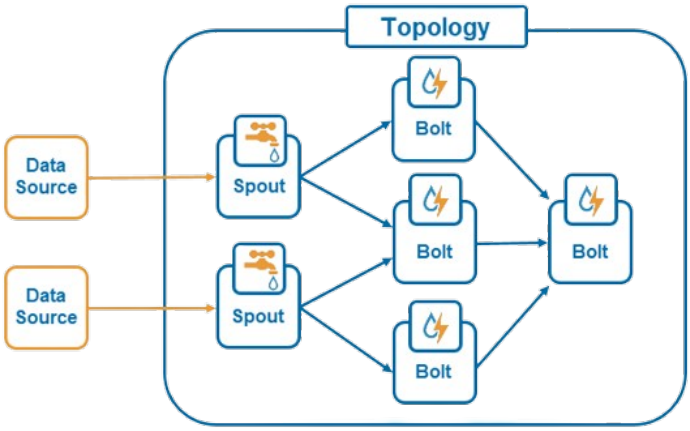

Figura 2.6: Topología base de una Aplicación Storm. Recuperado de [4]

.

2.2.5. Apache Spark Streaming

Spark es una plataforma para el procesamiento de datos distribuidos, escrita en los lenguajes de programación Java y Scala. Esta posee una serie de librerías de alto nivel ejecutándose en el core de Spark (*Spark Engine*), incluyendo Spark Streaming para el procesamiento de streams [7]. La abstracción de streams se denomina como Stream Discreto (*D-Stream*), definido como un conjunto de tareas cortas, sin estado y deterministas. En Spark, el calculo o cómputo de streaming se trata como series de cómputos deterministas en lotes sobre pequeños intervalos de tiempo. Un trabajo en Spark es definido como un cómputo paralelo que consiste de múltiples tareas y donde cada tarea es una unidad de trabajo que es enviado al gestor de tareas (*Task Manager*). Cuando un stream ingresa a Spark, este divide los datos en micro-lotes para posteriormente ser trabajados como datos de entrada o input data de los conjuntos de datos resilientes o Distributed Resilient Dataset (RDD) [49], una vez realizado este proceso la clase principal en el motor de Spark (Spark Engine) procede a almacenarlas en memoria. Finalmente este motor de Spark se ejecuta generando trabajos para procesar los micro-lotes.

Arquitectura Spark

Apache Spark se compone de varios componentes esenciales, entre los que destacan Spark Core y las bibliotecas de alto nivel. Spark Core opera en diversos gestores de clúster y puede acceder a datos en diferentes fuentes de datos Hadoop. Además, se pueden construir muchos paquetes adicionales para extender las capacidades de Spark Core y de las bibliotecas de alto nivel. A continuación, se presenta una breve descripción de cada capa que conforma la arquitectura de Spark mencionada anteriormente:

- Librerías de alto nivel (*Libraries*): Varias bibliotecas se han construido sobre la base del core de Spark para manejar diferentes cargas de trabajo. Estas incluyen MLlib de Spark para machine learning [50], GraphX para procesamiento de gráficos [51, 52], Spark Streaming para análisis de streamings[7], y Spark SQL para procesamiento de estructuras de datos [53].

- Motor Spark (*Spark Core*): Spark core proporciona una interfaz de programación simple para procesar grandes conjuntos de datos a gran escala. Cabe resaltar que el Spark core está implementado en Scala, pero cuenta con APIs en Scala, Java, Python y R. Estas APIs admiten operaciones esenciales para los algoritmos de análisis de datos en las bibliotecas de alto nivel. Además, el Spark core ofrece funcionalidades principales para la computación de clúster en memoria, incluida la gestión de memoria, la planificación de trabajos, la redistribución de datos y recuperación de fallos. Con estas funcionalidades, una aplicación de Spark puede desarrollarse utilizando los recursos de CPU, memoria y almacenamiento de un clúster de cómputo [5].

- Gestores de Clúster (*Cluster Management*: Un gestor de clúster se emplea para obtener recursos del clúster destinados a la ejecución de tareas. Además de administrar el intercambio de recursos de aplicaciones Spark. Spark core opera sobre diversos gestores de clústers que incluyen Hadoop YARN [54], Apache Mesos [41], Amazon EC2 y el gestor de clúster integrado de Spark.

- Almacenamiento (*Storage*): Spark core puede acceder a datos en HDFS, Cassandra , HBase, Hive, Alluxio y algunas fuente de datos Hadoop [5].

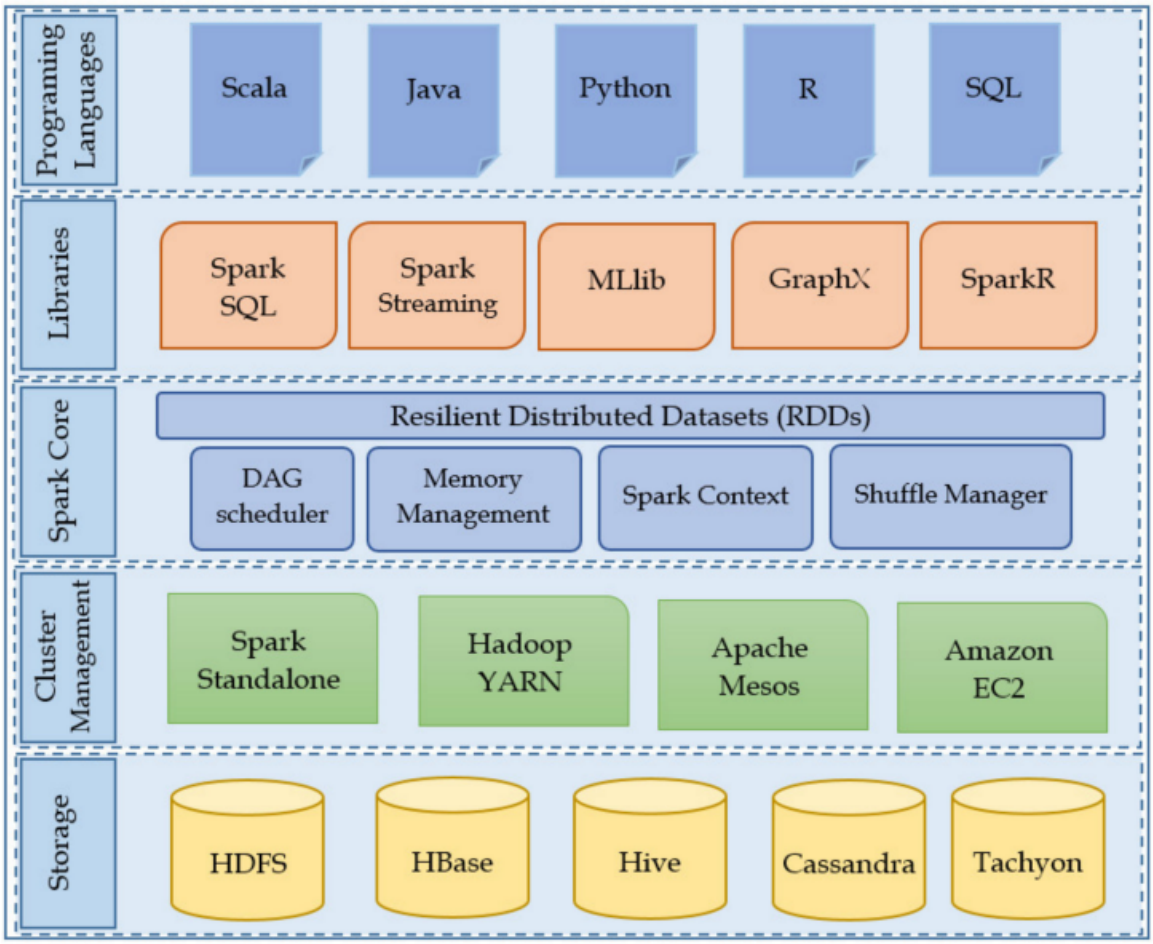

Figura 2.7: Arquitectura de alto nivel de Apache Spark. Recuperado de [5]

Aplicaciones en Spark Streaming

Para la ejecución de una aplicación de Spark se toman en cuenta los siguientes componentes clave (Ver Figura 2.8):

- Programa Controlador (*Driver Program*): Un programa controlador es una aplicación que utiliza Spark como una librería y contiene lógica de procesamiento de datos, este se encarga de crear el objeto *SparkContext* [5].
- Contexto de Spark (*SparkContext*): Cuando el SparkContext se ejecuta sobre un clúster, este tiene la capacidad de conectarse a varios tipos de gestores de clústers (ya sea el propio gestor de clúster independiente de Spark, Mesos, YARN o Kubernetes) y enviar tareas a los ejecutores [1].

- Ejecutores (*Executors*): Procesos JVM encargados de realizar cálculos y almacenar datos de la aplicación, cada aplicación obtiene sus propios procesos de ejecución que permanecen activos durante toda la duración de la aplicación y ejecutan tareas en múltiples hilos. Aislando las aplicaciones entre sí, tanto en el lado de la programación (cada controlador programa sus propias tareas) como en el lado del ejecutor (las tareas de diferentes aplicaciones se ejecutan en diferentes JVMs). Sin embargo, también significa que los datos no pueden compartirse entre diferentes aplicaciones de Spark (instancias de SparkContext) sin escribirlos en un sistema de almacenamiento externo [5].

- Nodo Trabajador (*Worker node*): Cualquier nodo que pueda ejecutar código de aplicación en el clúster. Permite alojar a los ejecutores y proveerlos de recursos de computación, tales como: CPU, memoria, y recursos de almacenamiento [1].

- Tarea (*Task*): Una tarea o task es la unidad de trabajo más pequeña que Spark envía a un ejecutor [5].

  A continuación se presenta la Figura 2.8 que resume el comportamiento de los componentes clave descritos con anterioridad. Aquí podemos observar la distribución de un clúster Spark, en donde las aplicaciones o trabajos dentro de Spark se ejecutan como procesos independientes en el clúster, sincronizados por el programa controlador, contexto de Spark, Ejecutores, nodos trabajadores y tareas. Spark es capaz de extrapolar el mecanismo de topologías de Apache Storm, de tal manera que las aplicaciones son equivalentes a las topologías presentes en Storm, con la desventaja de que este concepto en Spark se trabaja a través del intercambio de mensajes entre diferentes programas, realizado solamente de forma indirecta, como la escritura de datos en un archivo, repercutiendo en la latencia que podría estar entorno a segundos de respuesta en aplicaciones con varias operaciones [3].

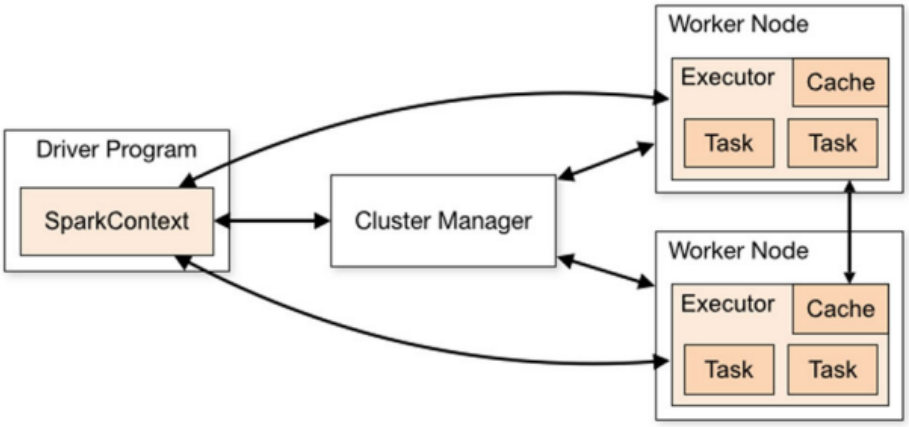

Figura 2.8: Entidades clave para ejecutar una aplicación Apache Spark. Recuperado de [6]

2.2.6. Trabajos Relacionados

A continuación, se presenta una recopilación de investigaciones previas relacionadas con la simulación de plataformas de procesamiento de streams:

Flow: Una plataforma de procesamiento de streams paralela y distribuida

La investigación realizada por **Park, S, et al.** nos presenta una plataforma paralela y distribuida (*Flow*) para simular aplicaciones de procesamiento de streams [55]. Este enfoque no incluye componentes de hardware o software del sistema de procesamiento de streams. En cambio, esta diseñado de tal forma que es capaz de capturar el flujo de datos a nivel de aplicación.

Flow utiliza un enfoque de sincronización conservativa híbrida, esta destaca por su capacidad para procesar eventos locales en orden de marca de tiempo inferior o LBTS (*Lower Bound Time Stamp*) y eventos que abarcan varios procesos a través de la sincronización conservativa. La plataforma aborda específicamente entornos de procesamiento de flujos en la nube, lo que la hace relevante para aplicaciones que manejan grandes volúmenes de datos en tiempo real.

RStorm: Desarrollo y prueba de algoritmos de streaming en R

Los autores **Muthukrishnan, P. y Fadnis**, nos presentan una herramienta diseñada para el desarrollo y la prueba de algoritmos de procesamiento de datos en tiempo real utilizando el lenguaje de programación R llamada RStorm [56]. RStorm es un paquete de simulación destinado a ayudar a los desarrolladores a analizar y evaluar sus algoritmos de streaming fácilmente y sin las dificultades de la implementación real en una plataforma de

procesamiento de streams dada. RStorm recurre a la terminología y conceptos de Storm, proporcionando una representación gráfica de los algoritmos de streaming. Sin embargo, este paquete analiza el algoritmo y no incluye los costos de hardware asociados (como la comunicación, múltiples hilos por nodo, etc).

Simulación Paralela de Sistemas de Procesamiento de Streams Distribuidos a Gran Escala

En el artículo presentado por los autores **Fang, Z. y Yu, H.**, se aborda la problemática de simular sistemas de procesamiento de streams distribuidos a gran escala. Para abordar este desafió, los autores proponen un enfoque paralelo que aprovecha la computación distribuida para acelerar la simulación [57]. La arquitectura propuesta incluye componentes específicos para la generación de eventos, la sincronización y comunicación entre nodos de simulación. Además de exponer técnicas para optimizar el rendimiento y la escalabilidad de las simulaciones paralelas, como la partición del espacio de eventos y la minimización de la comunicación entre nodos.

Los autores **Ma, Y. y Rundensteiner, E. A.** abordan las mismas problemáticas presentes en la simulación de sistemas de procesamiento de streams distribuidos, pero desde una perspectiva centrada en la escalabilidad y eficiencia [58]. Este enfoque se basa en el uso de técnicas de muestreo adaptativo y la generación de muestras de eventos para la reducción de la carga computacional. Además, se presentan algoritmos para optimizar el rendimiento y escalabilidad de las simulaciones distribuidas, como la adaptación dinámica de la tasa de muestreo y la paralelización de tareas.

Protocolo de simulación asíncrona para plataformas de procesamiento de streams

En el estudio llevado a cabo por **Gil-Costa, M, et al.**, en donde proponen el uso de un protocolo de simulación asíncrona capaz de ejecutarse en la plataforma de procesamiento de streams s4 [59]. La simulación asíncrona es crucial para capturar con precisión el comportamiento de sistemas complejos que procesan flujos de datos en tiempo real.

Este simulador fue desarrollado para simular la ejecución de consultas de usuario en un motor de búsqueda web. El simulador propuesto controla el avance del tiempo virtual en cada Proceso Lógico o LP(Logic Process, por sus siglas en inglés), empleando dos barreras. La primera barrera consiste en una ventana de tiempo que permite el procesamiento de eventos con marcas de tiempo dentro de la ventana de tiempo. La segunda barrera se basa en una barrera de tiempo de oráculo utilizada para calcular de manera adaptativa el valor de la ventana de tiempo en toda la simulación. Los dos mecanismos basados en barreras reducen el número de eventos rezagados, en una simulación asíncrona, en todos los LP. Aunque el caso de estudio presente en esta investigación simula un motor de búsqueda web, los autores mencionan la posibilidad de modelar y simular una plataforma de procesamiento de streams utilizando este simulador.

Apache Heron: Baja Latencia en el Procesamiento de Streams

Los autores **Kamburugamuve, Supun, et. al.** proponen Apache Heron como un framework robusto y escalable de procesamiento de streamings distribuidos diseñado para manejar grandes volúmenes de datos en tiempo real [60]. Apache Heron propone una nueva arquitectura para el procesamiento de datos de streamings basada en un diseño híbrido con algunas partes críticas para el rendimiento escritas en C++ y otras escritas en Java. Esta arquitectura permite la integración de directa de mejoras de alto rendimiento en lugar de pasar por envoltorios (wrappers) nativos tales como la interfaz nativa de Java (JNI, por sus siglas en inglés). Los autores además abordan la necesidad del uso de una interfaz TCP para la interconexión de alto rendimiento, teniendo un desempeño bajo en comparación con una implementación nativa. Para resolver estos problemas de hardware Heron propone la integración con Infiniband e Intel Omni-Path para interconectarlas con Apache Heron, con el fin de acelerar la comunicación.

Modelado y simulación de aplicaciones Spark y YARN utilizando lenguajes específicos de dominio

Los autores **Kross, D. y Krcmar, H.** abordan la necesidad de herramientas que permitan facilitar la representación y simulación de aplicaciones Spark y YARN en entornos de Big Data. En base a esta necesidad proponen un lenguaje específico de dominio o DSL(Domain-Specific Languages, por sus siglas en inglés) para modelar las características de las aplicaciones Spark y YARN. Al emplear DSL, se busca ofrecer una forma más intuitiva y eficiente de describir las características y el comportamiento de las aplicaciones, extrayendo automáticamente las especificaciones del modelo y transformándolas en una herramienta de evaluación. Mejorando significativamente el proceso de desarrollo y optimización de estas aplicaciones.

Capítulo 3

Definición del Problema y Análisis de Requerimientos

El propósito de este capítulo es profundizar en el problema presentado en la sección anterior (ver sección1), proporcionando un nivel adicional de detalle. Posteriormente, se presenta la propuesta de solución y se discutirá la relevancia que esta tiene en el ámbito científico y/o social.

3.1. Contexto del Problema

Las aplicaciones de simulación de plataformas de procesamiento de streams, como Apache Storm, desempeñan un papel crucial en la evaluación y optimización de sistemas de procesamiento de datos en tiempo real [61]. Estas herramientas permiten a los desarrolladores modelar y simular el comportamiento de diversas arquitecturas de procesamiento de streams en entornos controlados [62]. Al proporcionar una representación virtual de los componentes del sistema y sus interacciones, estas aplicaciones facilitan la identificación de cuellos de botella, la evaluación del rendimiento y la optimización del flujo de datos en tiempo real [63]. Además, su capacidad para manejar grandes volúmenes de datos y procesar eventos en tiempo real las convierte en herramientas indispensables para sectores como la analítica de datos, la IoT y la detección de anomalías [64]. En este contexto, exploraremos los desafíos y oportunidades asociados con el desarrollo, despliegue y escalado de aplicaciones tipo Apache Storm para la simulación de plataformas de procesamiento de streams.

Las aplicaciones de simulación de plataformas de procesamiento de streams se enfrentan a desafíos complejos en términos de escalabilidad y tolerancia a fallos. A medida que los sistemas de procesamiento de datos en tiempo real se vuelven más grandes y complejos, la capacidad de escalar horizontalmente y manejar flujos de datos de alta velocidad se convierte en una necesidad crítica. Además, la necesidad de garantizar la disponibilidad

continua y la integridad de los datos en entornos distribuidos plantea desafíos adicionales en términos de tolerancia a fallos y recuperación ante errores [65].

Uno de los principales desafíos en el desarrollo de aplicaciones Apache Storm es la complejidad asociada con el modelado y la implementación de topologías de procesamiento de streams eficientes. El diseño de topologías que puedan manejar flujos de datos de manera eficiente, distribuir la carga de trabajo de manera uniforme y adaptarse dinámicamente a cambios en los patrones de tráfico puede resultar complicado y requerir un profundo entendimiento de los principios de procesamiento de streams y las características del sistema subyacente [66].

Además de los desafíos técnicos, también existen consideraciones importantes relacionadas con la administración y el monitoreo de aplicaciones Apache Storm en producción. La capacidad de monitorear el rendimiento, la utilización de recursos y el estado de las topologías de procesamiento de streams en tiempo real es fundamental para garantizar la fiabilidad y el rendimiento óptimo del sistema. Por lo tanto, es crucial contar con herramientas y prácticas efectivas de monitoreo y gestión para identificar y mitigar problemas de manera proactiva y garantizar la disponibilidad continua de las aplicaciones críticas [67].

Estos desafíos plantean oportunidades significativas para la investigación y la innovación en el campo de las plataformas de procesamiento de streams. Al abordar los desafíos clave asociados con el desarrollo, despliegue y operación de aplicaciones Apache Storm, es posible mejorar la eficiencia, la escalabilidad y la confiabilidad de los sistemas de procesamiento de datos en tiempo real, lo que puede conducir a avances significativos en una amplia gama de aplicaciones y sectores industriales [68].

3.2. Definición del Problema

El trabajo se centra en abordar los desafíos identificados en la aplicación SimStream [69]. Esta aplicación, desarrollada en C++, se utiliza para la simulación y análisis de resultados a partir de una plataforma de procesamiento de streams en tiempo real. Durante la revisión literaria y análisis de la aplicación existente, se logró identificar una serie de mejoras tecnológicas.

3.2.1. Complejidad de Configuración y Uso de Aplicación

La complejidad en la configuración y uso de una aplicación puede ser un obstáculo significativo para los usuarios, especialmente si la interfaz no es intuitiva o si la documentación es insuficiente [70]. Cuando los usuarios encuentran dificultades para configurar o utilizar la aplicación de manera efectiva, esto puede limitar su utilidad práctica y afectar la adopción y satisfacción del usuario [71]. Por ejemplo, si un usuario enfrenta dificultades para comprender la configuración de servicios o la interacción con la aplicación debido a

una documentación poco clara, es probable que busque alternativas más sencillas y fáciles de usar.

3.2.2. Gestión eficiente del flujo de datos y comunicación entre componentes

La falta de un sistema gestor de colas de simulación puede ser un problema significativo en este contexto. Puesto que estos sistemas, como RabbitMQ, desempeñan un papel crucial en la gestión eficiente de los flujos de datos y la comunicación entre los distintos componentes de la aplicación [72]. Sin un sistema de este tipo, la aplicación puede enfrentarse a desafíos relacionados con la sincronización de eventos, la gestión de la concurrencia y la escalabilidad. Además, la ausencia de un sistema gestor de colas puede dificultar la implementación de patrones de diseño de software como el modelo de productor-consumidor, lo que podría impactar negativamente en la eficiencia y la fiabilidad del sistema en general [71]. En resumen, la carencia de un sistema gestor de colas de simulación puede limitar las capacidades de procesamiento y la escalabilidad de la aplicación, así como aumentar la complejidad del código y la dificultad en la gestión de los flujos de datos en tiempo real [70].

3.2.3. Portabilidad y Despliegue de Aplicación

El desarrollo de una aplicación de simulación de plataformas de procesamiento de streams en C++ conlleva desafíos considerables en cuanto a su portabilidad y despliegue, aspecto que puede dificultar la migración entre distintos sistemas operativos y plataformas de hardware [73]. La presencia de dependencias específicas del sistema y variaciones en las configuraciones de los entornos de desarrollo y producción agravan esta situación, limitando la flexibilidad y alcance de la aplicación. La configuración manual de las dependencias y la gestión de versiones de software pueden ser propensas a errores y consumir recursos, lo que impacta negativamente en la eficiencia del proceso de desarrollo y despliegue [74]. Estas dificultades, además, pueden obstaculizar la capacidad de la aplicación para adaptarse a las exigencias cambiantes del usuario y del mercado, limitando su escalabilidad y capacidad de respuesta ante nuevas demandas.

3.3. Solución Propuesta

Para abordar las problemáticas previamente mencionadas (ver secciones 3.2.1,3.2.2,3.2.3), se propone desarrollar e implementar un entorno web destinado a simular plataformas de procesamiento de streams. Uno de los principales objetivos de este entorno es simplificar el proceso de construcción y simulación de modelos de

procesamiento de plataformas de streams para los usuarios. Este objetivo se alcanza mediante la abstracción y el encapsulamiento de las complejidades relacionadas con la creación de modelos concurrentes o paralelos de plataformas de streams (ver Figura 3.1).

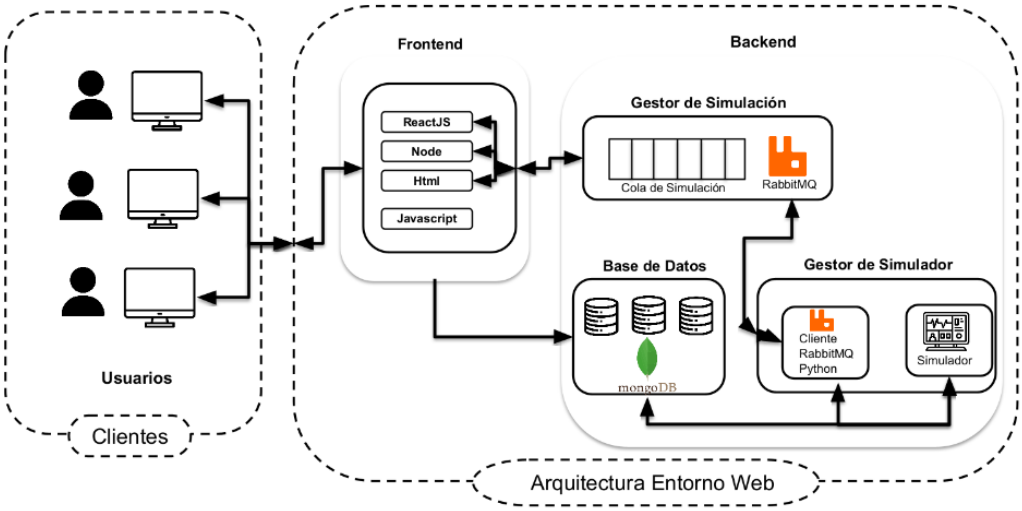

Figura 3.1: Arquitectura del entorno web para la simulación de plataformas de procesamiento de streams

En la implementación de la solución propuesta, se prioriza la adopción de una arquitectura basada en servicios (*SOA*) [75]. Este enfoque de desarrollo de software se sustenta en la creación de componentes de software denominados servicios. Estos servicios son independientes entre sí y pueden interactuar, combinarse o reutilizarse con otros servicios desarrollados en diversos lenguajes de programación para ejecutar tareas más complejas [76](Ver Figura 3.2). Al favorecer esta aproximación, se reduce la dependencia del sistema en su conjunto al abstraer los detalles de implementación o código fuente de cada servicio [77]. Esto conlleva a una mayor flexibilidad y escalabilidad en el desarrollo y mantenimiento del software, permitiendo una integración más fluida de nuevas funcionalidades y la adaptación a cambios en los requisitos del negocio [77].

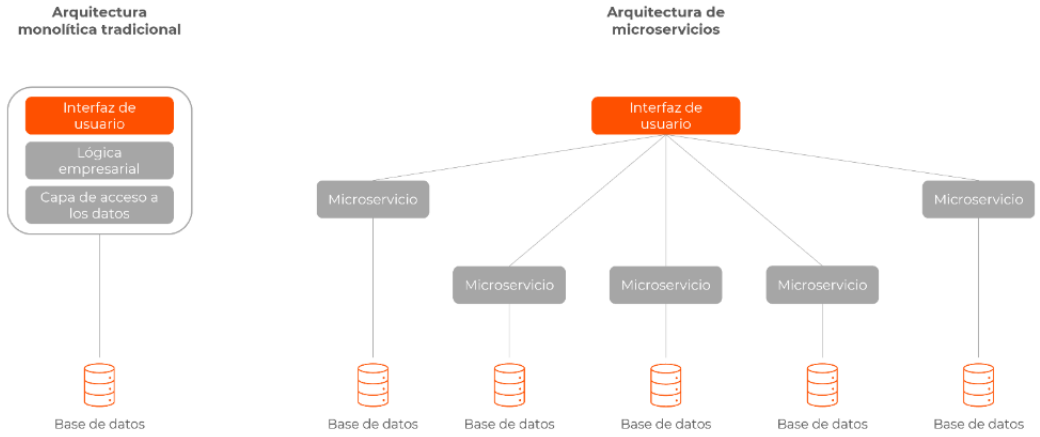

Figura 3.2: Arquitectura enfocada en micro servicios

El enfoque principal de utilizar esta arquitectura radica en la generación de una serie de servicios mediante la aplicación de contenedores disponibles en la plataforma de software Docker [16]. Uno de los principales contenedores generados a mencionar es el servicio de simulación de modelos de procesamiento de streams, el cual se basa en el simulador SimStream desarrollado en el lenguaje de programación C++ [69]. Este servicio recibe como entrada los modelos de simulación de plataformas de streams y lleva a cabo la simulación de los mismos, proporcionando los resultados de estas simulaciones para su posterior análisis.

Se establece un servicio especializado para administrar las colas de simulación de los modelos generados por los usuarios, además de encargarse de la recopilación y el almacenamiento de datos tanto de los usuarios como de los modelos [78]. Paralelamente, se iniciará una investigación y desarrollo dirigido hacia una arquitectura orientada a eventos que capitalice el sistema de mensajería RabbitMQ [72], con el propósito de facilitar la comunicación entre componentes y servicios distribuidos en diversas áreas del sistema de forma asíncrona.

A esto se suma el uso de Express como servicio para proporcionar una plataforma robusta y flexible para el desarrollo de aplicaciones web y APIs de manera eficiente y escalable [79]. A través de la implementación de rutas parametrizadas para admitir operaciones CRUD, validaciones de esquemas de solicitud por parte del usuario u otros servicios, gestión de middleware para manejar la lógica de la aplicación de forma modular y la configuración de seguridad para proteger contra vulnerabilidades [79]. Facilitando la gestión de sesiones de usuario y el manejo de errores de manera efectiva.

Por último, se realizarán pruebas exhaustivas de funcionalidad, rendimiento y usabilidad sobre el entorno web de simulación, en colaboración con modelos de simulación

previamente validados [80]. El propósito principal es obtener métricas comparativas en eficiencia, rendimiento, uso de memoria, así como verificar y validar la seguridad del sistema, entre otros aspectos [80].

3.4. Objetivos

3.4.1. Objetivo General

- Diseñar, Desarrollar e implementar un entorno web centrado en el uso de una arquitectura de servicios y el sistema de contenedores Docker como ejes principales, con el objetivo de resolver los problemas de complejidad presentes en la simulación de plataformas de procesamiento de streams.

3.4.2. Objetivos Específicos

- Diseñar una arquitectura de servicios (SOA) adecuada para el entorno de simulación de plataformas de procesamiento de streams, permitiendo el desacoplamiento y reutilización de componentes.
- Definir y modelar los elementos clave del dominio de negocio, incluyendo entidades, agregados y servicios de dominio, utilizando técnicas de modelado de dominio y patrones de diseño de DDD (Driven-Domain-Design), para asegurar una representación precisa y coherente del dominio en la arquitectura y el código de la aplicación.
- Desarrollar módulos y componentes específicos para el entorno web de simulación, incluyendo la interfaz de usuario, la gestión de simulaciones, la visualización de resultados y la integración con herramientas de terceros.
- Implementar el entorno de simulación utilizando Docker como plataforma de organización de contenedores, asegurando la portabilidad y la consistencia del entorno de ejecución.
- Desarrollar un proceso de transformación y generación de contenedores a partir del simulador Storm, con la capacidad de operar como un servicio funcional y autónomo.
- Diseñar e implementar un conjunto completo de rutas para la aplicación web, modularizando el código con middleware de enrutamiento y garantizando la seguridad y la protección contra errores mediante el uso de prácticas de seguridad y middleware de manejo de errores de Express.

- Configurar y gestionar las rutas parametrizadas para admitir operaciones CRUD, proporcionando una interfaz de programación de aplicaciones (API) que permita manipular datos a través de HTTP utilizando los métodos estándar (GET, POST, PUT, DELETE).
- Implementar un sistema de gestión de colas utilizando RabbitMQ como broker de mensajes para el entorno de simulación, con el fin de asegurar una distribución eficiente y confiable de los mensajes entre los diferentes componentes del sistema simulado.
- Evaluar y validar el entorno de simulación desarrollado mediante pruebas de funcionalidad, rendimiento y usabilidad, utilizando casos de prueba representativos y métricas relevantes para medir su eficacia y fiabilidad.
- Documentar el proceso de diseño, desarrollo e implementación del entorno de simulación, así como los resultados obtenidos en términos de funcionalidad, rendimiento y usabilidad, en un informe técnico detallado y enriquecido con ejemplos y casos de uso.

3.5. Metodología

3.5.1. Metodología de Desarrollo

Para la ejecución de este trabajo de título, se adoptará la metodología de desarrollo ágil Scrum [17]. En Scrum, cada ciclo de desarrollo se organiza en sprints, que son períodos de tiempo fijos y cortos, generalmente de una a cuatro semanas de duración. Durante la planeación de cada sprint, el equipo selecciona una lista de elementos de trabajo del producto y se compromete a completarlos antes del final del sprint. Estos elementos de trabajo se desglosan en tareas más pequeñas y se estiman en función de la complejidad y el esfuerzo requerido. Durante el sprint, el equipo se reúne diariamente en reuniones llamadas scrums, donde cada miembro comparte su progreso, identifica posibles obstáculos y coordina las tareas restantes. Al final del sprint, se lleva a cabo una retrospectiva para revisar lo que salió bien, lo que no salió tan bien y cómo se puede mejorar en el próximo sprint. Este enfoque iterativo permite que el equipo entregue un Producto Mínimo Viable (MPV) al final de cada sprint, lo que proporciona un valor tangible al cliente de manera incremental [17](ver Figura 3.3).

Figura 3.3: Metodología Desarrollo Scrum.

Después de varias iteraciones de sprints, se alcanza la fase de entrega, donde se presenta al cliente el producto funcional resultante de los sprints realizados [17]. Esta entrega incremental permite al cliente validar el producto en etapas tempranas y proporciona retroalimentación valiosa para orientar el desarrollo futuro. Una vez entregado el producto, se inicia la fase de soporte y mantenimiento, donde se brinda asistencia continua al cliente para garantizar el funcionamiento óptimo del producto en producción [81]. Esto implica la corrección de errores, la aplicación de parches de seguridad y la implementación de mejoras y actualizaciones según sea necesario. Durante esta fase, el equipo de desarrollo también puede continuar trabajando en nuevos sprints para agregar funcionalidades adicionales o realizar mejoras en el producto existente. A continuación se presenta una figura que resume los conceptos discutidos anteriormente:

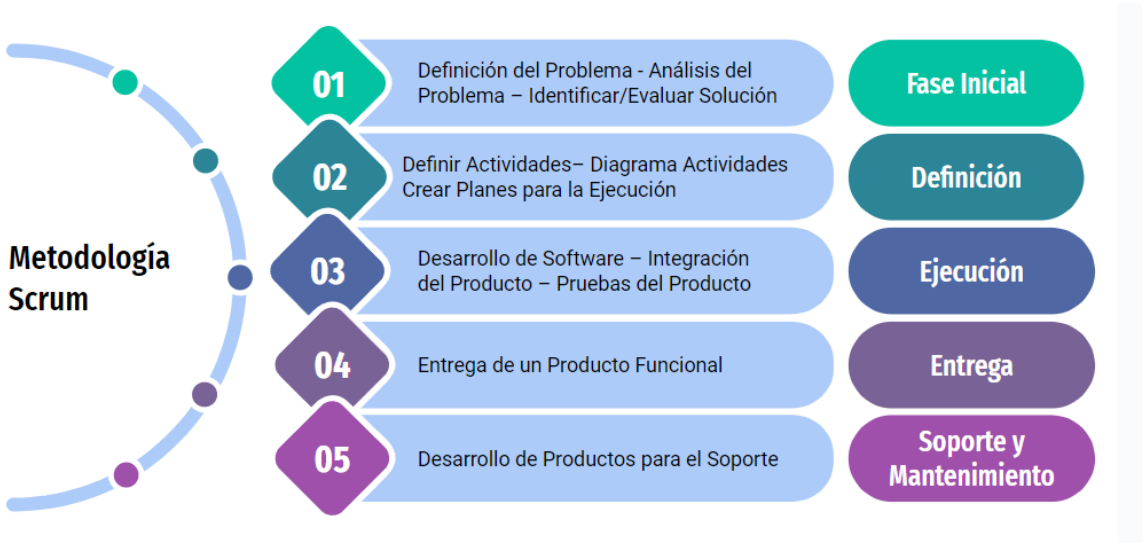

Figura 3.4: Fases de una Metodología Scrum.

3.5.2. Planificación de Desarrollo basada en Scrum

A continuación se presenta una planificación adaptada a la metodología de desarrollo Scrum, esta planificación adaptada de Scrum permite una entrega incremental de funcionalidades y una mayor flexibilidad para adaptarse a los cambios en los requisitos del proyecto a lo largo del tiempo:

- Sprint 1: Definición del Alcance y Diseño Inicial
  - Definir requisitos iniciales y alcance del proyecto.
  - Realizar análisis preliminar del dominio de negocio.
  - Diseñar arquitectura inicial del sistema y planificar las iteraciones futuras.
- Sprint 2: Implementación de la Arquitectura de Servicios
  - Implementar la arquitectura de servicios (SOA) básica.
  - Desarrollar modelos iniciales de dominio y servicios de dominio.
- Sprint 3: Desarrollo de Componentes
  - Desarrollar módulos y componentes específicos para el entorno web de simulación.
  - Implementar rutas básicas para la aplicación web utilizando Express.js.

    - Validar el ordenamiento cronológico de los eventos en la estructura de datos de implementada en la librería.

- Sprint 4: Implementación del Entorno de Simulación
    - Implementar el entorno de simulación utilizando contenedores Docker.
    - Desarrollar procesos básicos de transformación y generación de contenedores a partir del simulador Storm.
- Sprint 5: Configuración y Gestión de Rutas
    - Configurar y gestionar las rutas parametrizadas en Express.js.
    - Implementar la gestión básica de colas utilizando RabbitMQ como broker de mensajes.
- Sprint 6: Pruebas y Validación
    - Realizar pruebas de funcionalidad, rendimiento y usabilidad en el entorno de simulación.
    - Evaluar y validar el sistema mediante casos de prueba representativos.
- Sprint 7: Ajustes Finales y Documentación
    - Realizar ajustes finales en el sistema basados en los resultados de las pruebas y evaluaciones.
    - Documentar el entorno de simulación desarrollado y prepararse para la entrega.

3.6. Especificación de Requerimientos

3.6.1. Requerimientos Funcionales

Los requerimientos funcionales son especificaciones detalladas de las funciones y comportamientos que un sistema debe ser capaz de realizar [82]. Estos requerimientos no solo definen las acciones específicas que el sistema debe tomar en respuesta a las entradas del usuario, sino también las salidas o resultados esperados. Además, en un contexto más amplio, los requerimientos funcionales pueden incluir la descripción de los procesos comerciales que el sistema debe automatizar o mejorar, así como los casos de uso que representan las interacciones típicas entre el usuario y el sistema [83]. Estas especificaciones pueden expresarse en forma de historias de usuario, diagramas de flujo o casos de uso, y sirven como base para el diseño y la implementación del sistema.

Es importante destacar que los requerimientos funcionales deben ser claros, precisos y verificables [82]. Deben ser comprensibles tanto para los desarrolladores que trabajarán en la implementación del sistema como para los clientes o usuarios finales que utilizarán el sistema. Además, los requerimientos funcionales deben ser flexibles y estar sujetos a cambios a medida que evolucionan las necesidades del negocio o se identifican nuevos requisitos durante el proceso de desarrollo. A continuación se detallan algunos de los requerimientos funcionales de este trabajo de título:

- Inicio de Sesión:
  - El sistema debe permitir a los usuarios iniciar sesión utilizando un nombre de usuario y contraseña válidos.
  - Se debe verificar la autenticidad de las credenciales del usuario antes de permitir el acceso al sistema.
- Registro de Usuarios:
  - Los usuarios deben poder registrarse proporcionando información básica como nombre, dirección de correo electrónico y contraseña.
  - El sistema debe validar que la dirección de correo electrónico proporcionada sea única en la base de datos.
- Dashboard de Usuario:
  - Después de iniciar sesión, los usuarios deben ser redirigidos a una dashboard donde puedan realizar diversas operaciones.
  - La dashboard debe permitir a los usuarios realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) sobre modelos de simulación de plataformas de stream.
- Operaciones CRUD sobre Modelos de Simulación:
  - Los usuarios deben poder crear nuevos modelos de simulación proporcionando la información necesaria.
  - Deben poder ver una lista de todos los modelos de simulación existentes y editarlos si es necesario.
  - Los usuarios deben tener la opción de eliminar modelos de simulación que ya no sean necesarios.
- Almacenamiento en MongoDB:

    - Todos los datos de usuario, incluidos los detalles del registro e información de los modelos de simulación, incluyendo datos asociados a los resultados de las simulaciones realizadas deben almacenarse en bases de datos MongoDB para su posterior recuperación, procesamiento y posterior análisis.

- Servicio de Gestión de Colas de Simulación:

    - El sistema debe interactuar con un servicio de gestión de colas de simulación proporcionado por RabbitMQ.
    - Debe enviar los modelos de simulación creados por los usuarios a la cola de simulación para su procesamiento.

- Servicio de Simulación Desarrollado en C++:

    - El sistema debe incluir un servicio de simulación desarrollado en C++, modificado para funcionar como un contenedor de Docker.
    - Este servicio debe ser capaz de procesar los modelos de simulación recibidos desde la cola de simulación y generar simulaciones y resultados adecuados.

3.6.2. Requerimientos No Funcionales

Los requerimientos no funcionales, también conocidos como atributos de calidad o características de calidad, describen las cualidades globales que debe tener un sistema más allá de sus funcionalidades específicas. Estos requerimientos definen aspectos como el rendimiento, la seguridad, la usabilidad y la escalabilidad del sistema, que son cruciales para su éxito y aceptación en el entorno operativo [84].

Estos requerimientos no funcionales son igualmente importantes que los requerimientos funcionales, ya que contribuyen a la experiencia del usuario, la eficiencia del sistema y su capacidad para cumplir con los objetivos del negocio. Aunque suelen ser más difíciles de medir y cuantificar que los funcionales, son igualmente críticos para el éxito del proyecto de software en su conjunto [82]. A continuación se detallan algunos de los requerimientos no funcionales de este trabajo de título:

- Seguridad:

    - Los datos de los usuarios y los modelos de simulación deben almacenarse de forma segura y estar protegidos contra accesos no autorizados.
    - El sistema debe implementar medidas de seguridad como cifrado de contraseñas y protección contra ataques de inyección de código.

- Rendimiento:

  - El sistema debe ser capaz de manejar una carga de trabajo significativa sin degradación del rendimiento.
  - Las operaciones CRUD y la simulación de modelos de stream deben realizarse de manera eficiente y con tiempos de respuesta aceptables.

- Escalabilidad:

  - El sistema debe ser escalable horizontalmente para poder manejar un aumento en la cantidad de usuarios y modelos de simulación.
  - Debe ser fácil agregar nuevos recursos al sistema, como nuevas instancias o servicios de contenedores Docker, para aumentar su capacidad.

- Disponibilidad:

  - El sistema debe estar disponible las 24 horas del día, los 7 días de la semana, con un tiempo de inactividad mínimo planificado para mantenimiento.
  - Se deben implementar medidas de redundancia y respaldo para garantizar la disponibilidad continua de los datos y servicios críticos.

- Interfaz de Usuario:

  - La interfaz de usuario desarrollada en React JS debe ser intuitiva y fácil de usar para los usuarios, con un diseño atractivo y responsive que se adapte a diferentes dispositivos y tamaños de pantalla.
  - Debe enviar los modelos de simulación creados por los usuarios a la cola de simulación para su procesamiento.

- Compatibilidad:

  - El sistema debe ser compatible con una amplia gama de navegadores web modernos, incluidos Google Chrome, Mozilla Firefox, Safari y Microsoft Edge.
  - La aplicación también debe ser compatible con diferentes sistemas operativos, como Windows, macOS y Linux.

- Mantenibilidad:

  - El código del sistema debe seguir buenas prácticas de desarrollo de software, ser fácilmente legible y modular, y estar bien documentado para facilitar su mantenimiento y futuras actualizaciones.

3.7. Funcionalidades del Sistema

La sección de funcionalidades del sistema ofrece una visión detallada de las capacidades y características que proporciona la arquitectura del entorno web para la simulación de plataformas de procesamiento de streams. Desde la gestión de usuarios hasta la generación de informes, este sistema está diseñado para cubrir una amplia gama de necesidades. Entre las principales funcionalidades se incluyen la creación y edición de perfiles de usuario [85], la administración de contenido [86], la programación de eventos [87], la generación de informes y análisis de datos [88], y la integración con sistemas externos [89]. Cada funcionalidad se ha desarrollado pensando en la facilidad de uso y en la capacidad de adaptarse a las necesidades específicas de los usuarios. A continuación, se detallan las funcionalidades clave que hacen de este sistema una herramienta poderosa y versátil para la gestión y análisis de datos.

3.7.1. Diagramas de Casos de Uso

Actores

- **Usuario**: El actor ”Usuario” representa a cualquier persona que acceda al sistema como cliente. Este usuario tiene la capacidad de iniciar sesión en el sistema para acceder a sus funcionalidades, registrarse como nuevo usuario si no tiene una cuenta, gestionar los modelos de simulación de plataformas de stream, lo que incluye crear, guardar, eliminar y visualizar detalles de los modelos existentes. Además, puede generar informes detallados sobre los resultados de las simulaciones realizadas.

- **Administrador**: El actor ”Administrador” es responsable de administrar y supervisar el funcionamiento del sistema. Tiene las mismas capacidades que un usuario normal, como iniciar sesión y gestionar modelos de simulación, pero además cuenta con funcionalidades adicionales. El administrador puede simular modelos de manera directa, gestionar usuarios (ver lista de usuarios, editar información de usuarios existentes, desactivar o eliminar cuentas de usuarios si es necesario) y realizar cualquier acción necesaria para garantizar el correcto funcionamiento del sistema y la seguridad de los datos.

3.7.2. Casos de Uso

A continuación, se presenta el diagrama de caso de uso de la solución propuesta (Ver Figura 3.5). Este diagrama proporciona una visión general de las funcionalidades clave que ofrece el sistema y cómo interactúan los actores con el sistema para lograr sus objetivos. El diagrama está estructurado en dos partes principales: ”Gestión de Modelos de Simulación” y ”Administración del Sistema”.

Respecto al caso de uso gestión de modelos de simulación, tanto usuarios normales como administradores tienen la capacidad de realizar diversas acciones, como iniciar sesión, registrarse, gestionar modelos de simulación y generar informes basados en los resultados de las simulaciones realizadas. Cabe recalcar que el caso de uso llamado ”simular modelo” esta directamente relacionado con la generación de modelos de simulación y la carga de estos mismos para su posterior simulación.

Por otra parte, el caso de uso administración del sistema se reserva exclusivamente para los usuarios administradores, estos tienen la responsabilidad de gestionar usuarios y administrar el sistema en su conjunto. Esto incluye acciones como la gestión de la información de los usuarios, así como realizar tareas administrativas relacionadas con el sistema (configuración y mantenimiento).

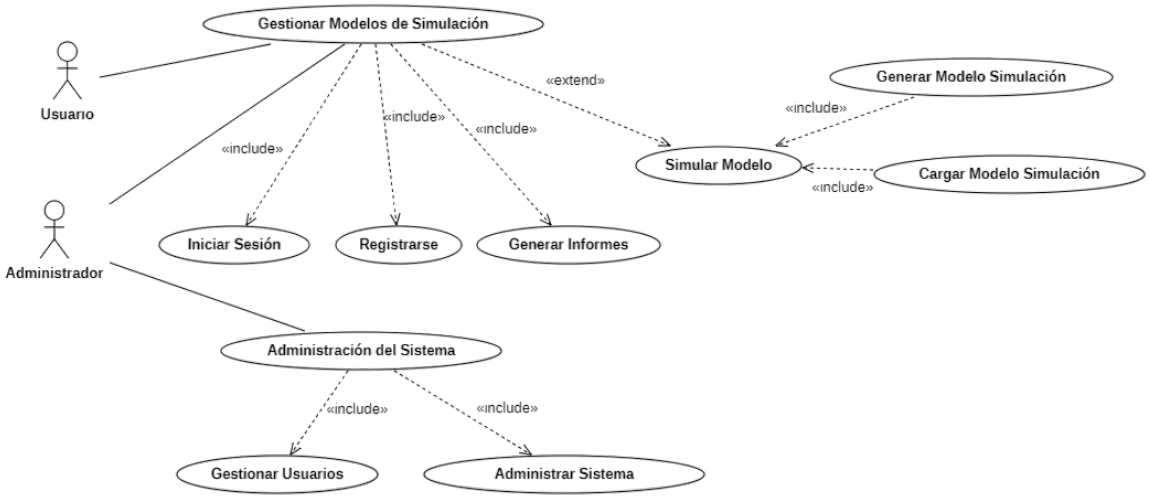

Figura 3.5: Diagrama General de Caso de Uso.

3.7.3. Diagramas de Estado

A continuación se presenta el diagrama de estado para usuarios (Ver Figura 3.6), en donde se muestran los estados y transiciones posibles del sistema, basado en el caso de uso anteriormente proporcionado (3.7.1). Los estados incluyen lo siguientes elementos:

- Usuario No Autenticado
- Usuario Autenticado
- Gestionar Modelos
- Simulando Modelo

- Finalizado

Las transiciones entre estados están etiquetadas en conjunto con las acciones que desencadenan un cambio de estado, como iniciar sesión, cerrar sesión, simular modelos, generar informes, y otras acciones relacionadas con la gestión de modelos y simulaciones pertinentes.

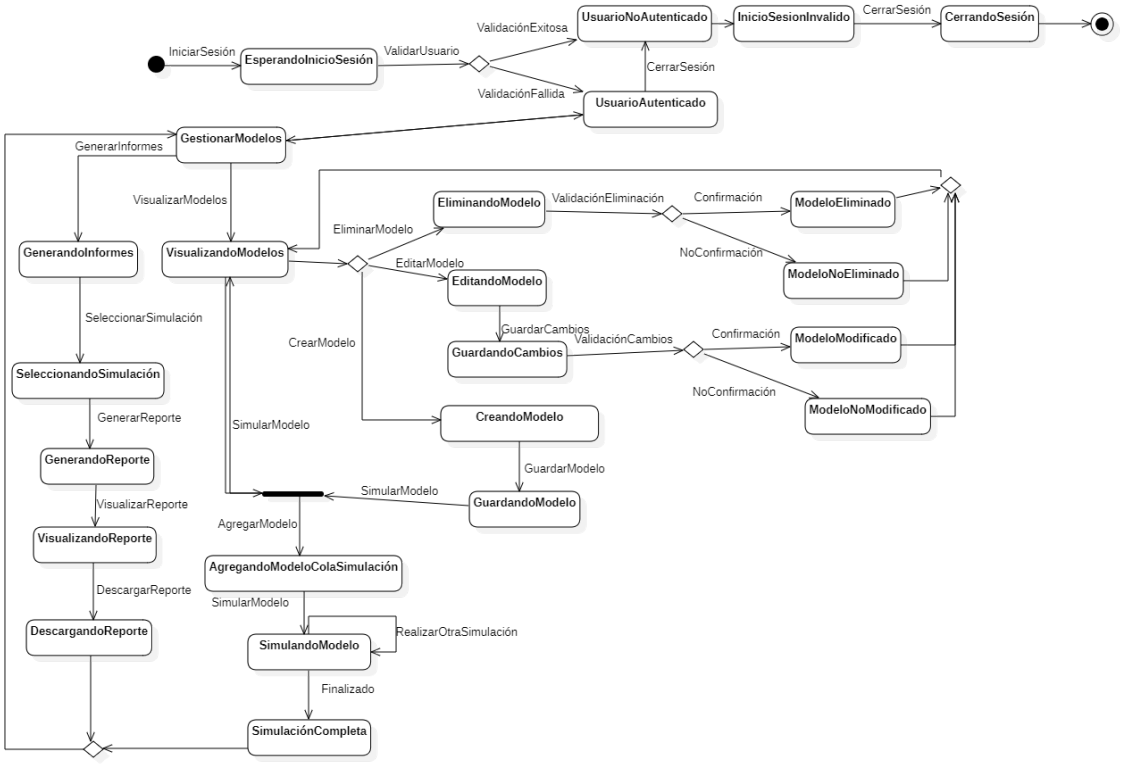

Figura 3.6: Diagrama de estado para usuarios.

3.7.4. Modelo Conceptual

Capítulo 4

Diseño de la solución

Este capítulo aborda en detalle la solución propuesta para el problema planteado, incluyendo los objetivos generales y específicos, la propuesta de solución, las formulaciones utilizadas con sus respectivas explicaciones, los algoritmos diseñados y la metodología de investigación aplicada.

4.1. Diseño Arquitectónico

4.1.1. Tecnologías utilizadas

A continuación, se presenta la asociación de cada uno de los requerimientos no funcionales (ver sección 3.6.2) con las tecnologías correspondientes utilizadas para el desarrollo del trabajo de título. Estas tecnologías han sido seleccionadas cuidadosamente con la finalidad de abordar cada uno de los aspectos claves identificados en los requerimientos, garantizando así un sistema seguro, eficiente, escalable, disponible, fácil de usar, compatible y fácil de mantener en el tiempo. Cada tecnología desempeña un papel crucial en la creación de una solución integral que cumpla con las expectativas de los usuarios y requerimientos de negocio.

- Seguridad
    - Docker: Capaz de garantizar la seguridad del entorno de ejecución, mediante la encapsulación de las aplicaciones en contenedores, controles de acceso y autenticación y monitorización continua de la actividad de contenedores [90].
    - RabbitMQ: Implementación de políticas de seguridad para restringir el acceso a las colas de mensajes y garantizar que solo los usuarios autorizados puedan enviar y recibir mensajes [91].

  - React JS: Implementación de medidas de seguridad en el frontend para la protección contra ataques scripting maliciosos y evitar exposición de datos sensibles [92].
  - Node JS: Uso de bibliotecas de cifrado y autenticación para garantizar que contraseñas y otros datos sensibles se almacenen de forma segura y se protejan contra ataques de inyección de código [93].
  - MongoDB: Implementar autenticación y autorización basadas en roles para controlar el acceso a las bases de datos y garantizar tanto el acceso como la manipulación de datos [94].
  - Express JS: Implementación de middleware de seguridad para protección contra ataques de scripting malicioso y prevenir la exposición de vulnerabilidades en la aplicación [95].

- Rendimiento

  - Docker: Optimización del rendimiento del sistema a través de una infraestructura ligera y portátil que minimiza el overhead de virtualización y maximiza la eficiencia de recursos [96].
  - RabbitMQ: Proporciona escalamiento horizontal para el manejo de cargas de trabajo significativas, distribuyendo mensajes entre múltiples nodos y balanceando la carga eficientemente [].
  - Node JS: Alta capacidad para el manejo de solicitudes asíncronas y no bloqueantes, permitiendo un rendimiento óptimo incluso bajo cargas de trabajo pesadas [97].
  - Express JS: Optimización de rendimiento mediante el uso de middleware eficiente e implementación de patrones de diseño que minimizan el tiempo de procesamiento de las solicitudes [98].

- Escalabilidad

  - Docker: Facilita la escalabilidad horizontal al permitir una implementación rápida de nuevos contenedores y distribución de cargas de trabajo entre múltiples instancias [99].
  - Node JS: Altamente escalable debido a su modelo de E/S no bloqueante, permitiendo manejar grandes cantidades de conexiones simultáneas sin degradación del rendimiento [100].

- Disponibilidad

  - Docker: Proporciona una infraestructura flexible y resistente que facilita la implementación de prácticas de alta disponibilidad, como la redundancia y el balanceo de carga [101].
  - RabbitMQ: Posibilidad de replicar mensajes entre múltiples nodos e implementación de políticas de recuperación ante fallos, minimizando el tiempo de inactividad [102].
  - Express JS: Uso de middleware para el manejo de errores e implementación de estrategias de recuperación ante fallos, garantizando una respuesta robusta a situaciones imprevistas [103].

- Compatibilidad

  - Docker: Mejora la compatibilidad del sistema, proporcionando una infraestructura unificada y coherente que facilita el despliegue de la aplicación en una variedad de entornos de ejecución [104].
  - RabbitMQ: Proporciona una capa de abstracción entre diferentes componentes del sistema, permitiendo una comunicación eficiente entre ellos independientemente de la tecnología utilizada.
  - Express JS: Proporciona una API RESTful que sigue estándares web abiertos y bien establecidos, facilitando la interoperabilidad con diferentes clientes y servicios [105].

- Mantenimiento

  - Docker: Proporciona un entorno de desarrollo consistente y reproducible, facilitando la gestión de versiones del código y dependencias [106].
  - MongoDB: Proporciona una base de datos flexible y fácil de usar que permite realizar cambios en el esquema de datos de manera rápida y sin problemas, facilitando la adaptación del sistema a medida que cambian los requisitos y necesidades de negocio [107].
  - Express JS: Proporciona una arquitectura modular y bien estructurada, facilitando la reutilización de código y la gestión eficiente de complejidades [78].

4.1.2. Flujo de datos / Vista de alto nivel

4.2. Diseño Lógico

4.2.1. Diagrama de despliegue

Los diagramas de despliegue (DD) se utilizan para visualizar los detalles de implementación de un sistema de software. Los diagramas incluyen más que sólo código, sino también bibliotecas separadas, un instalador, archivos de configuración y muchas otras piezas. Para que el software esté listo para ejecutarse, es necesario comprender todos los archivos y ejecutables involucrados y los entornos donde residen.

El DD Ilustra el hardware del sistema y su software. Útil cuando se implementa una solución de software en múltiples máquinas con configuraciones únicas.

Dos tipos especiales de dependencias: la importación de paquetes y la fusión de paquetes.

Pueden representar los diferentes niveles de un sistema para revelar la arquitectura. Deben indicar las dependencias entre paquetes y comunicación entre estos.

En la Figura 4.1, se presenta un ejemplo de un diagrama de despliegue.

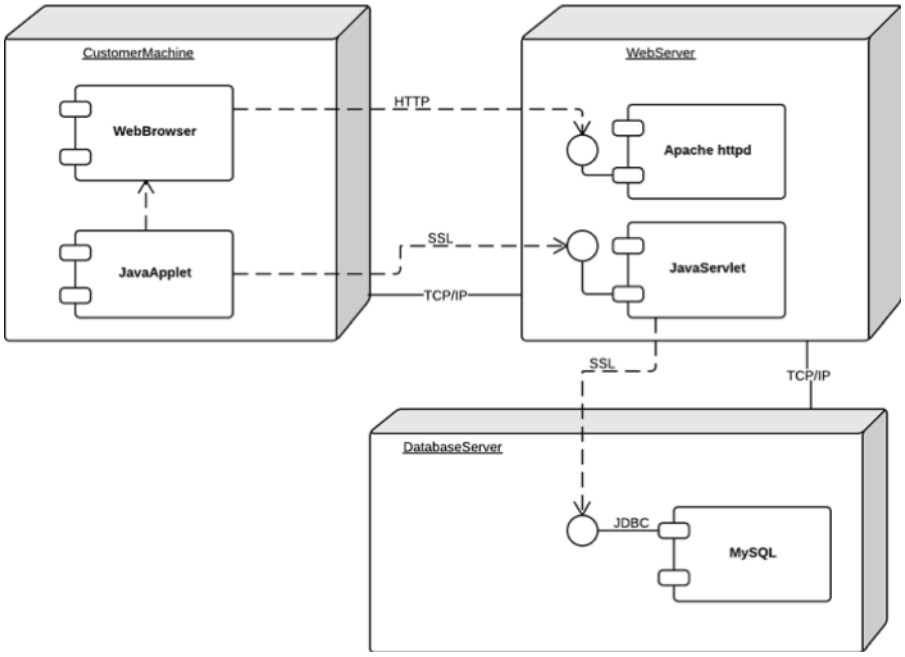

Figura 4.1: Ejemplo Diagrama de Despliegue

4.2.2. Diagrama de componentes

Los diagramas de componentes tienen relación con los componentes de un sistema. Los componentes son las unidades independientes y encapsuladas dentro de un sistema. Cada componente proporciona una interfaz para que otros componentes interactúen con él. Los diagramas de componentes se utilizan para visualizar cómo interactúan las piezas de un sistema y qué relaciones tienen entre ellas.

Los diagramas de componentes son diferentes de la mayoría de los otros diagramas, ya que muestran una estructura de alto nivel y no detalles como atributos y métodos. Están puramente enfocados en los componentes y sus interacciones entre sí.

Los diagramas de componentes son una vista estática del software y representan el diseño del sistema en un punto específico de su desarrollo y evolución. La base de los diagramas de componentes se centra en los componentes y sus relaciones. Cada componente de un diagrama tiene una relación muy específica con los otros componentes a través de la interfaz que proporciona.

Los diagramas de componentes tienen conectores de bola, que representan una interfaz proporcionada. Una interfaz provista muestra que un componente ofrece una interfaz para que otros interactúen con él. La interfaz provista significa que los componentes de clientes tienen una forma de comunicarse con ese componente.

Los diagramas de componentes también tienen conectores de "*socket*" que despliegan una interfaz requerida. En los diagramas de componentes, la interfaz requerida es esencial. Puesto que muestra que un componente espera una interfaz determinada. Esta interfaz esperada debe ser proporcionada por algún otro componente.

Finalmente, los diagramas de componentes pueden ilustrar una relación de dependencia. Una relación de dependencia ocurre cuando la interfaz provista por un componente coincide con la interfaz requerida por otro componente. La interfaz proporcionada está representada por una bola, y la interfaz requerida está representada por un *socket*.

A continuación, en la figura 4.2, se presenta un ejemplo de un diagrama de componentes para un videojuego:

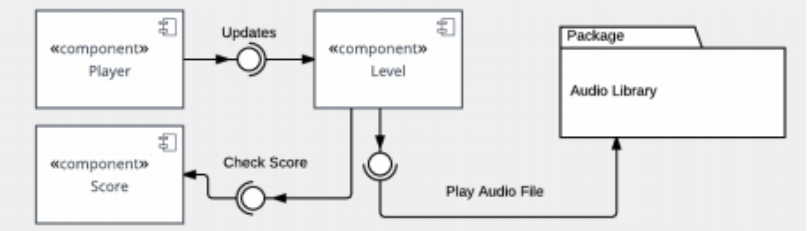

Figura 4.2: Ejemplo Diagrama de Componentes

Los diagramas de componentes son especialmente útiles al principio del proceso de

diseño, debido a su énfasis de alto nivel. Se pueden realizar en diferentes niveles y le permiten enfocarse no sólo en los sistemas sino también en los subsistemas.

4.2.3. Diagrama de paquetes

Los diagramas de paquetes muestran los paquetes y las dependencias entre ellos. Estos diagramas pueden organizar un sistema completo en paquetes de elementos relacionados, estos podrían incluir datos, clases o incluso otros paquetes. Los diagramas de paquetes ayudan a proporcionar agrupaciones de alto nivel de un sistema para que sea fácil visualizar cómo un paquete contiene elementos relacionados, así como la forma en que los diferentes paquetes dependen entre sí.

A continuación, en la figura 4.3, se presenta un ejemplo de un diagrama de paquetes para un videojuego:

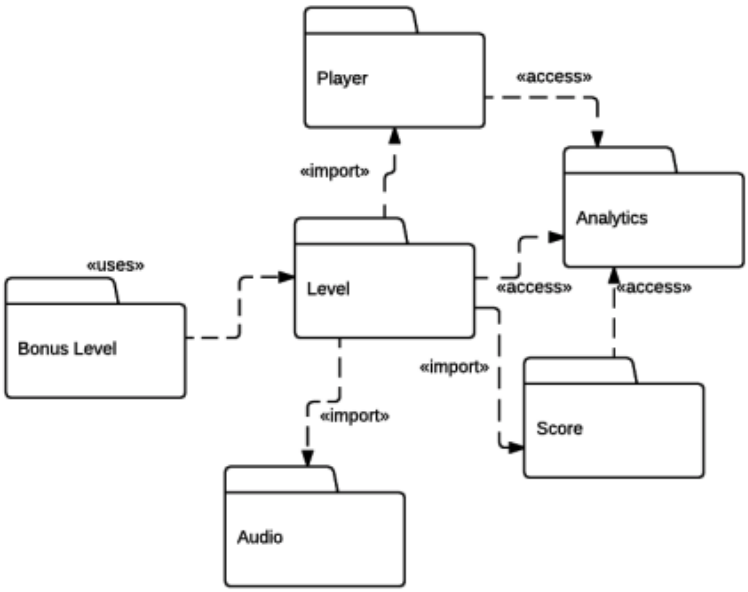

Figura 4.3: Ejemplo Diagrama de Paquetes

4.2.4. Diagrama de clases

El Diagrama de clase da la vista estática de una aplicación. Un diagrama de clase describe los tipos de objetos en el sistema y los diferentes tipos de relaciones que existen entre ellos. Este método de modelado se puede ejecutar con casi todos los métodos orientados a objetos.

El Diagrama de clase ofrece una descripción general de un sistema de software al mostrar clases, atributos, operaciones y sus relaciones. Este diagrama incluye el nombre de la clase, los atributos y la operación en compartimientos designados separados.

Para finalizar, el Diagrama de clase ayuda a construir el código para el desarrollo de aplicaciones de software.

Los elementos esenciales en un diagrama de clases son:

- Nombre de la clase
- Atributos
- Operaciones

A continuación, en la figura 4.4, se presenta un ejemplo de un diagrama de clases:

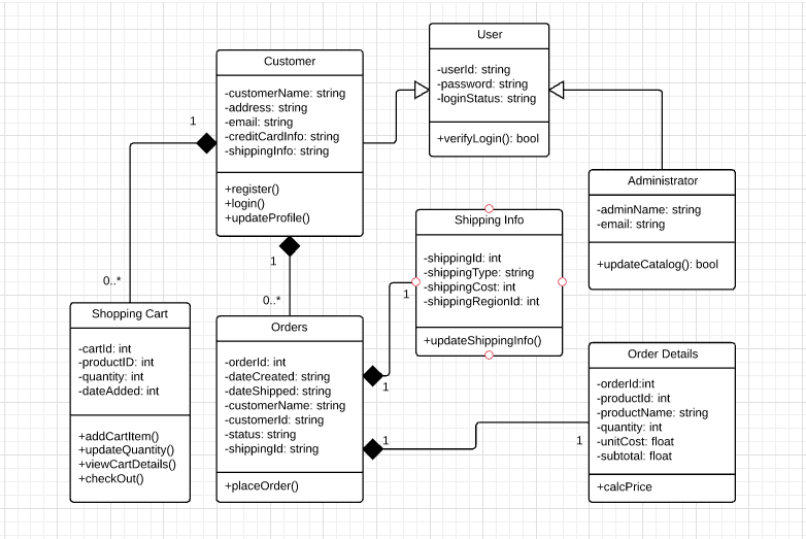

Figura 4.4: Ejemplo Diagrama de Clases

4.3. Diseño de Datos

4.3.1. Diagrama Entidad Relación

4.3.2. Modelo Relacional

4.3.3. Diccionario de Datos

4.4. Diseño de Interfaz

4.4.1. Arquitectura de la Información

4.4.2. Prototipos de Interfaces Gráficas

4.5. Diseño de Pruebas

4.5.1. Pruebas Unitarias

4.5.2. Pruebas de Integración

4.5.3. Pruebas de Sistema

4.5.4. Pruebas con Usuarios

4.5.5. Pruebas de Aceptación

4.6. Diagramas Opcionales / Complementario

Capítulo 5

Experimentación

5.1. Diseño de experimentos

Comenzaremos definiendo ciertos conceptos de ámbito general a tener en cuenta durante el proceso de diseño de experimentos y posterior procesamiento y almacenamiento de datos, estos son los siguientes:

- Diseño Experimental: Definir los objetivos de los experimentos a realizar, determinando las variables participantes y factores que pueden alterar o no el resultado del experimento en sí.
- Mediciones: Las mediciones permiten realizar un análisis estadístico de los resultados obtenidos en base a una selección de ciertos criterios o métricas de interés.
- Variabilidad : Es una característica del material biológico y plantea el problema de decidir si las diferencias entre unidades experimentales se deben a la variabilidad no ponderada o a los efectos reales del tratamiento.
- Reporte: Un reporte debe presentar de manera integra los resultados obtenidos durante las pruebas o experimentos realizados, en conjunto con su respectivo análisis, debe garantizar la reproducción de los experimentos.

5.1.1. Análisis Ordinario

Una vez realizado el análisis previo se procede a realizar un análisis de tipo ordinario. Posterior al análisis *outlier* y al removimiento de los datos atıpicos, se realizara un análisis ordinario. En este análisis se procederá a buscar una serie de medidas que nos ayudaran a comparar y contrastar los resultados obtenidos de los dos algoritmos. Las medidas a determinar son las siguientes:

5.2. Datasets y Casos de estudio

5.2.1. Caso de Estudio

A continuación se presenta el caso de estudio centrado en la herramienta de simulación *SimStream* [69], cuyos principales objetivos son el diseño para el modelamiento y simulación de plataformas de procesamiento de streams. Simstream simula un ejemplo popular de una aplicación de procesamiento en tiempo real como su caso de estudio. Los autores **Inostroza, Alonso, et al.** proponen una topología para este caso de estudio que esta compuesta por 2 componentes simples:

- Productor Kafka: Productor simple a cargo de leer Tweets desde la API de Streaming de Twitter y almacenarlos en *Apache Kafka*.
- Procesador Twitter: Una topología propia de Storm responsable de leer Tweets desde Apache Kafka y procesarlos de manera paralela, con el objetivo de realizar un análisis de sentimientos (*sentiment analysis*) y calcular los principales k hashtag.

La topología Storm presentada con anterioridad se compone de los siguientes componentes:

- Kafka Spout: Adaptador especifico de storm a cargo de leer tweets desde Kafka.
- Filtro Twitter: Bolt a cargo de filtrar todos los tweets que no estén en idioma inglés para aplicar adecuadamente el algoritmo de análisis de sentimientos.
- Sanitización de Texto: Bolt a cargo de la normalización del texto para aplicar apropiadamente el algoritmo de análisis de sentimientos.
- Análisis de Sentimientos: Bolt a cargo de calificar los tweets por cada una de sus palabras utilizando el clasificador *SentiWordNet*.
- Análisis de Sentimientos para Cassandra: Bolt a cargo de almacenar los tweets y su calificación de sentimiento en una base de datos de Cassandra.
- Divisor de Hashtags: Bolt encargado de dividir los diferentes hashtags y emitir una tupla por hashtag al siguiente bolt (contador de hashtags).
- Contador de Hashtags: Bolt encargado de contar el número de ocurrencias de un hashtag.
- Hashtag Principal: Bolt encargado de realizar un ranking de los hashtags principales k hashtags mediante un algoritmo de ventanas deslizantes (*Sliding Window*).
- Hashtag Principal para Cassandra: Bolt encargado de almacenar los principales k hashtags en una base de datos de Cassandra.

Respecto a Simstream, el modelo de simulación que fue implementado corresponde a un simulador orientado a procesos []. En donde los procesos son representados utilizando Bolts/Spouts. Estos se encargan del procesamiento de tuplas, mientras que los recursos son artefactos tales como los datos de los mensajes entrantes, variables globales como las solicitudes de entrada de cada proceso, CPU y redes de comunicación.

La implementación del programa de simulación utiliza la librería *libCppSim* propuesta por el Autor **Marzolla, Moreno** [108]. En esta cada proceso es implementado por una *corrutina* que puede bloqueada o desbloqueada durante la simulación, a través del uso de operaciones tales como: *hold()*, *passivate()* y *activate()*.

Una corrutina C_i puede ser pausada por un determinado tiempo δ_t, este ultimo representa la duración de una tarea. Una vez que el tiempo de simulación δ_t ha expirado, la corrutina C_i se activa por si misma siempre que una operación *hold()* haya sido ejecutada previamente. De otra forma, la corrutina C_i es activada por cualquier corrutina C_j usando la operación *activate()*. Este ultimó caso permite la representación de la interacción entre los diferentes componentes nombrados con anterioridad de la plataforma simulada.

Los nodos ubicados en en el mismo procesador físico se comunican con un bajo costo en comparación con nodos alojados en diferentes procesadores. Por otra parte, cada procesador posee recursos tales como procesadores, Ram y Memoria Cache. La política de reemplazo utilizada para ambas memorias corresponde a el menos recientemente usado o *Least Recently Used (LRU)* [109]. El tamaño de las memorias son configurados por los parámetros del simulador. Mientras que una función llamada *schedule_processing()* es utilizada para programar las tareas de los bolts/spouts en los núcleos de los procesadores a cargo.

Cabe recalcar que cada procesador posee una interfaz de red, encargada de dividir los mensajes en paquetes antes de enviarlos a través de la red de comunicación. La interfaz de red recolecta los paquetes pertenecientes al mismo mensaje al recibirlo.

Capítulo 6

Implementación

En este capítulo se detallan las herramientas, software, lenguajes de programación y dispositivos empleados en el desarrollo del proyecto. Además, se abordará la implementación de los códigos finales y se describirán las instancias utilizadas durante las pruebas realizadas.

6.1. Hardware utilizado

En esta sección, se presenta una descripción detallada del hardware utilizado en el desarrollo del proyecto. Se abordan los componentes físicos y dispositivos necesarios para respaldar las actividades de desarrollo, pruebas y despliegue de software. Cada elemento de hardware se seleccionó con el objetivo de satisfacer las demandas de rendimiento, capacidad y escalabilidad del proyecto, asegurando un entorno de trabajo óptimo y eficiente.

En el desarrollo de este proyecto, se implementó una arquitectura híbrida que aprovechaba los recursos de la estación de trabajo local para operar en la capa de Docker [110]. Esta estrategia permitió a Docker utilizar eficientemente los recursos de hardware locales para ejecutar contenedores de forma aislada en un entorno virtualizado [110]. Esta configuración proporcionó flexibilidad y escalabilidad al proyecto, al tiempo que garantizaba un rendimiento óptimo de las aplicaciones desplegadas [110].

Estos contenedores Docker comparten el kernel del sistema operativo del host, lo que permite un uso más eficiente de los recursos en comparación con las máquinas virtuales tradicionales [111]. Además, Docker utiliza directorios en el sistema de archivos local para almacenar imágenes de contenedores, datos persistentes y configuraciones de red [112]. Esto facilita la portabilidad de las aplicaciones y la gestión de los contenedores en diferentes entornos de desarrollo y producción. A continuación se detallan los componentes de la estación de trabajo local con sus respectivas especificaciones:

- Procesador Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz 2.59 GHz, 6 núcleos y

12 hilos.

- 8 GB de memoria RAM DDR4 2933 MHz.
- 500 GB (477 GB disponibles) de almacenamiento SSD.
- Sistema Operativo: Windows 11 Home (22H2 versión) de 64 bits.

6.2. Software utilizado

En esta sección, se presenta una descripción detallada del software utilizado en el desarrollo del proyecto. Se abordan herramientas y tecnologías fundamentales que respaldan diferentes aspectos del desarrollo de software, desde el diseño hasta la implementación y la gestión de la infraestructura. Cada herramienta se seleccionó cuidadosamente para satisfacer las necesidades específicas del proyecto, garantizando la eficiencia, la fiabilidad y la calidad del resultado final. A continuación, se detallan las principales características de cada software utilizado, junto con las referencias pertinentes que respaldan su elección y su idoneidad para el proyecto.

6.2.1. Visual Studio Code

Visual Studio Code o VS Code [113] es un entorno de desarrollo integrado (IDE) altamente versátil y ampliamente utilizado en la comunidad de desarrollo de software. Ofrece una amplia gama de características y extensiones que incluyen herramientas para la integración con sistemas de control de versiones como Git, resaltado de sintaxis para una variedad de lenguajes de programación, depuración de código en diferentes entornos y la integración con servicios en la nube para la colaboración en tiempo real [114]. Además, su arquitectura extensible permite a los desarrolladores personalizar y ampliar sus capacidades según sus necesidades específicas de desarrollo [113].

6.2.2. Docker y Docker Desktop

Docker es una plataforma que simplifica el desarrollo, envío y ejecución de aplicaciones en contenedores. Estos contenedores son entornos de software auto contenidos que encapsulan todo lo necesario para ejecutar una aplicación, incluyendo bibliotecas, dependencias y código. Esto permite la creación de entornos de desarrollo consistentes y portátiles en diferentes sistemas operativos[115]. Docker Desktop, por otro lado, es una aplicación diseñada para permitir a los desarrolladores ejecutar Docker en sistemas operativos de escritorio como Windows y macOS. Proporciona una interfaz gráfica intuitiva para gestionar contenedores, lo que facilita la creación, ejecución y depuración eficiente de contenedores[106].

6.2.3. Node JS

Node.js es un entorno de ejecución de JavaScript basado en el motor V8 de Google Chrome, conocido por su eficiencia y escalabilidad. Permite a los desarrolladores utilizar JavaScript tanto en el lado del cliente como en el servidor, lo que facilita el desarrollo de aplicaciones web y API RESTful de manera coherente y eficiente [116]. Además, Node.js cuenta con un amplio ecosistema de módulos y paquetes npm que permiten a los desarrolladores acceder a una amplia gama de funcionalidades predefinidas para sus proyectos, lo que acelera el proceso de desarrollo y mejora la productividad del equipo [117].

6.2.4. React JS

React.js es una biblioteca de JavaScript ampliamente utilizada para construir interfaces de usuario interactivas y dinámicas en aplicaciones web de una sola página (SPA). Proporciona una forma eficiente y declarativa de crear componentes reutilizables que representan diferentes partes de la interfaz de usuario. Además, React utiliza un enfoque basado en componentes, lo que permite dividir la interfaz de usuario en componentes independientes y luego componerlos para formar la aplicación completa [118]. Una de las características distintivas de React es su capacidad para actualizar eficientemente el DOM virtual, lo que mejora el rendimiento y la velocidad de la aplicación al minimizar las actualizaciones del DOM en el navegador del usuario. Además, React cuenta con una amplia comunidad de desarrolladores y una gran cantidad de bibliotecas y herramientas complementarias que facilitan el desarrollo de aplicaciones web modernas [119].

6.2.5. RabbitMQ

RabbitMQ es un sistema de mensajería de código abierto ampliamente utilizado para la implementación de colas de mensajes en arquitecturas de aplicaciones distribuidas. Permite la comunicación asíncrona entre diferentes componentes del sistema, facilitando la integración y la escalabilidad. La plataforma ofrece una variedad de características robustas, incluyendo enrutamiento flexible, confirmaciones de entrega, clustering y soporte para diversos protocolos de comunicación [120, 121, 122, 123]. Una de las ventajas clave de RabbitMQ es su capacidad para gestionar grandes volúmenes de mensajes de manera eficiente, lo que lo hace ideal para aplicaciones que requieren alta disponibilidad y rendimiento. Además, su arquitectura flexible y modular permite su integración con una amplia gama de tecnologías y sistemas, lo que lo convierte en una opción popular para casos de uso que van desde la mensajería entre microservicios hasta la integración de sistemas empresariales [120, 121, 122, 123].

6.2.6. MongoDB

MongoDB es una base de datos NoSQL altamente versátil y escalable que se ha convertido en una opción popular para el desarrollo de aplicaciones modernas. Su modelo de datos flexible basado en documentos JSON permite almacenar y manipular datos de manera eficiente [107]. Además, MongoDB ofrece características avanzadas como la indexación, consultas ad hoc y agregaciones, lo que facilita el análisis y la recuperación de datos [124]. Su arquitectura distribuida y capacidad de escalado horizontal hacen que sea adecuada para aplicaciones que requieren un alto rendimiento y una gran capacidad de almacenamiento [125]. La capacidad de replicación y fragmentación de datos garantiza la alta disponibilidad y confiabilidad del sistema, lo que lo convierte en una opción atractiva para entornos empresariales exigentes [126].

6.2.7. Express JS

Express.js es un marco de desarrollo web rápido y minimalista para Node.js que proporciona una serie de características para la construcción de aplicaciones web y API RESTful [127]. Destaca por su eficacia para simplificar el proceso de desarrollo del lado del servidor utilizando JavaScript [128]. Este marco, ampliamente documentado en ”Express.js Deep API Reference” de Azat Mardan Sriram, ofrece una API sencilla pero poderosa para manejar solicitudes HTTP, enrutamiento, gestión de sesiones, middleware y mucho más [129]. Además, se integra eficientemente con otras tecnologías populares como MongoDB y AngularJS para desarrollar aplicaciones web modernas y dinámicas [130].

6.3. Lenguajes de programación

En esta sección se presentan los lenguajes de programación utilizados en el desarrollo del trabajo. Se detalla el papel de cada lenguaje en la implementación de diferentes componentes del sistema, así como su relevancia en el logro de los objetivos del proyecto. Además, se proporciona una descripción de las características clave de cada lenguaje y se discuten las razones detrás de su elección para el desarrollo de este trabajo. A continuación se describe cada uno de los lenguajes de programación utilizados:

6.3.1. Javascript y JSX

JavaScript es un lenguaje de programación ampliamente utilizado en el desarrollo web debido a su capacidad para agregar interactividad y dinamismo a las páginas web. Es un lenguaje interpretado, orientado a objetos y basado en prototipos, que se ejecuta en el navegador del cliente. Se utiliza para manipular el contenido HTML, interactuar con el usuario y comunicarse con el servidor a través de solicitudes HTTP [131].

JSX, por otro lado, es una extensión de JavaScript utilizada en React.js para definir la estructura de los componentes de la interfaz de usuario. JSX permite escribir código HTML dentro de JavaScript de una manera más intuitiva y eficiente. Facilita la creación de componentes reutilizables y la composición de la interfaz de usuario mediante la definición de componentes anidados y la incorporación de lógica de JavaScript dentro del marcado [132].

La elección de JavaScript y JSX para el desarrollo de React.js se debe a varias razones. En primer lugar, JavaScript es el lenguaje principal utilizado en el ecosistema de desarrollo web, lo que lo convierte en una opción natural para el desarrollo de aplicaciones web [133]. Además, JSX simplifica la creación de componentes de interfaz de usuario en React.js al proporcionar una sintaxis familiar y expresiva. Esto hace que el desarrollo de aplicaciones web con React.js sea más rápido y eficiente. Además, React.js está diseñado para trabajar bien con JavaScript y JSX, lo que garantiza un rendimiento óptimo y una excelente experiencia de desarrollo para los desarrolladores.

6.3.2. Typescript

TypeScript es un lenguaje de programación que se ha vuelto popular en el desarrollo de backend debido a su tipado estático, que proporciona una capa adicional de seguridad y robustez al código [134]. Al ser un superconjunto de JavaScript, TypeScript permite a los desarrolladores utilizar las características avanzadas de ES6 y ES7, como las clases, módulos y funciones flecha, mientras agrega tipos estáticos opcionales que pueden detectar errores en tiempo de compilación. Esto hace que el código sea más fácil de mantener y escalar, especialmente en proyectos grandes.

Una de las ventajas clave de TypeScript es su interoperabilidad con JavaScript, lo que permite a los desarrolladores migrar gradualmente sus aplicaciones existentes a TypeScript sin tener que reescribir todo el código [134]. Además, TypeScript cuenta con un sistema de tipos robusto que permite a los desarrolladores definir interfaces personalizadas, tipos de datos complejos y uniones de tipos, lo que facilita la creación de APIs claras y bien definidas.

La elección de TypeScript como lenguaje de programación se justifica por su integración fluida con otros frameworks y bibliotecas ampliamente utilizados en Node.js, tales como Express.js y RabbitMQ [134, 78, 72]. Express.js facilita la creación de servidores y rutas HTTP de forma ágil y eficiente, mientras que RabbitMQ, un sistema de mensajería de código abierto, es esencial para la comunicación entre microservicios y otras partes de la arquitectura de la aplicación. Estas herramientas, en combinación con TypeScript, proporcionan una base robusta y mantenible para el desarrollo del backend de aplicaciones web modernas y escalables.

6.4. Estrategia de implementación

La estrategia de implementación para la implementación del trabajo de título centrado en desarrollar un entorno web para la simulación de plataformas de procesamiento de streams, se basa en una cuidadosa selección de tecnologías y enfoques arquitectónicos capaces de garantizar la efectividad, seguridad, escalabilidad y mantenibilidad del sistema.

Capítulo 7

Resultados

7.1. Implantación

7.1.1. Requerimientos

Requerimientos de Mínimos

Requerimientos de Recomendados

7.1.2. Preparación de Ambiente

7.1.3. Documentación

7.1.4. Manual de Usuario

7.1.5. Documentación de desarrollo

Capítulo 8

Conclusiones

8.1. Conclusiones

Bibliografía

[1] M. Khalid and M. M. Yousaf, "A comparative analysis of big data frameworks: An adoption perspective," *Applied Sciences*, vol. 11, no. 22, p. 11033, 2021.

[2] T. Rabl, J. Traub, A. Katsifodimos, and V. Markl, "Apache flink in current research," *it-Information Technology*, vol. 58, no. 4, pp. 157–165, 2016.

[3] M. A. Lopez, A. G. P. Lobato, and O. C. M. Duarte, "A performance comparison of open-source stream processing platforms," in *2016 IEEE Global Communications Conference (GLOBECOM)*. IEEE, 2016, pp. 1–6.

[4] PhoenixNAP, "Apache Storm," https://phoenixnap.com/kb/apache-storm, accedido el 12 de marzo de 2024.

[5] S. Salloum, R. Dautov, X. Chen, P. X. Peng, and J. Z. Huang, "Big data analytics on apache spark," *International Journal of Data Science and Analytics*, vol. 1, pp. 145–164, 2016.

[6] Apache Spark Documentation. Cluster Overview - Apache Spark Documentation. [Online]. Available: https://spark.apache.org/docs/latest/cluster-overview.html

[7] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: fault-tolerant streaming computation at scale," p. 423–438, 2013. [Online]. Available: https://doi.org/10.1145/2517349.2522737

[8] B. Babcock and S. Babu, "Models and issues in data stream systems," *Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pp. 1–16, 2002.

[9] T. Barnett, S. Jain, U. Andra, and T. Khurana, "Cisco visual networking index (vni) complete forecast update, 2017–2022," *Americas/EMEAR Cisco Knowledge Network (CKN) Presentation*, pp. 1–30, 2018.

[10] B. Dab, I. Fajjari, N. Aitsaadi, and G. Pujolle, "Vnr-ga: Elastic virtual network reconfiguration algorithm based on genetic metaheuristic," in *2013 IEEE Global Communications Conference (GLOBECOM)*. IEEE, 2013, pp. 2300–2306.

[11] Apache Software Foundation. (Fecha de acceso: Febrero 2024) Apache kafka. [Online]. Available: https://kafka.apache.org/

[12] N. Garg, *Apache kafka*. Packt Publishing Birmingham, UK, 2013.

[13] Apache Software Foundation, "Apache flink," *Sitio web de Apache Flink*, Fecha de acceso: Febrero 2024. [Online]. Available: https://flink.apache.org/

[14] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *The Bulletin of the Technical Committee on Data Engineering*, vol. 38, no. 4, 2015.

[15] Apache Software Foundation, "Apache spark," *Sitio web de Apache Spark*, Fecha de acceso: Febrero 2024. [Online]. Available: https://spark.apache.org/

[16] S. Kane and K. Matthias, *Docker: Up & Running*. O'Reilly Media, 2023. [Online]. Available: https://books.google.cl/books?id=hWC5EAAAQBAJ

[17] J. Schwaber, *Scrum: The Art of Doing Twice the Work in Half the Time*. Currency, 2017.

[18] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina *et al.*, "The design of the borealis stream processing engine." in *Cidr*, vol. 5, no. 2005, 2005, pp. 277–289.

[19] D. J. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, "Aurora: a new model and architecture for data stream management," *the VLDB Journal*, vol. 12, pp. 120–139, 2003.

[20] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom, "Stream: The stanford data stream management system," *Data Stream Management: Processing High-Speed Data Streams*, pp. 317–336, 2016.

[21] R. S. Barga, J. Goldstein, M. Ali, and M. Hong, "Consistent streaming through time: A vision for event stream processing," *arXiv preprint cs/0612115*, 2006.

[22] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah, "Telegraphcq: continuous dataflow processing," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, 2003, pp. 668–668.

[23] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk, "Gigascope: A stream database for network applications," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, 2003, pp. 647–651.

[24] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt *et al.*, "The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1792–1803, 2015.

[25] M. Fragkoulis, P. Carbone, V. Kalavri, and A. Katsifodimos, "A survey on the evolution of stream processing systems," *The VLDB Journal*, vol. 33, no. 2, pp. 507–541, 2024. [Online]. Available: https://rdcu.be/dBEoB

[26] M. Rychly, P. Smrz *et al.*, "Scheduling decisions in stream processing on heterogeneous clusters," in *2014 Eighth International Conference on Complex, Intelligent and Software Intensive Systems*. IEEE, 2014, pp. 614–619.

[27] H. C. Andrade, B. Gedik, and D. S. Turaga, *Fundamentals of stream processing: application design, systems, and analytics*. Cambridge University Press, 2014.

[28] J. Kreps, *I Heart Logs: Event Data, Stream Processing, and Data Integration*. O'Reilly Media, 2014.

[29] O. Etzion and P. Niblett, *Event processing in action*. Manning Publications Co., 2010.

[30] J. Boubeta-Puig, G. Díaz, H. Macià, V. Valero, and G. Ortiz, "Medit4cep-cpn: An approach for complex event processing modeling by prioritized colored petri nets," *Information systems*, vol. 81, pp. 267–289, 2019.

[31] W. A. Higashino, M. A. Capretz, and L. F. Bittencourt, "Cepsim: Modelling and simulation of complex event processing systems in cloud environments," *Future Generation Computer Systems*, vol. 65, pp. 122–139, 2016.

[32] T.-D. Nguyen and E.-N. Huh, "Ecsim++: An inet-based simulation tool for modeling and control in edge cloud computing," in *2018 IEEE International Conference on Edge Computing (EDGE)*, 2018, pp. 80–86.

[33] G. Amarasinghe, M. D. de Assuncao, A. Harwood, and S. Karunasekera, "Ecsnet++: A simulator for distributed stream processing on edge and cloud environments," *Future Generation Computer Systems*, vol. 111, pp. 401–418, 2020.

[34] P. Muthukrishnan and K. Fadnis, "Discrete event simulation based performance evaluation of stream processing systems," in *2020 IEEE International Conference on Big Data (Big Data)*. IEEE, 2020, pp. 1089–1098.

[35] Apache Software Foundation, "Apache samza," *Sitio web de Apache Samza*, Fecha de acceso: Febrero 2024. [Online]. Available: https://samza.apache.org/

[36] ——, "Apache storm," *Sitio web de Apache Storm*, Fecha de acceso: Febrero 2024. [Online]. Available: https://storm.apache.org/

[37] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell, "Samza: stateful scalable stream processing at linkedin," *Proceedings of the VLDB Endowment*, vol. 10, no. 12, pp. 1634–1645, 2017.

[38] "Aws kinesis," https://aws.amazon.com/kinesis/, accedido el 12 de marzo de 2024.

[39] "Azure eventhub," https://azure.microsoft.com/en-us/services/event-hubs/, accedido el 12 de marzo de 2024.

[40] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*, 2013, pp. 1–16.

[41] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: a platform for fine-grained resource sharing in the data center," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11. USA: USENIX Association, 2011, p. 295–308.

[42] P. Carbone, G. Fóra, S. Ewen, S. Haridi, and K. Tzoumas, "Lightweight asynchronous snapshots for distributed dataflows," *arXiv preprint arXiv:1506.08603*, 2015.

[43] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl, "Spinning fast iterative data flows," *arXiv preprint arXiv:1208.0088*, 2012.

[44] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino, "Apache tez: A unifying framework for modeling and building data processing applications," in *Proceedings of the 2015 ACM SIGMOD international conference on Management of Data*, 2015, pp. 1357–1369.

[45] J. Kreps, N. Narkhede, J. Rao *et al.*, "Kafka: A distributed messaging system for log processing," in *Proceedings of the NetDB*, vol. 11, no. 2011. Athens, Greece, 2011, pp. 1–7.

[46] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. Ieee, 2010, pp. 1–10.

[47] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham *et al.*, "Storm@ twitter," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, 2014, pp. 147–156.

[48] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "{ZooKeeper}: Wait-free coordination for internet-scale systems," in *2010 USENIX Annual Technical Conference (USENIX ATC 10)*, 2010.

[49] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A {Fault-Tolerant} abstraction for {In-Memory} cluster computing," in *9th USENIX symposium on networked systems design and implementation (NSDI 12)*, 2012, pp. 15–28.

[50] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen *et al.*, "Mllib: Machine learning in apache spark," *The journal of machine learning research*, vol. 17, no. 1, pp. 1235–1241, 2016.

[51] J. E. Gonzalez, "From graphs to tables the design of scalable systems for graph analytics," in *Proceedings of the 23rd International Conference on World Wide Web*, ser. WWW '14 Companion. New York, NY, USA: Association for Computing Machinery, 2014, p. 1149–1150. [Online]. Available: https://doi.org/10.1145/2567948.2580059

[52] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "Graphx: A resilient distributed graph system on spark," in *First international workshop on graph data management experiences and systems*, 2013, pp. 1–6.

[53] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark sql: Relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 1383–1394. [Online]. Available: https://doi.org/10.1145/2723372.2742797

[54] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache hadoop yarn: yet another resource negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2523616.2523633

[55] A. J. Park, C.-H. Li, R. Nair, N. Ohba, U. Shvadron, A. Zaks, and E. Schenfeld, "Flow: A stream processing system simulator," in *2010 IEEE Workshop on Principles of Advanced and Distributed Simulation*, 2010, pp. 1–9.

[56] M. Kaptein, "Rstorm: Developing and testing streaming algorithms in r," *The R Journal*, vol. 6, no. 1, pp. 123–132, 2014.

[57] Z. Fang and H. Yu, "Parallel simulation of large-scale distributed stream processing systems," in *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, 2017, pp. 1061–1068.

[58] Y. Ma and E. A. Rundensteiner, "Scalable and efficient simulation of distributed stream processing applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 9, pp. 2032–2046, 2018.

[59] M. Gil-Costa, L. Tapia, and J. Marin, "Asynchronous simulation protocol for stream processing platforms," *Simulation Modelling Practice and Theory*, vol. 68, pp. 123–136, 2016.

[60] S. Kamburugamuve, K. Ramasamy, M. Swany, and G. Fox, "Low latency stream processing: Apache heron with infiniband & intel omni-path," in *Proceedings of the10th International Conference on Utility and Cloud Computing*, 2017, pp. 101–110.

[61] A. S. Documentation. (2024) Apache storm documentation. [Online]. Available: https://storm.apache.org/documentation

[62] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, N. Bhagat, S. Mittal, and D. Ryaboy, "Storm@twitter," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, 2014, pp. 147–156.

[63] IBM. (2024) Stream processing. [Online]. Available: https://www.ibm.com/cloud/learn/stream-processing

[64] J. Kreps, "Questioning the lambda architecture," *O'Reilly Radar*, 2014. [Online]. Available: https://www.oreilly.com/radar/questioning-the-lambda-architecture/

[65] N. Bonvin, T. Papaioannou, and K. Aberer, "The quest for an understandable and usable model for distributed stream processing systems," *ACM Computing Surveys (CSUR)*, vol. 52, no. 1, pp. 1–38, 2019.

[66] A. S. Foundation. (2022) Apache storm architecture. [Online]. Available: https://storm.apache.org/releases/2.3.0/Concepts.html

[67] M. Sathiamoorthy, I. Moraru, S. A. Akhlaghi, I. Gupta, and I. Stoica, "Consistency, availability, and convergence: Beyond the cap theorem," *Communications of the ACM*, vol. 58, no. 12, pp. 68–77, 2015.

[68] K. Lakshmanan, A. Das, and R. Yerneni, "Apache storm: a real-time stream processing system," *ACM SIGMOD Record*, vol. 43, no. 2, pp. 105–112, 2014.

[69] A. Inostrosa-Psijas, R. Solar, M. Marin, V. Gil-Costa, and G. Wainer, "Modeling and simulating stream processing platforms," in *2023 Winter Simulation Conference (WSC)*, 2023, pp. 3130–3141.

[70] J. Smith, "Complexity in software configuration," *Journal of Software Engineering*, 2023.

[71] J. Doe, "Usability challenges in software applications," *International Journal of Human-Computer Interaction*, 2022.

[72] "Rabbitmq," accessed: 2024-02-15. [Online]. Available: https://www.rabbitmq.com/

[73] S. Krug, *Don't Make Me Think: A Common Sense Approach to Web Usability*. New Riders, 2000.

[74] R. S. Pressman, *Software Engineering: A Practitioner's Approach*. McGraw-Hill Education, 2014.

[75] M. P. Papazoglou, "Service-oriented architecture: characteristics and challenges," *Information systems*, vol. 35, no. 4, pp. 357–370, 2010.

[76] S. Newman, *Building microservices*. O'Reilly Media, Inc., 2015.

[77] T. Erl, *SOA: principles of service design*. Prentice Hall, 2009.

[78] "Express.js," accessed: 2024-02-15. [Online]. Available: https://expressjs.com/

[79] E. Wilson and S. Raj, *Express in Action: Writing, building, and testing Node.js applications*. Manning Publications, 2016.

[80] A. Ferscha and S. K. Tripathi, "Parallel and distributed simulation of discrete event systems," Nombre de la institución, Tech. Rep., 1998.

[81] J. Sutherland, "Scrum: The future of work," *Harvard Business Review*, vol. 92, no. 6, pp. 53–62, 2014.

[82] I. Sommerville, *Software Engineering*. Pearson, 2016.

[83] R. S. Pressman, *Software Engineering: A Practitioner's Approach*. McGraw-Hill Education, 2014.

[84] M. A. Babar, *Requirements Engineering for Software and Systems*. CRC Press, 2019.

[85] J. Smith, "User management in web applications," *Journal of Web Development*, vol. 10, no. 2, pp. 45–60, 2020.

[86] J. Doe, "Content management systems: A comprehensive guide," *Journal of Content Management*, vol. 5, no. 3, pp. 112–128, 2019.

[87] D. Brown, "Efficient event scheduling techniques for web applications," in *Proceedings of the International Conference on Web Engineering*. ACM, 2021, pp. 234–245.

[88] E. Johnson, "Advanced reporting and analytics tools for business intelligence," *Journal of Business Intelligence*, vol. 15, no. 4, pp. 78–92, 2022.

[89] M. Garcia, *Integration Patterns: Integrating External Systems with Web Applications*. Tech Publishing, 2020.

[90] M. K. Patra, A. Kumari, B. Sahoo, and A. K. Turuk, "Docker security: Threat model and best practices to secure a docker container," in *2022 IEEE 2nd International Symposium on Sustainable Energy, Signal Processing and Cyber Security (iSSSC)*, 2022, pp. 1–6.

[91] "Rabbitmq documentation: Parameters," https://www.rabbitmq.com/docs/parameters, Rabbit Technologies Ltd., accedido: 28 de Marzo 2024.

[92] "How to secure your react.js application," hhttps://www.freecodecamp.org/news/best-practices-for-security-of-your-react-js-application/, accedido: 28 de Marzo 2024.

[93] "Node.js security best practices," https://nodejs.org/en/learn/getting-started/security-best-practices, Node.js Foundation., accedido: 28 de Marzo 2024.

[94] "Mongodb security checklist & best practices," https://www.mongodb.com/features/security/best-practices, MongoDB,Inc., accedido: 28 de Marzo 2024.

[95] "Express.js security best practices," https://expressjs.com/en/advanced/best-practice-security.html, Express.js, accedido: 28 de Marzo 2024.

[96] "How to optimize docker image," https://www.geeksforgeeks.org/how-to-optimize-docker-image/, GeeksforGeeks, accedido: 28 de Marzo 2024.

[97] "How to measure and improve node.js performance," https://raygun.com/blog/improve-node-performance/, Raygun, accedido: 28 de Marzo 2024.

[98] "Production best practices: performance and reliability," https://expressjs.com/en/advanced/best-practice-performance.html#do-logging-correctly, Express.js, accedido: 28 de Marzo 2024.

[99] L. Community, "¿cuáles son las formas más eficaces de escalar una aplicación en contenedores?" https://www.linkedin.com/advice/1/what-most-effective-ways-scale-containerized-application-y4ldf, accedido: 28 de Marzo 2024.

[100] V. Acharya, "Scaling node.js applications for high traffic," https://www.linkedin.com/pulse/scaling-nodejs-applications-high-traffic-vishwas-acharya-oxdwf, accedido: 28 de Marzo 2024.

[101] Linkedin, "¿cómo se puede garantizar una alta disponibilidad en la contenedorización?" https://www.linkedin.com/advice/0/how-can-you-ensure-high-availability-containerization-gh6sf, accedido: 28 de Marzo 2024.

[102] "Disaster recovery and high availability 101," https://www.rabbitmq.com/blog/2020/07/07/disaster-recovery-and-high-availability-101, Rabbit Technologies Ltd., accedido: 28 de Marzo 2024.

[103] "Using middleware," https://expressjs.com/en/guide/using-middleware.html, Express.js, accedido: 28 de Marzo 2024.

[104] M. H. Ibrahim, M. Sayagh, and A. E. Hassan, "A study of how docker compose is used to compose multi-component systems," *Empirical Software Engineering*, vol. 26, pp. 1–27, 2021.

[105] V. Bojinov, *RESTful Web API Design with Node. js*. Packt Publishing, 2015.

[106] Docker hub. [Online]. Available: https://hub.docker.com/

[107] K. Chodorow and M. Dirolf, *MongoDB: The Definitive Guide*. O'Reilly Media, Inc., 2013.

[108] M. Marzolla *et al.*, "libcppsim: a simula-like, portable process-oriented simulation library in c++," in *Proc. of ESM*, vol. 4. Citeseer, 2004, pp. 222–227.

[109] A. Vakali, "Lru-based algorithms for web cache replacement," in *Electronic Commerce and Web Technologies: First International Conference, EC-Web 2000 London, UK, September 4–6, 2000 Proceedings 1*. Springer, 2000, pp. 409–418.

[110] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, 2014.

[111] J. Smith, "The benefits of docker containers," *TechToday*, 2020. [Online]. Available: https://www.techtoday.com/benefits-of-docker

[112] A. Jones, *Docker Storage Essentials*. Packt Publishing, 2019.

[113] "Visual Studio Code," https://code.visualstudio.com/, accedido: 14 febrero 2024.

[114] "Visual Studio Code Marketplace," https://marketplace.visualstudio.com/, accedido: 14 febrero 2024.

[115] Docker - build, ship, and run any app, anywhere. [Online]. Available: https://www.docker.com/

[116] (2022) Node.js. Node.js Foundation. [Online]. Available: https://nodejs.org/

[117] (2022) About node.js. Node.js Foundation. [Online]. Available: https://nodejs.org/en/about/

[118] Facebook. (2022) React documentation. [Online]. Available: https://reactjs.org/docs/getting-started.html

[119] React Training. (2022) React router documentation. [Online]. Available: https://reactrouter.com/

[120] A. Videla and J. J. W. Williams, *RabbitMQ in Action*. Manning Publications, 2012.

[121] D. Dossot, *RabbitMQ Essentials*. Packt Publishing Ltd, 2014.

[122] S. Boschi, *RabbitMQ Cookbook*. Packt Publishing Ltd, 2013.

[123] M. Toshev, *Learning RabbitMQ*. Packt Publishing Ltd, 2015.

[124] R. Prasad, *MongoDB Applied Design Patterns*. Packt Publishing Ltd, 2013.

[125] K. Copeland, *MongoDB Applied Design Patterns*. Manning Publications, 2010.

[126] K. Banker, P. Branchedor, and S. Hopson, *Scaling MongoDB*. O'Reilly Media, Inc., 2011.

[127] J. K. Brown, *Web Development with MongoDB and Node.js*. Packt Publishing Ltd, 2014.

[128] J. R. W. . A. Boduch, *Node.js 8 the Right Way: Practical, Server-Side JavaScript That Scales*. Pragmatic Bookshelf, 2019.

[129] A. Mardan, *Express.js Deep API Reference*. Apress, 2014.

[130] S. Shaver, *Getting MEAN with Mongo, Express, Angular, and Node*. Manning Publications, 2015.

[131] D. Flanagan, *JavaScript: The Definitive Guide*. O'Reilly Media, 2011.

[132] Facebook, "Introducing jsx," React Documentation, 2022, retrieved from https://reactjs.org/docs/introducing-jsx.html.

[133] A. Granger, *React.js Essentials*. Packt Publishing, 2016.

[134] M. Chris, *Programming TypeScript*. O'Reilly Media, 2019.