Facultad de Ingeniería
Escuela de Ingeniería Informática

MODELADO Y SIMULACIÓN DE PLATAFORMAS DE PROCESAMIENTO DE STREAMS UTILIZANDO FORMALISMO DEVS

Por

Eliezer Andrés Zúñiga Nanjari

Trabajo realizado para optar al Título de
INGENIERO/A CIVIL INFORMÁTICO/A
Prof. Guía: Alonso Leonidas Psijas Inostrosa
Prof. Co-Referente: Daniel Cabrera Panigua
Agosto 2025

Resumen

Las plataformas de procesamiento de flujo de datos (streams) permiten manejar y analizar datos en tiempo real, en aplicaciones como la detección de fraudes, el análisis predictivo o el monitoreo de tráfico. Para dar soporte a estos escenarios, suele ser necesario desplegar dichas plataformas en conjuntos de computadoras que ejecuten la aplicación de forma distribuida. Sin embargo, dimensionar la infraestructura requerida —en términos de procesadores y nodos de cómputo— no es trivial, especialmente cuando se deben atender altas cargas de trabajo. Ante ello, la simulación resulta una alternativa eficaz para experimentar y evaluar el rendimiento sin recurrir al sistema real, evitando así costos excesivos o riesgos operativos. Además, contar con una herramienta de este tipo resulta relevante para los ingenieros responsables de los centros de datos, pues les permite identificar configuraciones óptimas de hardware y software antes de un despliegue real, reduciendo costos en la adquisición y consumo energético.

El trabajo de título propone el diseño e implementación de un modelo de simulación de plataformas de procesamiento de streams basado en el formalismo DEVS (Discrete Event Specification), enfatizando el comportamiento con la arquitectura de Apache Flink. La ventaja principal de DEVS consiste en aplicar una representación modular y reproducible del sistema, aportando en el modelado escalable. El modelo desarrollado considera los recursos de hardware como la infraestructura y las características de la plataforma, permitiendo analizar métricas clave como throughput y utilización, evaluando su rendimiento bajo distintas cargas de trabajo y estimar el hardware necesario para alcanzar el rendimiento requerido.

Índice general

Resumen III

1. Introducción 1
- 1.1. Principales contribuciones de este trabajo . . . 2
- 1.2. Estructura del Documento . . . 3

2. Marco Conceptual y Estado del Arte 4
- 2.1. Marco Conceptual . . . 4
  - 2.1.1. Procesamiento de flujo . . . 4
  - 2.1.2. Apache Flink . . . 9
  - 2.1.3. Conceptos de Modelamiento y Simulación . . . 15
  - 2.1.4. Simulación de Eventos Discretos (SED) . . . 18
  - 2.1.5. Especificación Formal de Sistemas de Eventos Discretos (DEVS) . . . 19
- 2.2. Estado del Arte . . . 21
  - 2.2.1. Trabajos relacionados con simulación de procesamiento de flujo . . . 21
  - 2.2.2. Otros trabajos relacionados con simulación . . . 23
  - 2.2.3. Herramientas de simulación usando formalismo DEVS . . . 24

3. Definición del Problema y Análisis de Requerimientos 25
- 3.1. Formulación del Problema . . . 25
- 3.2. Solución Propuesta . . . 26
- 3.3. Objetivos . . . 27
  - 3.3.1. Objetivo General . . . 27
  - 3.3.2. Objetivos Específicos . . . 27
- 3.4. Metodología . . . 27
- 3.5. Especificación de Requerimientos . . . 28
  - 3.5.1. Requerimientos Funcionales . . . 28
  - 3.5.2. Requerimientos No Funcionales . . . 29

4. Diseño **30**
4.1. Diseño Arquitectónico . . . 30
4.1.1. Tecnologías utilizadas . . . 30
4.1.2. Flujo de datos / Vista de alto nivel . . . 31
4.2. Diseño Lógico . . . 37
4.2.1. Diagrama modelo DEVS . . . 37
4.2.2. Diagrama de despliegue . . . 38
4.2.3. Diagrama de clases . . . 38
4.2.4. Diagrama de Flujo . . . 39
4.3. Diseño de Datos . . . 44
4.3.1. Parámetros y resultados . . . 44
4.4. Diseño de Pruebas . . . 45
4.4.1. Pruebas Unitarias . . . 46
4.4.2. Pruebas de Integración . . . 47
4.4.3. Pruebas de Sistema . . . 47
4.4.4. Pruebas de Verificación . . . 48
4.4.5. Pruebas con Aceptación / Validación . . . 49

5. Implementación **50**
5.1. Hardware utilizado . . . 50
5.2. Software utilizado . . . 51
5.3. Lenguajes de programación . . . 52
5.4. Estrategia de implementación . . . 53
5.4.1. Descripción de las etapas del proceso . . . 53
5.4.2. Patrones de implementación . . . 54

6. Pruebas **57**
6.1. Pruebas Unitarias . . . 57
6.2. Pruebas de Integración . . . 59
6.3. Pruebas de Sistema . . . 60
6.4. Pruebas de Verificación . . . 61
6.5. Pruebas de Aceptación / Validación . . . 62
6.5.1. Primer alcance . . . 62
6.5.2. Escenarios y resultados . . . 66

7. Implantación **74**

8. Conclusiones **75**

Bibliografía **76**

Índice de figuras

2.1. Procesamiento de flujo y por lotes. 5
2.2. Fases de un procesador de flujo de datos. 5
2.3. Aplicación de detección de fraudes en tiempo real. 6
2.4. Arquitectura en capas DSPF [1]. 7
2.5. Grafo definido del usuario [1]. 8
2.6. Grafo de ejecución [1]. 8
2.7. Aplicación representada en grafo lógico. [2] 9
2.8. Interacción de sistemas con Flink. [2] 10
2.9. Grafo lógico con vista paralela. [2] 10
2.10. Modelo de entorno de ejecución Flink. [3] 12
2.11. Aplicación de flujo con 5 subtareas. [3] 14
2.12. Subtareas en task slots. [3] 15
2.13. Resolución de problemas mediante la experimentación. [4] 15
2.14. Resolución de problemas mediante modelado analítico. [4] 16
2.15. Resolución de problemas mediante computación. [4] 17
2.16. Entidades básicas en Modelo & Simulación y sus relaciones. [4] 18
2.17. Un modelo acoplado. [4] 21

3.1. Flujo iterativo de modelado. 28

4.1. Modelo de alto nivel del entorno de ejecución de una aplicación de procesamiento de flujo de datos en un cluster. 32
4.2. Modelo de alto nivel de un DataFlow graph administrado por el ResourceManager. 33
4.3. Modelo interno de un nodo de cómputo. 34
4.4. Modelo resultante de aplicación desplegada en un cluster. 34
4.5. Modelo resultante de aplicación ejecutándose en un cluster. 35
4.6. Diagrama de flujo de datos. 36
4.7. Modelo DEVS del sistema. 37
4.8. Diagrama de despliegue. 38
4.9. Diagrama de clases. 39

4.10. Diagrama de flujo proceso de despliegue de trabajo. 40
4.11. Diagrama de flujo proceso de llegada y salida de un mensaje hacia un nodo. 41
4.12. Diagrama de flujo proceso de llegada y salida de mensajes del switch. . . . 42
4.13. Diagrama de flujo proceso de generación de mensajes/eventos del productor. 43
4.14. Estructura de archivos de configuración del modelo. 44
4.15. Registros de la simulación. 44
4.16. Estructura de métricas calculadas. 45
4.17. Esquema de pruebas unitarias. 46
4.18. Esquema de pruebas de integración. 47
4.19. Esquema de pruebas de sistema. 48
4.20. Esquema de pruebas de verificación. 49

5.1. Esquema alto nivel hardware [5]. 50
5.2. Esquema alto nivel de componentes de Software. 52
5.3. Trabajo en tiempo de compilación. 55
5.4. UML herencia de clases. 56

6.1. Throughput vs Tasa de arribo con 1 nodo y tiempos de procesamiento mínimos fijos en los operadores. 63
6.2. Throughput vs Tasa de arribo con 1 nodo y tiempos de procesamiento mediante distribución en los operadores. 64
6.3. Throughput vs Tasa de arribo con 1 nodo y tiempos de procesamiento con distribución en los operadores replicados 8 veces. 64
6.4. Throughput vs Tasa de arribo con 4 nodos y tiempos de procesamiento mediante distribución en los operadores replicados 8 veces. 65
6.5. Throughput vs Tasa de arribo con 8 nodos y tiempos de procesamiento mediante distribución en los operadores replicados 8 veces. 65
6.6. Throughput vs Tasa de arribo. 66
6.7. Utilización de un nodo con 1 núcleo. 67
6.8. Utilización de operadores sin réplica. 68
6.9. Utilización de 6 nodos con 1 núcleo c/u. 69
6.10. Utilización de operadores sin réplica. 69
6.11. Utilización de 6 nodos con 1 núcleo c/u. 70
6.12. Utilización de operadores con replicación = 5 para *map*, *keyBy* y *windows* . 70
6.13. Utilización de 6 nodos con 1 núcleo c/u. 71
6.14. Utilización de operadores, todos con nivel de replicación = 5. 72
6.15. Utilización de 6 nodos con 2 núcleos c/u. 73
6.16. Utilización de operadores, todos con nivel de replicación = 5. 73

Índice de tablas

2.1. Herramientas DEVS. 24

3.1. Requerimientos funcionales. 28
3.2. Requerimientos no funcionales. 29

4.1. Patrón de asignación de recursos. 33
4.2. Patrón de asignación de recursos final. 35
4.3. Métricas de interés. 45

6.1. Pruebas unitarias modelo Productor. 57
6.2. Pruebas unitarias modelo Nodo. 58
6.3. Pruebas unitarias modelo Switch. 58
6.4. Pruebas de integración modelo Productor-Nodo. 59
6.5. Pruebas de integración modelo Productor-Nodo-Switch. 59
6.6. Pruebas de integración modelo Productor-Nodo-Switch. 60
6.7. Pruebas de verificación modelo Productor. 61
6.8. Pruebas de verificación modelo Nodo. 61
6.9. Pruebas de verificación modelo Switch. 61
6.10. Topología. 62
6.11. Propiedad de operadores. 62
6.12. Resumen de resultados. 66

Capítulo 1

Introducción

Las plataformas de procesamiento de flujo están diseñadas para gestionar cantidades sustanciales de datos de diversas fuentes, por ejemplo contenidos de una red social, aplicaciones científicas [6], detección de fraudes con tarjetas de crédito, análisis predictivos, monitoreo, control de tráfico, etc [7]. Aquellas plataformas tienen como objetivo procesar datos en tiempo real y de manera totalmente integrada, para proporcionar información y resultados para los consumidores y/o usuarios finales, con el objetivo de integrar nueva información para apoyar la toma de decisiones a mediano y largo plazo [8]. Por ende, estas plataformas necesitan procesar grandes cantidades de datos de manera casi instantánea, con muy baja latencia y alto rendimiento, para producir algún conocimiento o información a partir de los datos. Por lo tanto, la velocidad de procesamiento debe solventar el volumen de datos entrantes y, al mismo tiempo, proporcionar un análisis de resultados de alta calidad lo más rápido posible [6].

La herramienta de procesamiento de flujo más popular -Apache Kafka- [9] recibió su nombre en honor a Franz Kafka por parte de su creador, Jay Kreps, aficionado a su lectura[10]. Actualmente, existen varias plataformas de flujo, como Apache Storm [11], Amazon Kinesis [12], Google Cloud Pub/Sub [13], con sus principios subyacentes similares. Generalmente, las aplicaciones de flujo de datos utilizan el paso de mensajes en una colección (cluster) de computadoras conectadas por canales de datos o enlaces. Cada máquina incluye una variedad de elementos de procesamiento (PE) que se utilizan para procesar y transferir datos desde su cola de mensajes de entrada y producir resultados en su cola de salida. Cada PE tiene un espacio de direcciones independiente, mientras que todas las dependencias entre las unidades de procesamiento se realizan explícitamente mediante canales o enlaces de mensajes [6].

Estas plataformas requieren de una infraestructura diseñada para procesar grandes volúmenes de datos originados en diversas fuentes en tiempo real y capaz de aplicar

diversas operaciones de acuerdo a la lógica de negocio. Por lo tanto, requerirá recursos necesarios para cierto propósito, inherentemente un sistema distribuido desplegado en un cluster con múltiples procesadores para garantizar alto rendimiento. Dimensionar estos requerimientos, principalmente en términos de nodos o procesadores, obligará a probar el sistema con recursos disponibles (incluso invertir en mas), sin embargo no sería apropiado debido al desperdicio en tiempo de ejecución o gastos innecesarios en más recursos sin medida, por lo que es conveniente experimentar y medir pero sin recurrir al sistema real, mas bien imitar aquel sistema usando un modelo abstracto mediante la simulación. Por lo tanto, simular aplicaciones que se ejecutan sobre una plataforma de procesamiento de flujo puede ser útil para evaluar su rendimiento antes de ponerlos en producción.

En este trabajo, se propone modelar y simular plataformas de procesamiento de flujo mediante formalismo DEVS, considerando las características de la plataforma e infraestructura para evaluar su desempeño en diversas situaciones y estimar los recursos de hardware requerido. Con DEVS es posible aplicar una representación modular y reproducible del sistema mediante componentes atómicos y acoplados. El caso de estudio de plataforma de flujo es Apache Flink y para ello es necesario entender cómo funciona su arquitectura para el comportamiento de la simulación. Flink es un marco y motor de procesamiento distribuido para cálculos con estado, diseñado para ejecutarse en todos los entornos de cluster comunes, realizar cálculos a velocidad en memoria y a cualquier escala [14]. El funcionamiento y arquitectura de Flink se explica en la 2.1.2.

1.1. Principales contribuciones de este trabajo

Estimar los recursos y evaluar el rendimiento necesario para ejecutar aplicaciones de manera eficiente en plataformas de flujo de datos es una tarea crucial para los ingenieros de centros de datos y los diseñadores de estrategias de procesamiento en tiempo real. La correcta asignación de recursos y la optimización del rendimiento pueden tener un impacto significativo en la capacidad del sistema para manejar grandes volúmenes de datos y mantener una baja latencia. El uso de herramientas de modelado y simulación permiten prever cómo se comportan las aplicaciones bajo diferentes configuraciones y condiciones de carga, simulando diversos escenarios, analizar el impacto de distintos parámetros, y ajustar las estrategias de implementación para maximizar el rendimiento y la eficiencia operativa. Como resultado, los ingenieros a menudo se enfrentan a desafíos significativos al intentar optimizar el rendimiento y la escalabilidad de sus aplicaciones en plataformas como Flink. Es necesario avanzar en la investigación y desarrollo de modelos de predicción de rendimiento más completos y precisos, proporcionando grandes beneficios en términos de eficiencia operativa y capacidad para manejar aplicaciones de flujo complejas.

1.2. Estructura del Documento

El documento está organizado de la siguiente manera. El Capítulo 1 presenta la introducción y principales contribuciones de este trabajo. El Capítulo 2 describe el marco conceptual y sus tópicos, y trabajos relacionados al estado del arte. El Capítulo 3 describe el problema, solución, objetivos y requerimientos de la solución. Capítulo 4 presenta el diseño tanto de alto y bajo nivel, interfaces y diseño de pruebas. El Capítulo 5 describe la implementación tanto en hardware, software, lenguajes de programación y estrategias. El Capítulo 6 presenta el estado de la aplicación de las pruebas definidas en el Capítulo 4. El Capítulo 7 detalla la instalación y documentación y el Capítulo 8, las conclusiones.

Capítulo 2

Marco Conceptual y Estado del Arte

En este capítulo, la Sección 2.1 presenta los conceptos generales de procesamiento de flujo y descripción de la arquitectura de la plataforma Apache Flink. También se describe los conceptos de modelamiento y simulación, luego modelado y simulación por eventos discretos y finalmente la especificación formal de sistemas de eventos discretos (DEVS). En la Sección 2.2, se expone trabajos relacionados con modelos de simulación, sobre todo los modelos de sistemas de procesamiento de flujo de datos y se presentan herramientas de simulación que usan formalismo DEVS.

2.1. Marco Conceptual

2.1.1. Procesamiento de flujo

El procesamiento de flujo es un proceso de análisis y procesamiento de datos en tiempo real a medida que se generan, proporcionando información sobre los datos que se pueden utilizar para tomar mejores decisiones. Aquel enfoque se contrasta con el procesamiento por lotes (*batch proccessing*) que analiza los datos después de haberlos recopilado y almacenado [15] (ver Figura 2.1).

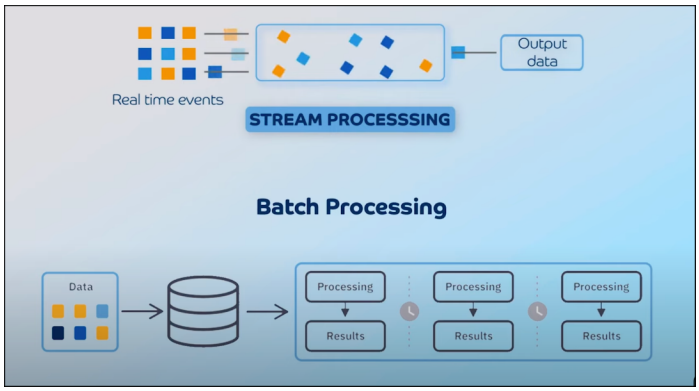

Figura 2.1: Procesamiento de flujo y por lotes.

- Los motores o plataformas de procesamiento de flujo de datos generalmente trabajan en tres fases:
  - **Ingesta**: obtención de datos de una variedad de fuentes, como dispositivos, sensores y aplicaciones (sistema de pagos, por ejemplo). Los datos generalmente se asocian a eventos que describen algo que sucedió.
  - **Proceso**: una vez que se han ingerido los datos, el procesador de flujo los procesa de acuerdo con el conjunto de reglas u operaciones como filtrar, transformar y agregar.
  - **Salida**: luego el procesador genera los datos procesados en una variedad de destinos como bases de datos, paneles y otras aplicaciones (ver Figura 2.2).



Figura 2.2: Fases de un procesador de flujo de datos.

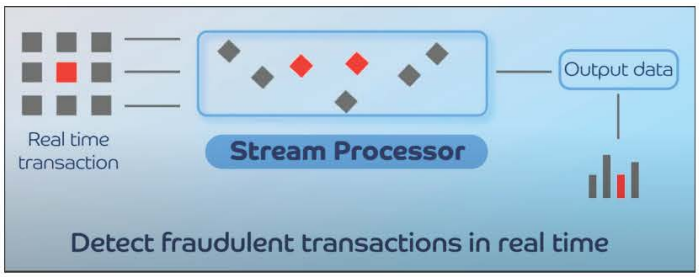

Figura 2.3: Aplicación de detección de fraudes en tiempo real.

En el ejemplo de Figura 2.3, hay que considerar lo que un sistema de procesamiento de flujo debe requerir para el análisis y detección de cada cuenta única en ciento de miles de personas que están generando transacciones en tiempo real. Similarmente ocurre en otras aplicaciones que incluyen monitoreo como negociación de acciones, procesamiento de tráfico web, etc. Entonces, esta clase de sistemas llamados frameworks de procesamiento de flujo distribuido (DSPF, *Distributed Stream Processing frameworks*) han emergido para facilitar el análisis de datos en tiempo real a gran escala. A medida que continúa el crecimiento exponencial de los dispositivos conectados a Internet, se puede esperar que la demanda de procesamiento de flujo aumente significativamente en los próximos años [1]. Hasta el día de hoy, existen muchos frameworks desarrollados para implementar, ejecutar y administrar aplicaciones basadas en flujo de datos. Entre los primeros frameworks se incluyen Aurora, Borealis [16], StreamIt [17] y SPADE [18]. Con la aparición de aplicaciones a escala de Internet en los últimos años, se han desarrollado nuevos modelos DSPF, como Apache S4, Apache Storm [19], Apache Flink [20][21], Apache Samza [22], Spark Streaming [23], Heron de Twitter [24] y Nepture [25], con soluciones comerciales que incluyen Google Millwheel [26], Azure Stream Analytics [27] y Amazon Kinesis [12]. Las aplicaciones de flujo de datos son implementadas en DSPF, e inerentemente deben ser desplegadas y ejecutadas distribuidamente en un conjunto de unidades de computo o cluster para garantizar velocidad de procesamiento del alto volumen de datos de manera continua.

En los motores DSPF modernos, se identifican algunas capas comunes. Estas capas se muestran en Figura 2.4:

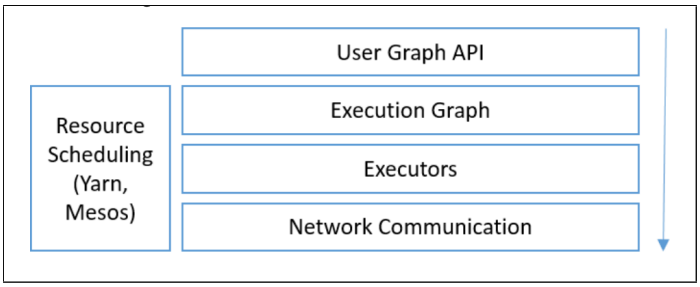

Figura 2.4: Arquitectura en capas DSPF [1].

- La capa superior define las API de usuario, donde una aplicación de flujo se define como un grafo lógico o topología.
- El grafo de usuario se convierte luego en un grafo de ejecución y los componentes de este se distribuyen a un conjunto de executors/workers en el cluster.
- Las comunicaciones de red se estabilizan entre los workers para completar el grafo.
- Las tres capas inferiores pueden trabajar con una capa de gestión de recursos como Apache Yarn [28] o Mesos [29] para adquirir los recursos de cluster necesarios para ejecutar el grafo de ejecución.

En resumen, el usuario mediante una API inyecta una aplicación especifica representada por un grafo lógico, la cual el framework distribuye las operaciones del la aplicación en el cluster.
Respecto a la aplicación, el flujo es una secuencia de tuplas o eventos ilimitados de la forma $(a_1, a_2, ..., a_n, t)$ generados continuamente en el tiempo, donde a_1 denota un atributo y t denota el tiempo [1]. El modelo de procesamiento de flujo es una topología representada por un grafo dirigido de nodos de operación y conectados a través de flujo de eventos (aristas). Entonces, los operadores procesan los mensajes y las aristas son enlaces de comunicación, y generalmente, el motor de ejecución convierte el grafo definido por el usuario en un grafo de ejecución [1].

- Un ejemplo de topología [1]: En Figura 2.5, un grafo definido para una aplicación de procesamiento con 3 operadores conectados por flujos. La Figura 2.6 muestra el grafo convertido en un grafo correspondiente de ejecución, donde se ejecutan en paralelo 2 instancias del operador S y 3 instancias del operador W distribuidos en el cluster.

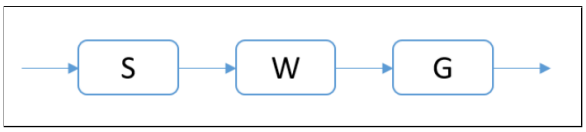

Figura 2.5: Grafo definido del usuario [1].

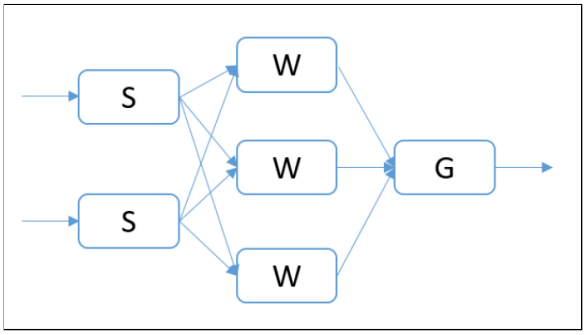

Figura 2.6: Grafo de ejecución [1].

Esta transformación del grafo de usuario al grafo de ejecución se implementa de forma diferente en los distintos DSPF. En general, una categoría de DSPF permite al usuario proporcionar información directa para convertir el grafo de usuario en un grafo de ejecución. La segunda categoría de DSPF convierte el grafo de usuario de un grafo de ejecución utilizando detalles implícitos. El grafo de ejecución se distribuye en un conjunto de procesos que se ejecutan en un cluster [1].

Los factores clave para diseñar el grafo de ejecución son:

1. Tolerancia a fallas
2. Garantías de entrega de mensajes
3. Rendimiento (throughput) vs latencia
4. Gestión de recursos y programación de operadores de flujo
5. Comunicación entre operadores

En este trabajo, se tiene como caso de estudio el DSPF Apache Flink. En Flink los operadores se conectan mediante los datos. Cada operador crea un flujo de datos y el siguiente operador trabaja en este flujo. Las aristas del grafo no las define el usuario y son creados por Flink basándose en los datos y los operadores definidos en ellos.
En la siguiente sección se presenta una descripción general y los principales componentes de la arquitetura de la plataforma de procesamiento de flujo Apache Flink.

2.1.2. Apache Flink

Descripción General

En Flink, las aplicaciones se componen de flujo de datos en tiempo real que pueden transformarse mediante operadores definidos por el usuario. Estos flujos de datos forman grafos dirigidos que comienzan con uno o más fuentes (*Sources*) y terminan en uno o más sumideros (*Sinks*) [2] (ver Figura 2.7).

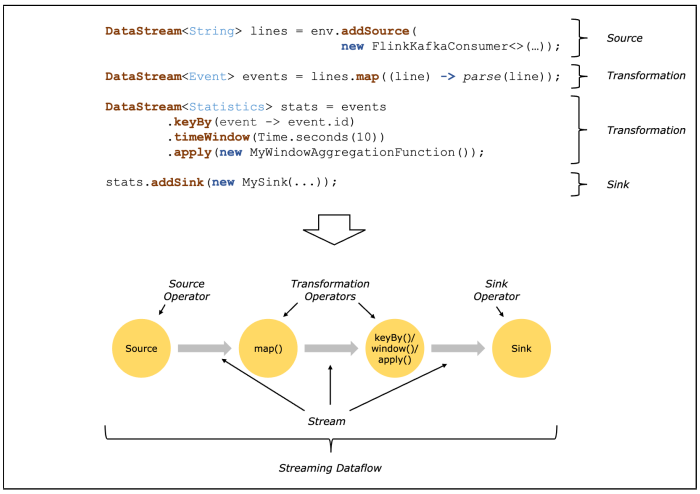

Figura 2.7: Aplicación representada en grafo lógico. [2]

Los Sources generan el flujo en una aplicación Flink y por lo general leen los eventos desde fuentes externas. Los operadores se encargan de realizar el procesamiento de flujo mediante una tarea determinada para cada tupla. Por lo general procesan una secuencia de entrada, pero también pueden procesar múltiples secuencias de entrada, y generar un flujo de salida que se utiliza para alimentar otros operadores. Una aplicación puede consumir datos en tiempo real de fuentes de transmisión, como colas de mensajes o registros distribuidos, como Apache Kafka o Kinesis. Pero Flink también puede consumir datos históricos acotados de una variedad de fuentes de datos. De manera similar, los flujos de resultados que produce una aplicación Flink se pueden enviar a una amplia variedad de sistemas que se pueden conectar con los Sinks [2] (ver Figura 2.8).

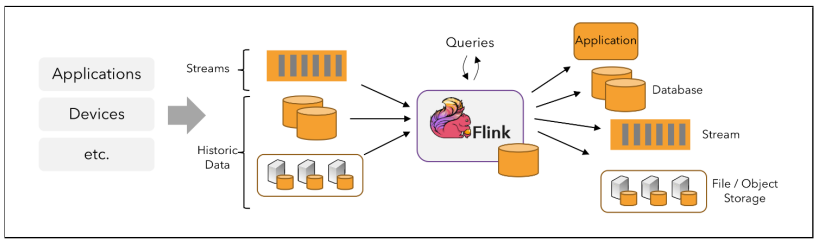

Figura 2.8: Interacción de sistemas con Flink. [2]

Para que una aplicación desplegada sea de alto rendimiento, los programas en Flink son inherentemente paralelos y distribuidos. Durante la ejecución, un flujo tiene una o más particiones de flujo y cada operador tiene una o más subtareas de operador. Las subtareas de operador son independientes entre sí y se ejecutan en diferentes threads (hilos de ejecución) y posiblemente en diferentes máquinas o contenedores. La cantidad de subtareas del operador es el paralelismo de ese operador en particular. Diferentes operadores del mismo programa pueden tener diferentes niveles de paralelismo [2] (ver Figura 2.9).

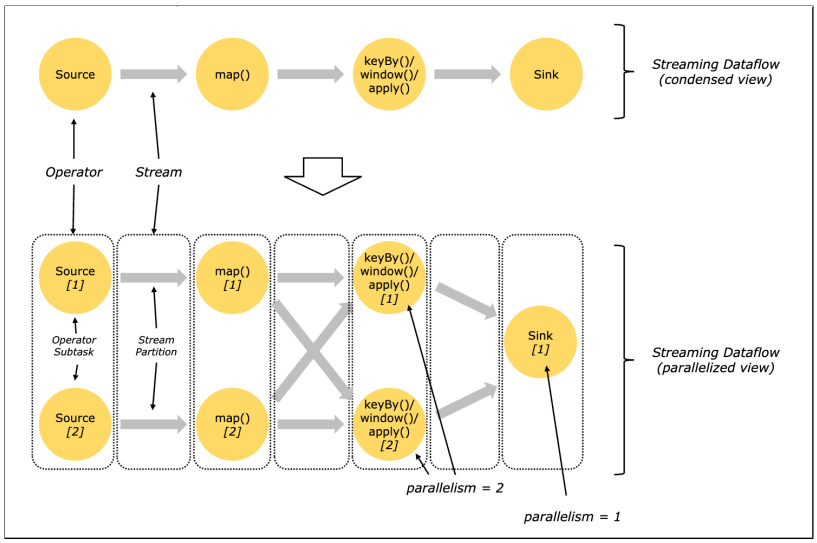

Figura 2.9: Grafo lógico con vista paralela. [2]

El flujo de datos se puede transportar entre dos operadores en un patrón uno a uno (o de reenvío) o en un patrón de redistribución:

- **Flujo uno a uno** [2]:
  - Preservan partición y el orden de los elementos.
  - A modo de ejemplo, en la Figura 2.9 en la vista paralela, entre Source y operador map, map()[1] verá los mismo elementos en el mismo orden en que fueron producidos por el operador Source[1].
- **Redistribución de flujo** [2]:
  - A modo de ejemplo, en la Figura 2.9 en la vista paralela, entre map y keyBy/window, así como entre keyBy/window y Sink cambia la partición de los flujos.
  - Cada subtarea del operador envía datos a diferentes subtareas de destino, según la transformación seleccionada:
    - keyBy() (que reparticiona mediante una clave en forma de arreglo asociativo (hash))
    - broadcast() o rebalance() (que reparticiona aleatoriamente)
  - En un intercambio de redistribución, el orden entre los elementos solo se conserva dentro de cada par de subtareas de envío y recepción (por ejemplo, subtask[1] de map y subtask[2] de keyBy/window)
  - se debe tener en cuenta que la redistribución entre los operadores keyBy/window y Sink introduce un no determinismo con respecto al orden en el que los resultados agregados para las diferentes claves llegan al Sink.

Arquitectura Flink

Flink es un sistema distribuido y requiere una asignación y gestión eficaz de recursos informáticos para ejecutar aplicaciones de flujo de datos. Se integra con todos los administradores de recursos de cluster habituales, como Hadoop YARN [30] y Kubernetes [31], pero también se puede configurar para que se ejecute como un cluster independiente o incluso como una biblioteca [3].
El entorno de ejecución de Flink consta de dos tipos de procesos: un **JobManager** y uno o más **TaskManagers**, como se ilustran en Figura 2.10.

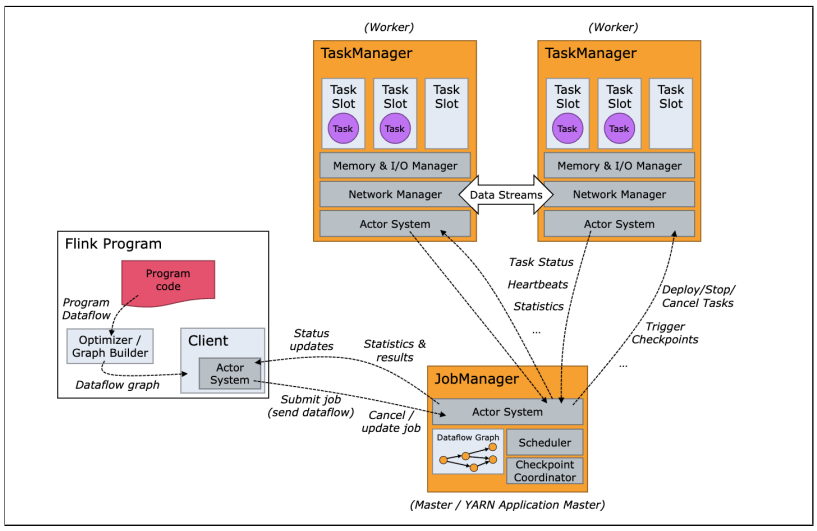

Figura 2.10: Modelo de entorno de ejecución Flink. [3]

El cliente compila aplicaciones de flujo en un grafo de flujo de datos, que luego se envía al JobManager. Los JobManager y TaskManagers pueden iniciarse de varias maneras: directamente en las máquinas como un cluster independiente, en contenedores o administrados por marcos de recursos como YARN. Los TaskManagers se conectan a los JobManager, se anuncian como disponibles y se les asigna trabajo.

- **JobManager** [3]:
  - Es el proceso central que coordina la ejecución distribuida de una aplicación (*job*). Su función principal es la planificación y el seguimiento del estado de la aplicación.
  - En general, el JobManager recibe la aplicación, lo divide en tasks, asigna esos tasks a los TaskManagers y supervisa la ejecución de la aplicación. También maneja la recuperación ante fallos y el almacenamiento de estado (checkpointing). Todo el proceso lo involucra con 3 componentes:
  - **Dispatcher**:
    - Habilita una interfaz REST para enviar aplicaciones Flink para su ejecución.
    - Un cliente envía una solicitud para iniciar una nueva aplicación de procesamiento de datos, y el Dispatcher se encargará de recibir la solicitud, enviándola al JobManager.
    - Inicia un nuevo JobMaster para cada aplicación enviada.

    - También ejecuta la interfaz web de Flink para proporcionar información sobre las ejecuciones de aplicaciones.
  - **Resource Manager**:
    - Responsable de la asignación y el aprovisionamiento de recursos en un cluster de Flink.
    - Administra las unidades de memoria (*task slot*).
    - Flink implementa varios ResourceManagers para diferentes entornos y proveedores de recursos (YARN, Kubernetes).
    - En una configuración independiente, ResourceManager sólo puede distribuir las unidades de memoria de los TaskManagers disponibles y no puede iniciar nuevos TaskManagers por sí solo.
  - **JobMaster**:
    - Responsable de gestionar la ejecución de un único JobGraph [32].
    - Recibe el JobGraph del Dispatcher y se encarga de planificar y coordinar la ejecución del job.
    - Se pueden ejecutar varios jobs simultáneamente en un cluster de Flink, cada uno con su propio JobMaster.

  Siempre hay al menos un JobManager. Una configuración de alta disponibilidad [33] puede tener varios JobManagers, uno de los cuales siempre es el líder y los demás son de reserva.

- **TaskManagers** [3]:
  - Ejecutan el trabajo real en Flink.
  - Ejecutan las tareas de un flujo de datos y almacenan en buffer e intercambian los flujos de datos.
  - Siempre debe haber al menos un TaskManager.
  - La unidad más pequeña de programación de recursos en un TaskManager es un task slot.
  - La cantidad de task slots en un TaskManager indica la cantidad de tareas de procesamiento concurrente (*thread*).
  - Se pueden ejecutar varios operadores en un task slot.

Tareas y cadenas de operadores [3]

Para la ejecución distribuida, Flink encadena las subtareas (*subtasks*) de operadores

en tareas (*tasks*). Cada tarea es ejecutada por un hilo (*thread*). Encadenar operadores en tareas es una optimización útil: reduce la sobrecarga de la transferencia y el almacenamiento en buffer de hilo a hilo, y aumenta el rendimiento general al tiempo que disminuye la latencia. El comportamiento del encadenamiento se puede configurar [34]. El flujo de datos en Figura 2.11 se ejecuta con cinco subtareas y, por lo tanto, con cinco hilos paralelos.

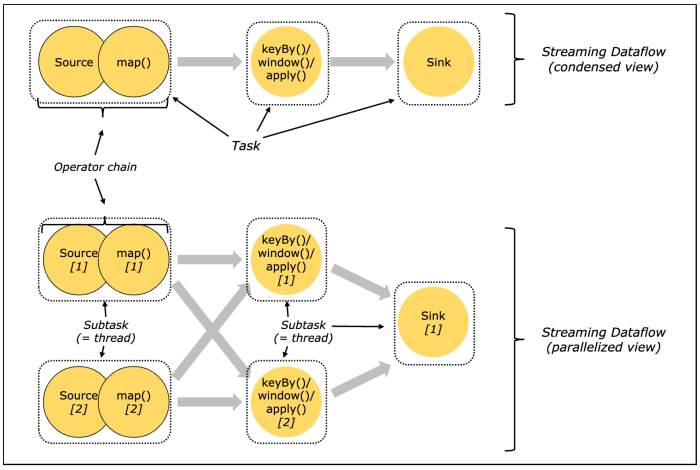

Figura 2.11: Aplicación de flujo con 5 subtareas. [3]

Espacios para tareas y recursos [3]

Cada TaskManager es un proceso de JVM (*Java Virtual Machine*) y puede ejecutar una o más subtareas en hilos separados. Para controlar cuántas tareas acepta un TaskManager, tiene las llamadas *task slots* (al menos una):

- Cada task slot representa un subconjunto fijo de recursos del TaskManager.
- Un TaskManager con tres slots, por ejemplo, dedicará 1/3 de su memoria administrada a cada slot.
- La asignación de recursos a los slots significa que un subtask no competirá con subtasks de otros jobs por la memoria administrada, sino que tendrá una cierta cantidad de memoria administrada reservada.
- Aquí no se produce ningún aislamiento de CPU, los slots solo separan la memoria administrada de los tasks.
- Al ajustar la cantidad de task slots, los usuarios pueden definir cómo se aíslan los subtasks entre sí:

- Tener un slot por TaskManager significa que cada grupo de tasks se ejecuta en una JVM independiente.
- Tener múltiples slot en TaskManager significa que más subtask comparten la misma JVM.
- Los tasks en la misma JVM comparten conexiones TCP.
- También pueden compartir conjuntos de datos y estructuras de datos, lo que reduce la sobrecarga por tarea.

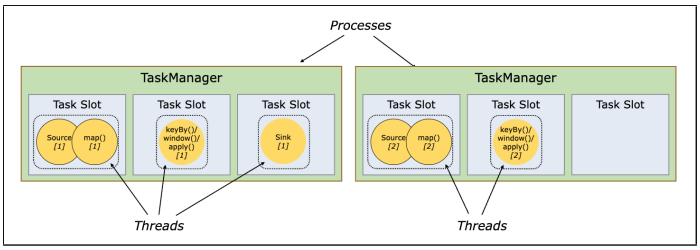

Figura 2.12: Subtareas en task slots. [3]

2.1.3. Conceptos de Modelamiento y Simulación

Los seres humanos, impulsados por la curiosidad y la necesidad de resolver problemas, han recurrido históricamente a la experimentación como principal método para adquirir conocimiento. En este contexto, se distinguen dos elementos clave: la entidad en estudio y el marco experimental (EF) (ver Figura 2.13), el cual establece las condiciones bajo las cuales se realiza la experimentación y se obtienen los resultados [4].

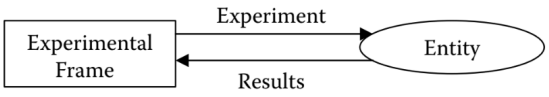

Figura 2.13: Resolución de problemas mediante la experimentación. [4]

Sin embargo, la experimentación directa no siempre es factible debido a restricciones éticas, riesgos o altos costos, e incluso puede ser imposible en ciertos casos, como estudiar fenómenos astronómicos. Ante estas limitaciones, una alternativa efectiva es la construcción de modelos que abstraen el problema real, permitiendo razonar sobre él de manera controlada. Esta capacidad de modelar forma parte natural del pensamiento humano y facilita una mejor comprensión del sistema en estudio [4].

A lo largo de los últimos siglos, las ecuaciones diferenciales desarrolladas por Newton y Leibniz han sido la herramienta predominante para el modelado formal de sistemas, proporcionando un enfoque matemático riguroso para analizar fenómenos complejos [35]. En este contexto, la experimentación sigue siendo útil para recolectar datos que determinan la construcción del modelo. El marco experimental, por su parte, define los supuestos y las preguntas válidas que pueden formularse, otorgando contexto y orientación al proceso de modelado [4].

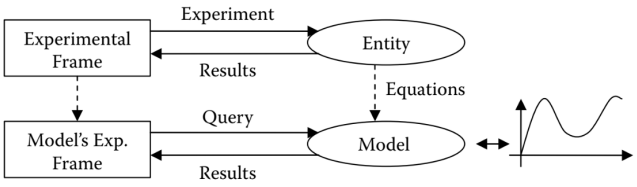

Figura 2.14: Resolución de problemas mediante modelado analítico. [4]

El modelo, junto con el marco experimental, permite reproducir condiciones de un sistema real para obtener resultados útiles, como estimar la temperatura en un aula en función de la distribución de estudiantes. Estos resultados pueden servir para modificar el sistema original. Incluso en ausencia de datos empíricos, es posible construir modelos teóricos que luego pueden validarse mediante comparación con observaciones reales, teorías existentes o sistemas similares [4]. Para que un modelo sea considerado válido, en En Figura 2.14 presenta un comportamiento que debe coincidir con el de la entidad original, lo que requiere una fase de validación rigurosa. Cuando los modelos son demasiado complejos para resolverse analíticamente, se recurre a métodos numéricos, como la discretización mediante ecuaciones diferenciales, que proporcionan soluciones aproximadas pero suficientemente precisas para muchos propósitos [36]. Con el uso de métodos numéricos, surge la necesidad de verificar la exactitud de los resultados obtenidos, comparándolos con datos experimentales para dimensionar cuánto se aproxima el modelo al comportamiento del sistema real (ver Figura 2.15). Este enfoque impulsó avances significativos en la ciencia y tecnología, promoviendo el desarrollo de nuevas técnicas matemáticas para enfrentar problemas cada vez más complejos. En el siglo XX, la creciente complejidad de los sistemas —como los de control y monitoreo del tráfico— superó las capacidades de los métodos analíticos tradicionales, haciendo necesaria la adopción de enfoques más sofisticados y automatizados [4].

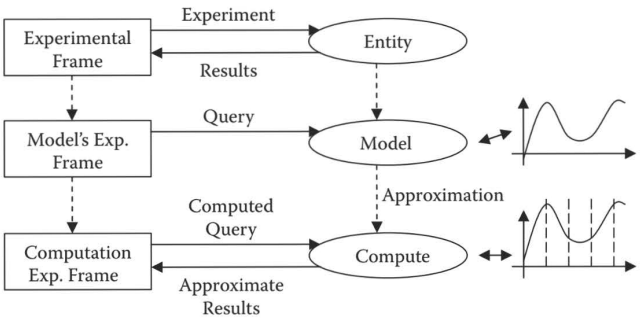

Figura 2.15: Resolución de problemas mediante computación. [4]

El uso de ordenadores ha transformado las técnicas de modelado numérico en simulaciones, reduciendo errores y acelerando el análisis de sistemas complejos. El marco experimental se implementa ahora como programas que generan casos de prueba para aproximar el comportamiento del modelo. Gracias a la reducción de costos de hardware y mejoras en interfaces y lenguajes, la simulación se ha convertido en una herramienta accesible, consolidándose como una metodología flexible y en constante evolución para la resolución de problemas [4].

Modelo de Simulación

El modelo es una versión simplificada de la realidad que se construye teniendo en cuenta las condiciones de experimentación del sistema de interés, incluyendo su entorno de trabajo y dominio de aplicación. Así, el modelo queda limitado al marco experimental en el que fue desarrollado, lo que afecta su construcción, experimentación y validación.
El modelo se utiliza para crear un simulador, que ejecuta las instrucciones del modelo y genera su comportamiento. Al implementarlo, se produce una reducción adicional en la precisión debido a las limitaciones de los lenguajes de programación y del ordenador, que afectan factores como la duración de la simulación, la memoria disponible y la precisión de las variables. La Figura 2.16 ilustra dos relaciones clave: verificación y validación. La validación relaciona el modelo con el sistema real dentro del marco experimental, evaluando en qué medida el comportamiento del modelo coincide con el del sistema bajo condiciones específicas. La verificación, por su parte, conecta el simulador con su modelo, enfocándose en la precisión del simulador al ejecutar las instrucciones del modelo, lo que puede verse afectado por la precisión de la computadora y los algoritmos de simulación [4].

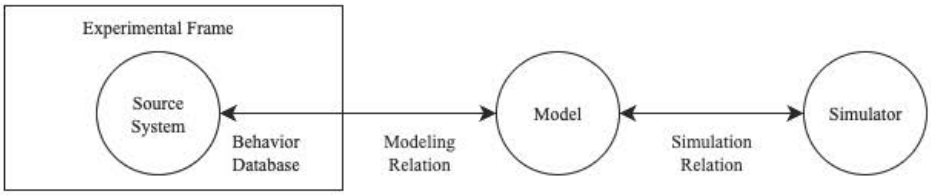

Figura 2.16: Entidades básicas en Modelo & Simulación y sus relaciones. [4]

2.1.4. Simulación de Eventos Discretos (SED)

El modelo de eventos discretos supone que, aunque el tiempo es continuo, solo puede ocurrir un número finito de eventos en un período determinado. Por lo tanto, un simulador SED puede ser muy eficiente porque solo se necesita representar los cambios de estado cuando ocurren los eventos en momentos específicos a medida que avanza el tiempo [4]. Ejemplos de sistemas incluyen colas en un banco, sistemas de fabricación y gestión de inventarios. Estos sistemas son impulsados por eventos que suceden en momentos discretos, como llegadas o salidas de clientes, fallas de máquinas, etc [37].
En la década de 1960, las técnicas SED ganaron popularidad, lo que llevó a la creación de lenguajes de simulación avanzados como SLAM, Arena, Simula y SimScript [38]. Aunque estos lenguajes permiten abordar problemas complejos, carecen de la formalidad de métodos de modelado, lo que dificulta la prueba, mantenimiento y verificación del software resultante. Además, los cambios en el lenguaje pueden afectar gravemente a los modelos existentes, ya que su semántica no suele estar formalmente definida. Estos lenguajes no ofrecen un enfoque suficientemente abstracto para abordar problemas o probar propiedades de las entidades, para mejorar la calidad del análisis y reducir costos [4].
Diferentes grupos han investigado técnicas formales para el modelado de eventos discretos, que ofrecen un lenguaje matemático como guía para especificar la semántica del sistema de manera no ambigua. Estos formalismos permiten representar las entidades en estudio, crear modelos abstractos y traducirlos en modelos ejecutables. Un modelo formal proporciona un mecanismo robusto para especificar la entidad, permitiendo su verificación formal, lo que mejora la detección de errores y reduce el tiempo de desarrollo y verificación del software de simulación, aumentando la seguridad y reduciendo costos [4].
La siguiente sección se centra en uno de estos formalismos de modelo y simulación de eventos discretos, llamado DEVS (Especificación de Sistemas de Eventos Discretos), una sólida teoría que ha evolucionado en los últimos 30 años [4].

2.1.5. Especificación Formal de Sistemas de Eventos Discretos (DEVS)

Dentro de la simulación dinámica de eventos discretos existe un formalismo llamado DEVS (Discrete Event System Specification) para el modelado y el análisis de sistemas de eventos discretos que proporciona los medios para describir tales sistemas mediante el uso de dos tipos diferentes de componentes: modelos atómicos y acoplados. Aquí, se utiliza la definición de DEVS con puertos (en lugar del DEVS clásico [39]), explicitando canales de entrada y salida, lo que permite la comunicación directa y estructurada entre modelos a través de puertos designados.
Los modelos atómicos son las entidades más elementales para construir representaciones de sistemas. Sus estados cambian al recibir un evento de entrada o después de un retardo. Los modelos acoplados, por el contrario, permiten la construcción jerárquica de modelos al agrupar varios modelos DEVS en uno compuesto, lo que permite la reutilización de modelos. Los modelos acoplados pueden considerarse, gracias a la propiedad de clausura, como otro modelo DEVS con un comportamiento equivalente a un modelo atómico.
Los modelos acoplados pueden tener sus propios puertos de entrada y salida. Al ocurrir un evento externo, un modelo acoplado redirige la entrada a uno o más de sus componentes (EIC). Además, cuando un componente produce una salida, esta debe asignarse como entrada de otro componente (IC) o como salida del propio modelo acoplado (EOC). Mientras que los modelos atómicos representan el comportamiento de entidades individuales, los modelos acoplados definen la estructura y las interacciones dentro del sistema.

Una definición formal de un modelo atómico con puertos se describe mediante la tupla [4] [39]:

$$DEVS = (X, Y, S, \delta\text{int}, \delta\text{ext}, \lambda, \text{ta})$$

donde

$X = \{(p, v) \mid p \in \text{InPorts},\ v \in \text{X}_\text{p}\}$ es el conjunto de puertos de entrada y valores correspondientes
$Y = \{(p, v) \mid p \in \text{OutPorts},\ v \in \text{Y}_\text{p}\}$ es el conjunto de puertos de salida y valores correspondientes
S es un conjunto de estados secuenciales
$\delta\text{int} : S \times S \to S$ es la función de transición interna
$\delta\text{ext} : Q \times X \times S \to S$ es la función de transición externa, donde
$Q = \{(s, e) \mid s \in S,\ 0 \leq e \leq ta(s)\}$ es el conjunto de estados totales
e es el tiempo transcurrido desde la última transición
$\lambda : S \to Y$ es la función de salida
$\text{ta} : S \to \mathbb{R}_0^+ \cup \infty)$ es la función de avance de tiempo

Un modelo acoplado se define formalmente como [4] [39]:

$$N = (X, Y, D, Md \mid d \in D, EIC, EOC, IC, Select)$$

donde X y Y conjunto de puertos como en la definición formal de un modelo atómico y D es el conjunto de nombres de componentes.

Requerimientos de componentes:
Componentes son modelos DEVS:
Por cada d ∈ D

$$M_{\mathrm{d}} = (X_{\mathrm{d}}, Y_{\mathrm{d}}, S, \delta ext, \delta int, \lambda, ta)\ es\ un\ DEVS$$
$$with\ X_{\mathrm{d}} = (p, v) \mid p \in IPorts_{\mathrm{d}}, v \in X_{\mathrm{p}}$$
$$Y_{\mathrm{d}} = (p, v) \mid p \in OPorts_{\mathrm{d}}, v \in Y_{\mathrm{p}}$$

Requerimientos de acoplamiento:
El acoplamiento de entrada externa conecta las entradas externas a las entradas del componente:

$$\mathrm{EIC} \subseteq \{((N, \mathrm{ip_N}), (d, \mathrm{ip_d})) \mid \mathrm{ip_N} \in \mathrm{IPorts},\ d \in \mathrm{D},\ \mathrm{ip_d} \in \mathrm{IPorts_d}\}$$

El acoplamiento de salida externa conecta las salidas de los componentes a las salidas externas:

$$\mathrm{EOC} \subseteq \{((d, \mathrm{op_d}), (N, \mathrm{op_N})) \mid \mathrm{op_N} \in \mathrm{OPorts},\ d \in \mathrm{D},\ \mathrm{op_d} \in \mathrm{OPorts_d}\}$$

El acoplamiento interno conecta las salidas de los componentes a las entradas de los componentes:

$$\mathrm{IC} \subseteq \{(a, \mathrm{op_a}), (b, \mathrm{ip_b})) \mid a, b \in \mathrm{D},\ \mathrm{op_a} \in \mathrm{OPorts_a},\ \mathrm{ip_b} \in \mathrm{IPorts_b}\}$$

No se permiten bucles de retroalimentación directa, es decir, ningún puerto de salida de un componente puede estar conectado

$$((d, \mathrm{op_d}), (e, \mathrm{ip_d})) \in IC \implies d \neq e$$

$\mathrm{Select} : 2^D \to D,$ es la función de TIE-BREAKING.

La función TIE-BREAKING es utilizada en DEVS que determina cuál función de transición llamar primero cuando suceden eventos inminentes tanto de entrada como de salida en un modelo, lo cual es seleccionar que función llamar de acuerdo al modelo que se está simulando.

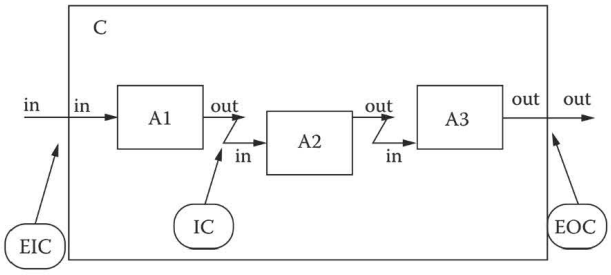

Figura 2.17: Un modelo acoplado. [4]

2.2. Estado del Arte

2.2.1. Trabajos relacionados con simulación de procesamiento de flujo

Alonso Inostrosa-Psijas et al. [6] propusieron simular una aplicación para el análisis de transmisiones de Twitter utilizando Apache Storm [19]. Evaluaron el rendimiento del simulador, denominado SimStream, en diferentes cargas de trabajo, logrando reproducir métricas de una aplicación real con un error mínimo. El trabajo incluye una descripción de los componentes de sistemas de procesamiento de flujo y la arquitectura de Apache Storm, así como de SimStream, desarrollado en C++ y basado en un modelo orientado a procesos utilizando la librería libCppSim [40]. El simulador permite crear perfiles para medir el rendimiento, considerando nodos Spouts y Bolts, hardware y redes de comunicación (red Fat-Tree [41]). Los autores definieron tres archivos de configuración para inicializar el simulador, especificando la topología y los parámetros de cada componente. Presentaron un caso de estudio con una topología simple que incluye un productor de Kafka para leer tweets y un procesador de Twitter que calcula análisis de sentimiento y hashtags top-k. Finalmente, evaluaron la aplicación Storm simulada en términos de throughput, tiempos de respuesta y utilización, encontrando un comportamiento similar entre la simulación y la aplicación real. Al variar la tasa de llegada de tuplas y realizar experimentos con y sin replicación, demostraron que el simulador puede reproducir resultados del sistema real y manejar cuellos de botella al aumentar réplicas de componentes.

La tesis de César Escobal Blanco [8] realizó un trabajo similar mencionado anteriormente [6], centrado en el uso de Redes de Petri Coloreadas [42] para simular un modelo de una aplicación que opera en la plataforma de procesamiento de flujo Apache Storm [11]. Este modelo, con un diseño modular y jerárquico, permite evaluar el rendimiento de la aplicación en distintos escenarios. El autor describe el paradigma de procesamiento de flujo y la arquitectura de Storm, seguido de una introducción a las Redes de Petri y

las Redes de Petri Coloreadas. Utiliza la herramienta CPNTools [43] (ahora reemplazada por CPN IDE) para su simulación. Su caso de estudio se enfoca en una aplicación de análisis de sentimientos que combina Kafka y Storm para procesar tweets en tiempo real [44], con componentes clave como un productor de Kafka y un procesador de Twitter. Escobal modela la aplicación con Redes de Petri Coloreadas, presentando el modelo general, submodelos, y las declaraciones necesarias. Evalúa el comportamiento del modelo utilizando CPNTools, llevando a cabo ejecuciones en la plataforma real de Storm con tweets de 2021 y 2022, procesando lotes de hasta 100.000 tweets. Los resultados mostraron que el modelo generado presenta un error porcentual cercano a cero al compararlo con el sistema real, indicando que es una representación precisa. Esto sugiere que el modelo es una herramienta valiosa para evaluar el rendimiento y escalabilidad del sistema Apache Storm en la aplicación de análisis de sentimientos.

Los autores Veronica Gil-Costa, Emilio Tapia y Mauricio Marín [45] propusieron un protocolo de simulación asincrónica que se puede ejecutar en la plataforma de procesamiento de flujo S4. Este simulador fue desarrollado para simular la ejecución de consultas de usuarios en un motor de búsqueda web. El simulador propuesto controla el avance de tiempo virtual en cada proceso lógico (LP) por medio de 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 oráculo utilizada para calcular de manera adaptativa el valor de la ventana de tiempo a través de la simulación. El mecanismo basado en dos barreras está destinado a reducir la cantidad de eventos rezagados en los procesos lógicos. A pesar de que el caso de estudio simula un motor de búsqueda Web, los autores mencionan la posibilidad de modelar y simular una plataforma de procesamiento de flujo utilizando este simulador.

El autor M.C. Kaptein [46] desarrolló RStorm, un paquete R para el desarrollo y evaluación de algoritmos de flujo de datos. RStorm es un paquete de simulación destinado a los desarrolladores para ayudarlos a analizar y evaluar sus algoritmos de flujo de una manera fácil y sin las dificultades de una implementación real de una plataforma de procesamiento de flujo dada. RStorm recurre a la misma terminología y conceptos utilizados en Storm, proporcionando una representación gráfica de algoritmos de flujo. Sin embargo, este paquete es solo para analizar el algoritmo y no incluye el costo del hardware (como la comunicación, múltiples threads por nodo, etc.).

Hay algunos simuladores de procesamiento de flujo para la nube como CEPSim [47]. CEPSIm se utilizó principalmente para el estudio de la escalabilidad y el rendimiento de los sistemas de procesamiento de eventos complejos (CEP) y los efectos de diferentes estrategias de procesamiento de consultas. Se centra en el procesamiento de consultas. CEPSim se creó sobre la herramienta CloudSim.

2.2.2. Otros trabajos relacionados con simulación

Los autores de [5] proponen un modelo de simulación del proceso de carga y transporte en la minería chilena, utilizando el formalismo DEVS (Discrete Event System Specification). Este modelo se centra en optimizar la gestión de los costos operativos, dado que el proceso de carga y transporte puede representar entre el 45 % y el 65 % del costo total operacional en una faena minera. La simulación permite experimentar con diferentes escenarios, facilitando la toma de decisiones para la flota de camiones y mejorando la eficiencia del proceso. El enfoque utiliza la simulación por eventos discretos (DES) para representar el comportamiento de las entidades involucradas, donde cada evento provoca cambios en el sistema. A través de este modelo, los autores buscan proporcionar una herramienta analítica que ayude a los despachadores a gestionar y optimizar las operaciones mineras, destacando la utilidad de la simulación en sistemas complejos donde los métodos analíticos pueden resultar insuficientes.

Los autores de [48] presentaron un estudio sobre motores de búsqueda web, que son sistemas complejos operados en grandes grupos de procesadores. Dada la creciente cantidad de usuarios, es esencial evaluar continuamente la capacidad de estos motores. Para ello, se utilizan métodos de planificación de capacidad que ayudan a estimar el número necesario de procesadores en sistemas de cluster. La investigación propone una metodología que combina simulación de eventos discretos con análisis operativo clásico, centrándose en sistemas abiertos. Esta metodología permite evaluar solo una fracción de las implementaciones para determinar la asignación óptima de recursos según una carga de trabajo específica. Además, se presenta un módulo de planificación de capacidad basado en simulación, integrado con una implementación OpenMP+MPI de un motor de búsqueda web, que ajusta de manera flexible los recursos computacionales en función de las métricas de rendimiento. Los resultados indican que la metodología es efectiva para ofrecer configuraciones adaptativas, asegurando la disponibilidad de recursos para satisfacer eficientemente las demandas de los usuarios.

Los autores de [49] presentaron una estrategia paralela optimista aproximada y semiasíncrona basada en la Especificación del Sistema de Eventos Discretos (DEVS) y su extensión, Parallel DEVS (PDEVS). Esta estrategia está diseñada para la simulación de motores de búsqueda web, sistemas complejos que operan en grandes grupos de procesadores. La propuesta gestiona el nivel de optimismo de manera eficiente durante la ejecución de la simulación, ajustándolo automáticamente según su evolución. Además, la carga y la partición del modelo se realizan automáticamente mediante un algoritmo que aprovecha el patrón de comunicación del modelo simulado. Los resultados indican que esta metodología reduce tanto los tiempos de ejecución como el uso de memoria en simulaciones optimistas estándar, aunque con pequeños errores aceptables. La investigación

sugiere que la estrategia puede mejorar significativamente el rendimiento en la simulación de sistemas complejos en entornos de búsqueda web.

Los autores de [50] presentaron un enfoque para modelar y simular el comportamiento de personas durante evacuaciones masivas en áreas costeras. Este modelo tiene como objetivo mejorar la eficiencia computacional en la evaluación de planes de evacuación, utilizando simulaciones informáticas en lugar de depender únicamente de simulacros costosos y difíciles de coordinar. La propuesta divide el área geográfica en celdas de tamaños fijos y, para optimizar los cálculos en cada paso de tiempo, considera solo a los agentes en celdas vecinas cercanas. El modelo combina datos geográficos, del censo público, y características demográficas para lograr resultados realistas, minimizando el error en la toma de decisiones y permitiendo una mejor estimación de la capacidad de los grupos para alcanzar zonas seguras. La simulación se implementó utilizando programación basada en agentes en C++ y OpenMP, y fue evaluada con datos reales de las ciudades chilenas de Iquique y Viña del Mar, así como de Kesennuma en Japón. Los resultados demuestran que el enfoque propuesto es efectivo para simular evacuaciones de manera precisa y eficiente.

2.2.3. Herramientas de simulación usando formalismo DEVS

Existen diversas herramientas que permiten simular modelos formalizados mediante DEVS, que se pueden encontrar en los trabajos [51][52], donde presentan y comparan la mayoría de las herramientas descritas en la Tabla 2.1.

Tabla 2.1: Herramientas DEVS.

Nombre	Lenguaje	Descripción	Documentación
aDEVS	C++	Es una biblioteca para construir simulaciones basadas en los formalismos Parallel DEVS y Dynamic DEVS (dynDEVS)	No Disponible
CD++	C++	CD++ es un kit de herramientas DEVS que proporciona una biblioteca de clases de C++ para especificar modelos en varios formalismos como CDEVS, PDEVS y principalmente enfocado en Cell-DEVS [53]	http://cell-devs.sce.carleton.ca/
Cadmium	C++	Es una biblioteca de solo encabezado multiplataforma [54]. Esta herramienta viene de una mejora realizada a CD++ gracias a la inclusion de la librería CD-Boost.	https://www.sce.carleton.ca/courses/sysc-5104/lib/exe/fetch.php?media=cadmium.pdf
VLE	C++	Virtual Laboratory Environment. Su nucleo consiste en un conjunto de bibliotecas (VFL), que implementan varios formalismos (redes de Petri, autómatas celulares 2D/3D, Sistemas de estado cuantificados (QSS), etc.) [55]	https://sourceforge.net/projects/powerdevs/
PyPDEVS	Python	PyPDEVS es una variante de PyDEVS [56], el cual solo tenía soporte para Classic DEVS, esta variante implementa Parallel DEVS [57].	http://msdl.cs.mcgill.ca/projects/DEVS/PythonPEVS

Capítulo 3

Definición del Problema y Análisis de Requerimientos

En este capítulo, en la Sección 3.1 se presenta la descripción del problema, así como en la Sección 3.2 se detalla la solución. En en la Sección 3.3 se describen los objetivos tanto general como específicos, en sección 3.4 se describe la metodología de desarrollo y por ultimo los requerimientos y diagramas 3.5.

3.1. Formulación del Problema

Las aplicaciones de procesamiento de flujo están concebidas para manejar en tiempo real un elevado volumen de tareas, requiriendo que los resultados sean lo más rapido posible [8]. Estas aplicaciones corresponden a topologías de grafos dirigidos como interpretación de la implementación del código del usuario. El usuario define operadores, comenzando con el que conecta a una fuente de datos, los operadores que aplican alguna operación sobre los datos, y operadores que depositan los resultados hacia otros sistemas. No basta solo con instanciar operadores individuales con sus respectivos propósitos. Debido a la alta demanda de la aplicación dada la inmensa cantidad de datos que deben ser procesados en tiempo real, la plataforma en la que es desplegada debe situarse en un sistema de alto desempeño, como lo es un cluster de computadores, de modo que las aplicaciones sean paralelas, es decir, poder replicar los operadores y distribuirlas en las unidades de cómputo para que sean ejecutadas de manera independiente. Bajo situaciones de alta demanda de trabajo, los operadores podrán encontrarse en un estado de sobrecarga, con un alto consumo de recursos, como lo es la CPU o múltiples nodos de un cluster, provocando demora en los tiempos de respuesta de las tareas procesadas.

3.2. Solución Propuesta

Es de suma importancia contar con el recurso necesario para ejecutar cierta aplicación de flujo de datos, lo cual lleva a la necesidad de experimentar y medir para planificar la capacidad de recurso requerido, pero sin recurrir al sistema real, mediante un modelo abstracto de la realidad usando la simulación. Por lo tanto, simular aplicaciones que se ejecuten sobre una plataforma de procesamiento de flujo puede ser útil para evaluar su rendimiento antes de ponerlos en producción. Uno de los paradigmas mas comunes de simulación son el modelado y el análisis de sistemas de eventos discretos (SED) que contepla una serie de eventos en marcas de tiempo discreto a lo largo del tiempo. DEVS (Discrete Event Specification) es un formalismo perteneciente a DES que proporciona los medios para describir tales sistemas mediante el uso de dos tipos diferentes de componentes: modelos atómicos y acoplados. DEVS permite modelar la complejidad del sistema bajo estudio al contar con modelos organizados jerárquicamente (es decir, organizados de tal manera que cada elemento es superior a su precedente) y los componentes de nivel superior del sistema se descomponen en elementos más simples [4].

Se propone el diseño e implementación de un modelo de simulación utilizando formalismo DEVS de plataformas de procesamiento de flujo, considerando la infraestructura y características de la plataforma. El caso de estudio es analizar la arquitectura y componentes de la plataforma Apache Flink para el comportamiento del modelo, conforme también con la ejecución de aplicaciones o alguna en específico bajo diferentes cargas de trabajo. El simulador debe conseguir métricas parecidas al finalizar una ejecución comparado con la ejecución real de una aplicación Flink, con el fin de que el simulador sea una herramienta de pruebas para predecir el rendimiento de un aplicación que se ejecuta sobre un motor de procesamiento de flujo de datos con niveles de replicación y cargas de trabajo, estimando la cantidad de hardware sin sobreestimar, para una ejecución eficiente por debajo de la utilización total del activo, evitando casos de cuello de botella. Cumplir con aquellos objetivos es necesario que el modelo incluya la arquitectura Flink, los recursos como el conjunto de procesadores y enlaces de comunicación. Se partirá con una granularidad gruesa e ir afinando, sin sobre afinar el modelo. Por último, las consideraciones por parte de los ejecutores de plataformas de procesamiento son al recurso a evaluar del cluster, la aplicación objetivo y cargas de trabajo aproximadas como parámetros principales del simulador.

3.3. Objetivos

3.3.1. Objetivo General

- Modelar y simular mediante formalismo DEVS plataformas de procesamiento de flujo bajo diferentes cargas de trabajo para evaluar el rendimiento y estimar recursos de hardware necesarios.

3.3.2. Objetivos Específicos

1. Revisar literatura y documentación de plataformas de flujo de datos y Apache Flink como caso de estudio.
2. Definir un modelo basado en formalismo DEVS del ambiente distribuido de ejecución junto con la arquitectura y componentes de la plataforma Flink.
3. Implementar el modelo en C++ utilizando la librería Cadmium [54].
4. Verificar el funcionamiento esperado del modelo.
5. Validar resultados obtenidos del simulador respecto a una aplicación real Flink.

3.4. Metodología

El desarrollo tiene un enfoque iterativo incremental, debido a que el modelo se refina a medida que se entiende mejor el sistema, dado un punto en el cual se aprecie si efectivamente el modelo de simulación tiene una correspondencia con el comportamiento de la plataforma. La metodología entonces, va de acuerdo a las etapas de generación del modelo de simulación:

- Construcción del modelo del sistema:
  - Identificación de las entidades principales del sistema y atributos.
  - Identificación y representación de las reglas que gobiernan el sistema que se quiere simular.
  - Captación de la naturaleza de las interacciones lógicas del sistema.
  - Verificación de que las reglas incorporadas al modelo son una representación válida de las del sistema que se modela.
  - Representación del comportamiento aleatorio.

- Iteraciones para la obtención del simulador Figura 3.1:

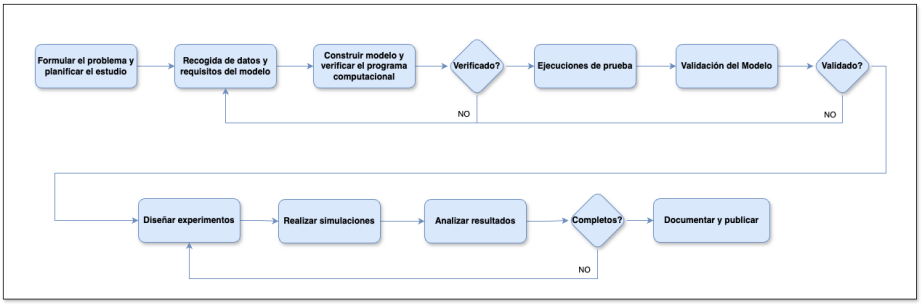

Figura 3.1: Flujo iterativo de modelado.

3.5. Especificación de Requerimientos

3.5.1. Requerimientos Funcionales

Realizar de experimentos con el simulador se requiere proceder a múltiples escenarios mediante la lectura de archivos externos de configuración. También, se necesita generar métricas a partir de registros para la evaluación del rendimiento, siendo los requerimiento funcionales presentes en la Tabla 3.1:

Tabla 3.1: Requerimientos funcionales.

Id	Nombre	Descripción	Historia
RF1	Parámetros (txt): tasas de llegada, hardware, topología y operadores	Los parámetros de la simulación deben ser mediante formato txt.	Como analista de un centro de datos quiero entregar archivos de configuración al simulador para ejecutar diversos escenarios.
RF2	Salidas desplegadas en gráficos: nivel de utilización, throughput vs tasa promedio de llegada, etc.	Los registros generados deben ser procesados para generar métricas visualizadas en gráficos para su comprensión.	Como analista de un centro de datos quiero conocer métricas obtenidas de la simulación por medio de la visualización de gráficos para evaluar el rendimiento del sistema.

3.5.2. Requerimientos No Funcionales

Hasta el momento el simulador solo es una herramienta que presta servicio de evaluar un sistema, por ende solo se formula según en requerimientos no funcionales de producto, presentados en la Tabla 3.2 (A futuro es posible que se requiera de un simulador eficiente para disminuir el tiempo de ejecución).

Tabla 3.2: Requerimientos no funcionales.

Id	Nombre	Descripción
RNF1	Producto.	El modelo de simulación será implementado con la herramienta Cadmium [54].
RNF2	Producto.	El modelo será 90 % implementado en C/C++ y el resto en Python.
RNF3	Producto.	El simulador será desarrollado en un SO basado en Unix.

Capítulo 4

Diseño

En este capítulo, presenta la etapa de diseño, en la cual se describe el modelado del sistema: en en la Sección 4.1 se presenta el diseño arquitectónico, es decir tecnologías y esquema de alto nivel del modelo. El diseño lógico en cuanto a la relación de componentes se presenta en Sección 4.2, el diseño de datos en Sección 4.3 y por ultimo en Sección 4.4 el diseño de pruebas unitarias, de integración y sistema.

4.1. Diseño Arquitectónico

4.1.1. Tecnologías utilizadas

Para el diseño e implementación del modelo de simulación se utilizan las siguientes tecnologías:

- **GCC/G++**: "GCC" es un término abreviado común para la Colección de compiladores GNU. Es el nombre más general del compilador y el nombre que se utiliza cuando el énfasis está puesto en la compilación de programas en C (la abreviatura antigua significaba "GNU C Compiler"). Cuando se hace referencia a la compilación de C++, es habitual llamar al compilador "G++". Dado que solo hay un compilador, también es preciso llamarlo "GCC" sin importar el contexto del lenguaje; sin embargo, el término "G++" es más útil cuando el énfasis está puesto en la compilación de programas de C++. G++ es un compilador, no un simple preprocesador. G++ genera código objeto directamente a partir del código fuente de su programa C++ [58].

- **C++**: Lenguaje de programación de propósito general orientado a la programación de sistemas que: es una mejora de C, admite abstracción de datos, programación orientada a objetos y programación genérica. C++ es un lenguaje para definir y utilizar abstracciones ligeras. Tiene importantes ventajas en áreas en las que el hardware

debe manejarse de forma eficaz y hay una complejidad significativa a la que enfrentarse. Esto incluye muchos sistemas con recursos limitados y mucho código básico y de infraestructura [59].

- **C++ STL**: La biblioteca de plantillas estándar proporciona un conjunto de componentes genéricos de C++ bien estructurados que funcionan juntos de manera perfecta [60].
    - ¿Qué es la STL?
        - Una colección de plantillas de clases y funciones componibles.
        - Permite la programación genérica en C++.
    - ¿Por qué utilizar la STL?
        - Reutilizar: “escribe menos, haz más, oculta detalles, compatibilidad”.
        - Flexibilidad: ”iteradores desacoplan los algoritmos de los contenedores”.
        - Eficiencia: ”evitar sobrecarga de funciones virtuales”.
    - Características de la STL
        - Contenedores (sequencial, asociativo, adaptadores).
        - Iteradores (entrada, salida, reenvío, acceso bidireccional y aleatorio).
        - Algoritmos genéricos (mutante, no mutante, clasificación y numérico).
- **Cadmium**: Es una biblioteca de solo encabezado multiplataforma implementada en C++ para modelado y simulación de eventos discretos, basada en el formalismo DEVS. DEVS es un paradigma de eventos discretos que permite una descripción jerárquica y modular de los modelos. Cada modelo DEVS puede ser conductual (atómico) o estructural (acoplado), y estar compuesto por entradas, salidas, variables de estado y funciones para calcular los siguientes estados y salidas. Esta herramienta viene de una mejora realizada a CD++ gracias a la inclusión de la librera CD-Boost [54].

4.1.2. Flujo de datos / Vista de alto nivel

Previo a la presentación del diseño lógico y de datos para el desarrollo de software, se presenta un modelo de alto nivel respecto a un sistema distribuido o cluster donde se despliegan las aplicaciones de procesamiento de flujo de datos. De acuerdo a lo mencionado en el marco conceptual sobre los componentes que involucran la ejecución de una aplicación Flink, se condensa en los siguientes puntos:

- El cliente envía un aplicación (*job*) de interés como un grafo de flujo de datos al motor de procesamiento de flujo en el nodo master.

- El JobManager recibe la aplicación, lo divide en tareas, asigna esas tareas a los TaskManagers y supervisa la ejecución de la aplicación.

El primer modelo se aprecia en Figura 4.1:

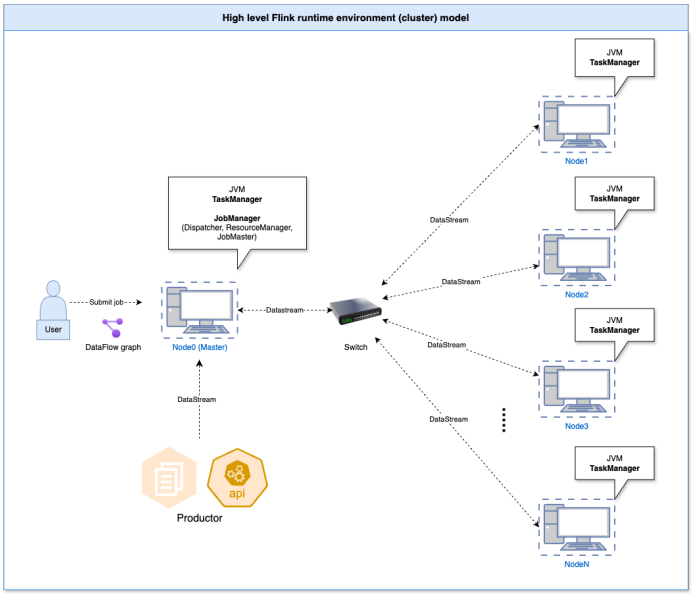

Figura 4.1: Modelo de alto nivel del entorno de ejecución de una aplicación de procesamiento de flujo de datos en un cluster.

Como se observa en Figura 4.1, el usuario envía una aplicación al JobManager para distribuirla en los trabajadores o TaskManagers, cada uno situado en un nodo de cómputo, los cuales para su comunicación de flujo de datos se representa mediante un conmutador. Desplegada la aplicación, comienza la generación de tuplas de datos mediante un productor, normalmente una API o archivo que la aplicación consume.

En el momento que al JobManager recibe un grafo de flujo de datos, hay que proceder a la gestión del recurso necesario a asignar a los operadores, que en este punto pasan a ser tareas, y cada uno con un nivel de replicación, pasando a ser subtareas, es decir, para ser ejecutadas paralelamente. JobManager tiene un componente para la asignación de recursos llamado ResourceManager, por lo tanto se puede pensar en una entidad el cual distribuye cada subtarea en el conjunto de nodos, para que cada una le toque una parte del total, coloquialmente dicho. Lo anterior se puede representar como el modelo visualizado en Figura 4.2, considerando un trabajo recibido con cierta cantidad de operadores y réplicas distribuido en un cluster de 4 nodos de cómputo.

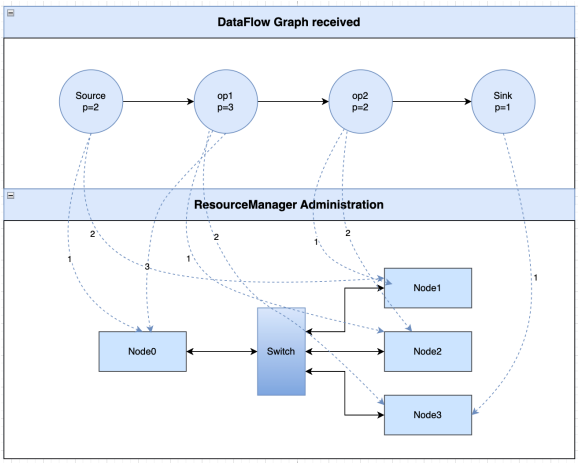

Figura 4.2: Modelo de alto nivel de un DataFlow graph administrado por el ResourceManager.

Hasta ahora, la administración del ResourceManager de forma exacta falta profundizar, pero el modelo incluye esta acción del ResourceManager que asigna recursos de forma equitativa ubicando las subtareas en el cluster total disponible. Aquello se aprecia mediante las flechas junto al nivel i de réplica de los operadores hacia los nodos, representando un patrón de asignación apreciada en la Tabla 4.1.

Tabla 4.1: Patrón de asignación de recursos.

Réplica	**Id nodo**
Source[1]	0
Source[2]	1
op1[1]	2
op1[2]	3
op1[3]	0
op2[1]	1
op2[2]	2
Sink[1]	3

Respecto a la representación de un nodo, existe un TaskManager encargado de ejecutar las tareas de flujo de datos, y de acuerdo con el marco conceptual, puede ejecutar una o más subtareas en threads separados, y para controlar cuántas tareas acepta un TaskManager, tiene los llamados taskslot (al menos uno). El modelo interno de un nodo se aprecia en Figura 4.3:

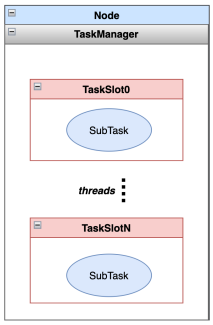

Figura 4.3: Modelo interno de un nodo de cómputo.

Con la representación interna de un nodo, el modelo del cluster con la aplicación desplegada en Figura 4.2, el resultado se visualiza en Figura 4.4:

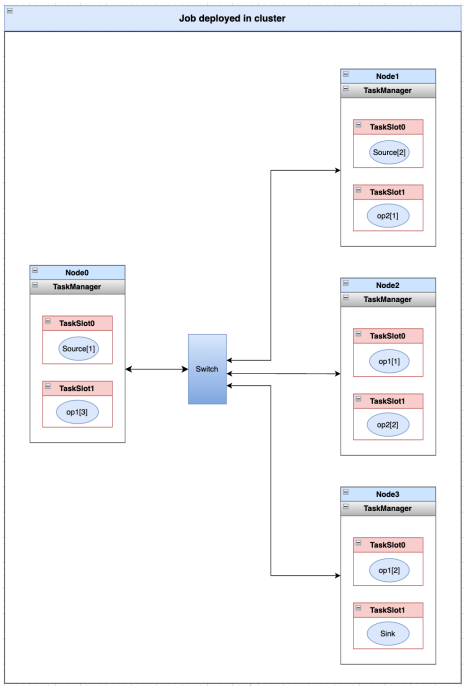

Figura 4.4: Modelo resultante de aplicación desplegada en un cluster.

Con el modelo del despliegue de la aplicación resultante, se observa que las réplicas pertenecen a un nodo y también a un taskslot, entonces la Tabla 4.1 termina siendo la Tabla 4.2:

Tabla 4.2: Patrón de asignación de recursos final.

Réplica	Id nodo	Id slot
Source[1]	0	0
Source[2]	1	0
op1[1]	2	0
op1[2]	3	0
op1[3]	0	1
op2[1]	1	1
op2[2]	2	1
Sink[1]	3	1

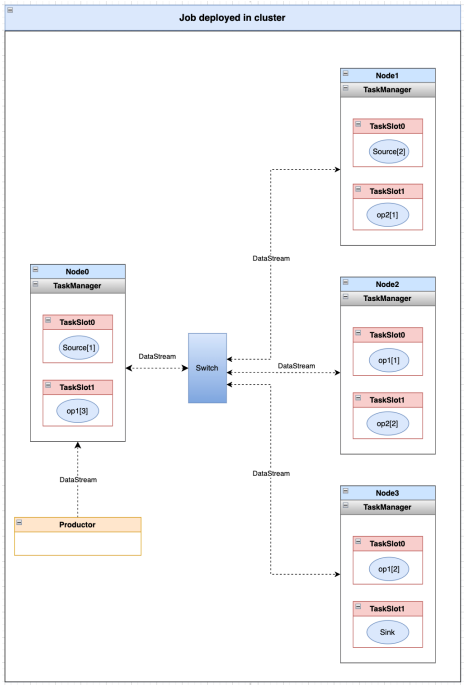

Figura 4.5: Modelo resultante de aplicación ejecutándose en un cluster.

Ya estando desplegada la aplicación, se procede a la ejecución, representada en Figura 4.5. Para ello y de acuerdo al marco conceptual, JobManager tiene otro componente llamado JobMaster, siendo responsable de gestionar la ejecución de un único DataFlow graph o grafo lógico creado al momento que un cliente envía una aplicación. Entonces, gestionar la ejecución del grafo lógico, se puede determinar como primera instancia que aquella entidad necesita saber:

- El grafo lógico (ciertamente), ya que es un grafo dirigido en el que los nodos son operadores y las aristas definen relaciones de entrada/salida del flujo de datos entre operadores. El JobMaster necesita de aquel grafo dado que le indica hacia qué subtarea debe ingresar el flujo de datos de la salida de otra subtarea.
- La ubicación física de las subtareas, ya que las subtareas de operador están distribuidas en todas las máquinas.

Por lo tanto, el modelo de simulación debe considerar la entrada de una topología que vendría a ser el grafo lógico, y mediante el patrón de asignación de recursos es posible generar la ubicación de las subtareas de operador como se consigue en la Tabla 4.2. Con el alcance de alto nivel se consigue la noción de un modelo inicial para inicializar el sistema distribuyendo una aplicación, pero al observar la Figura 4.5, también se considera la cantidad de nodos del cluster y la cantidad de réplicas por cada operador. Por ende, hay que tener en cuenta los siguientes parámetros para el modelo:

- Topología (definir enlace entre un operador y otro).
- Número de nodos y núcleos de manera homogénea como primer alcance, para ejecutar hilos (*threads*) representados por task slot.
- La cantidad de task slot vendría a ser la cantidad de réplicas de cada operador como primera instancia.
- Los operadores: nombre, réplica y función de distribución junto a sus argumentos para generar tiempos de operación.
- Tasas de arribo e instante de tiempo para el productor de flujo de datos.

Se determina entonces, que el simulador debe considerar un requisito funcional parametrizable, esto es mediante lectura de archivos de configuración para instanciar todas las estructuras que el programa requiere, luego escribir los registros generados de la simulación y desplegar métricas, aquello comprendido en el diagrama de Figura 4.6.

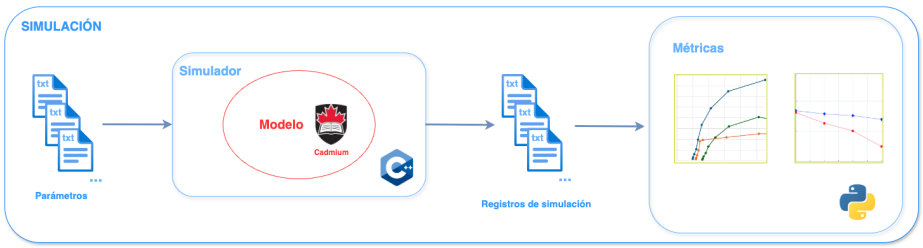

Figura 4.6: Diagrama de flujo de datos.

4.2. Diseño Lógico

Teniendo el alcance de un modelo inicial como se muestra en Figura 4.5, se consigue definir el diseño del modelo DEVS como diseño lógico del sistema. También se presenta el diagrama de despliegue, diagrama clases y diagrama de flujo.

4.2.1. Diagrama modelo DEVS

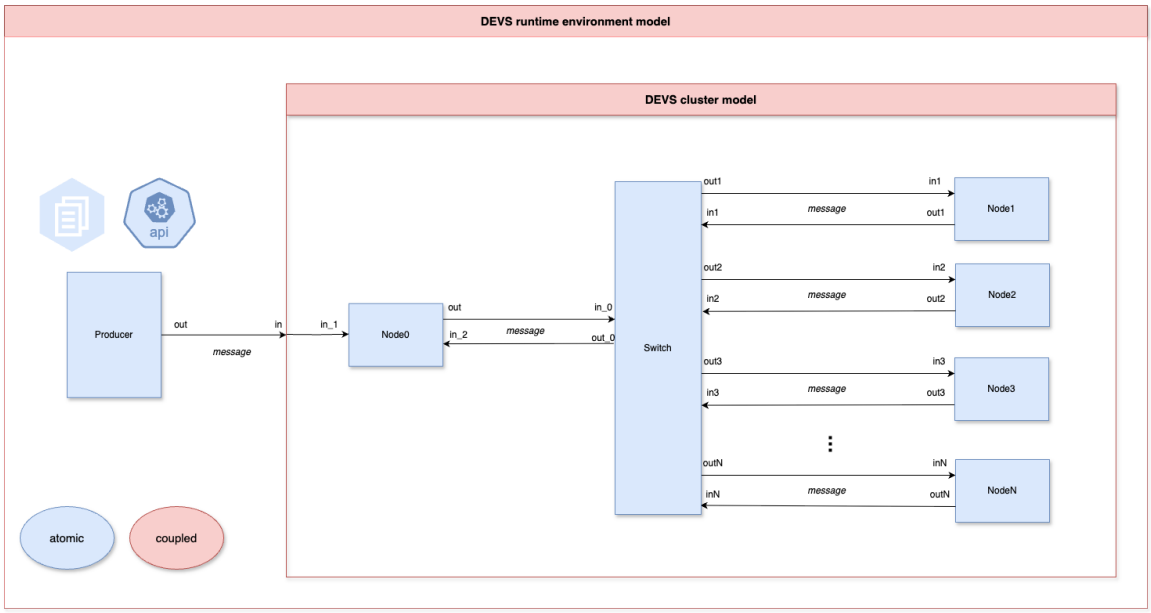

Figura 4.7: Modelo DEVS del sistema.

El modelo DEVS visto en Figura 4.7 comprende el ambiente de ejecución de procesamiento distribuido de flujo de datos. Aquel modelo superior está compuesto por un modelo atómico que se encarga de producir mensajes (tuplas) representando una API o fuente de datos externa, tales mensajes siendo consumidos por un segundo modelo acoplado representando un cluster. El modelo cluster esta compuesto por modelos atómicos siendo los nodos los cuales son conectados a través de puertos de entrada y salida hacia un modelo atómico switch, todo aquello representando la comunicación necesaria para la transferencia de mensajes entre operadores distribuidos en distintas máquinas.

4.2.2. Diagrama de despliegue

El diagrama de despliegue puede representar la topología del hardware junto con el sistema. En Figura 4.8, se presenta el diagrama de despliegue, que presenta el código fuente del modelo de simulación, el cual utiliza Cadmium principalmente y otras librerías estándar de templates de C++ mencionados en Sección 4.1.1. El simulador debe usar las entradas en formato txt con los parámetros del escenario a simular y a partir de aquella configuración se obtendrán las salidas en formato txt con los registros de la simulación.

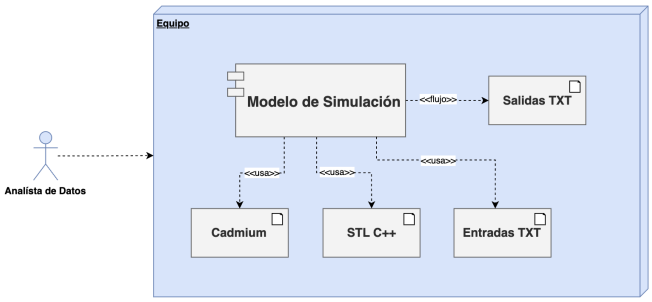

Figura 4.8: Diagrama de despliegue.

4.2.3. Diagrama de clases

En el proceso de transferencia de flujo de datos se involucran los principales modelos DEVS atómicos: switch, nodo y productor. Para efectuar tal caso, también es necesario contar con ciertos componentes como el JobManager al momento de iniciar la simulación, que es requerir el despliegue de una aplicación de acuerdo a la configuración entregada. Como se observa en el diagrama de clases de la Figura 4.9, se presentan varios conceptos representados por clases, empezando desde arriba hacia abajo donde se aprecia estructuras que almacenan la configuración inicial para luego desplegar la aplicación mediante el JobManager, que conceptualmente lo aplica mediante sus componentes entre el ResourceManager y JobMaster, observando métodos como: asignar recursos a un operador especifico, añadir localización de un operador para saber en que nodo y taskslot se asignó, obtener localización del operador, obtener operadores destino de acuerdo al grafo lógico, el proceso de ejecutar un operador en un taskslot mediante el TaskManager, conocer el estado de un taskslot, etc.

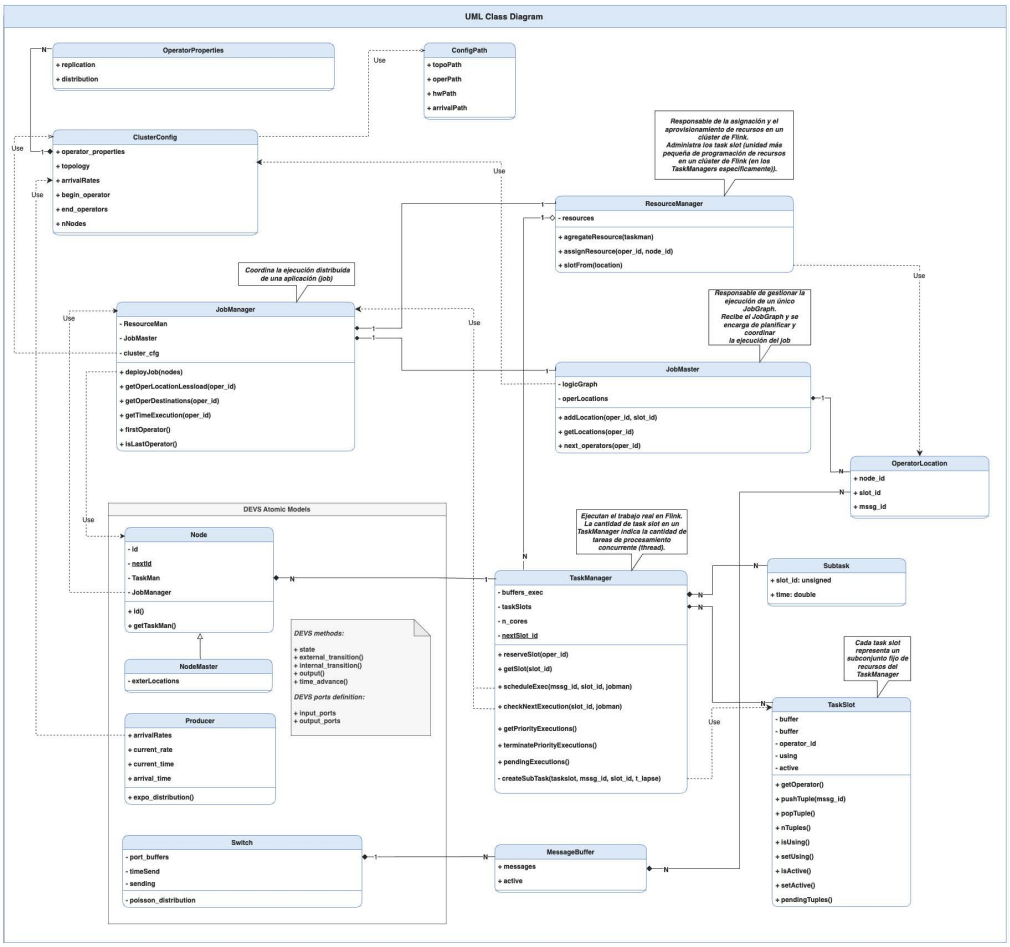

Figura 4.9: Diagrama de clases.

4.2.4. Diagrama de Flujo

En esta sección se presentan los diagramas de flujo de la lógica del proceso de simulación. En Figura 4.10 se presenta el flujo del despliegue de una aplicación o trabajo dado los parámetros de configuración, en Figura 4.11 se presenta el flujo del proceso de llegada y salida de un mensaje hacia un nodo, en Figura 4.12 muestra el flujo de la entrada y salida de mensajes hacia el switch y en Figura 4.13 muestra el flujo de generación de mensajes del productor. Los diagramas, a excepción del diagrama de Figura 4.10, corresponden a los modelos atómicos DEVS, los cuales dentro de su flujo dividen el comportamiento conforme a las funciones que el formalismo DEVS define (funciones de transición (interna, externa), función de salida y función de avance de tiempo). Para el modelo productor, solo requiere de la función de salida y transición interna, ya que solo procesa y genera mensajes, no recibe eventos externos.

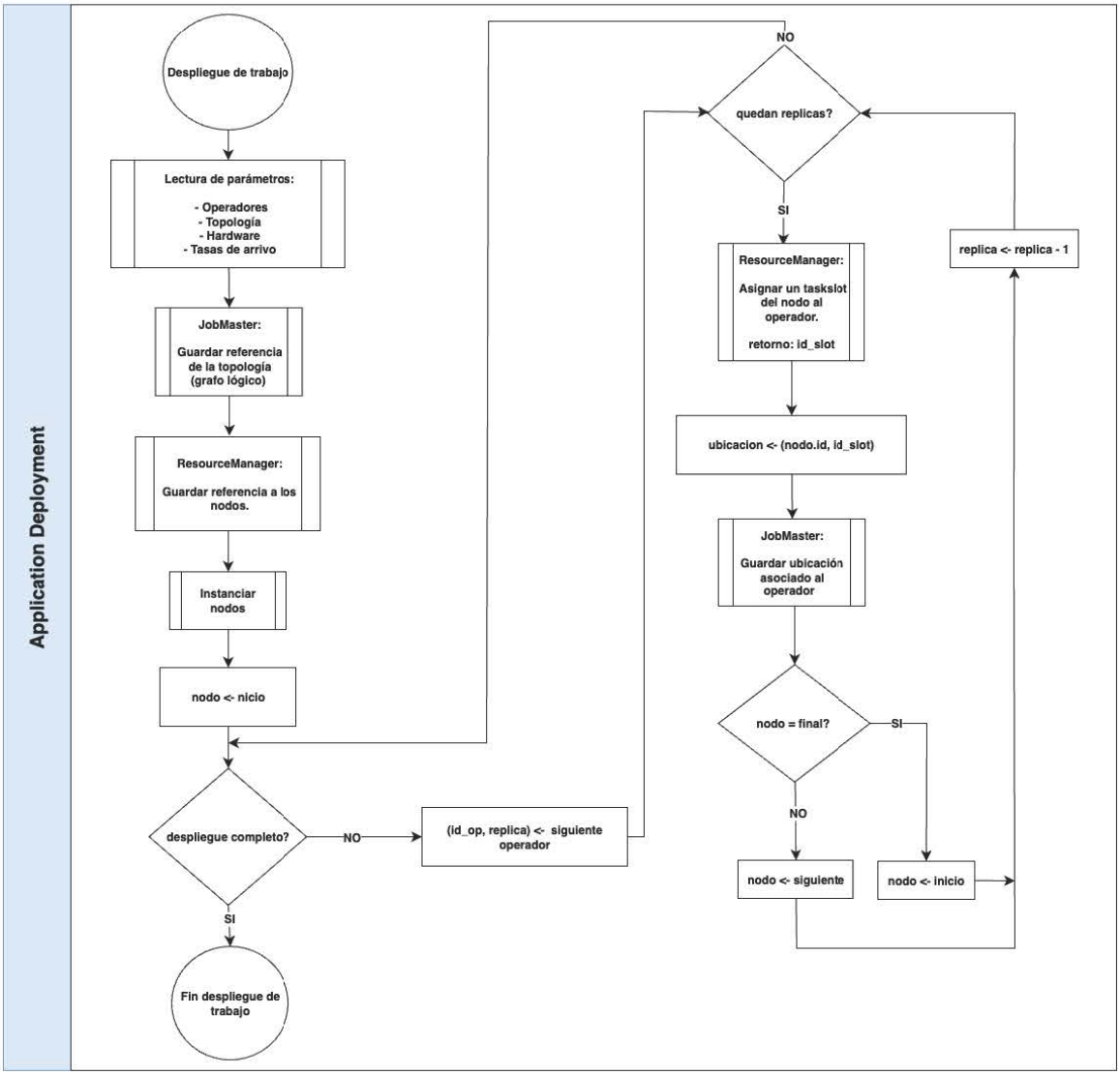

Figura 4.10: Diagrama de flujo proceso de despliegue de trabajo.

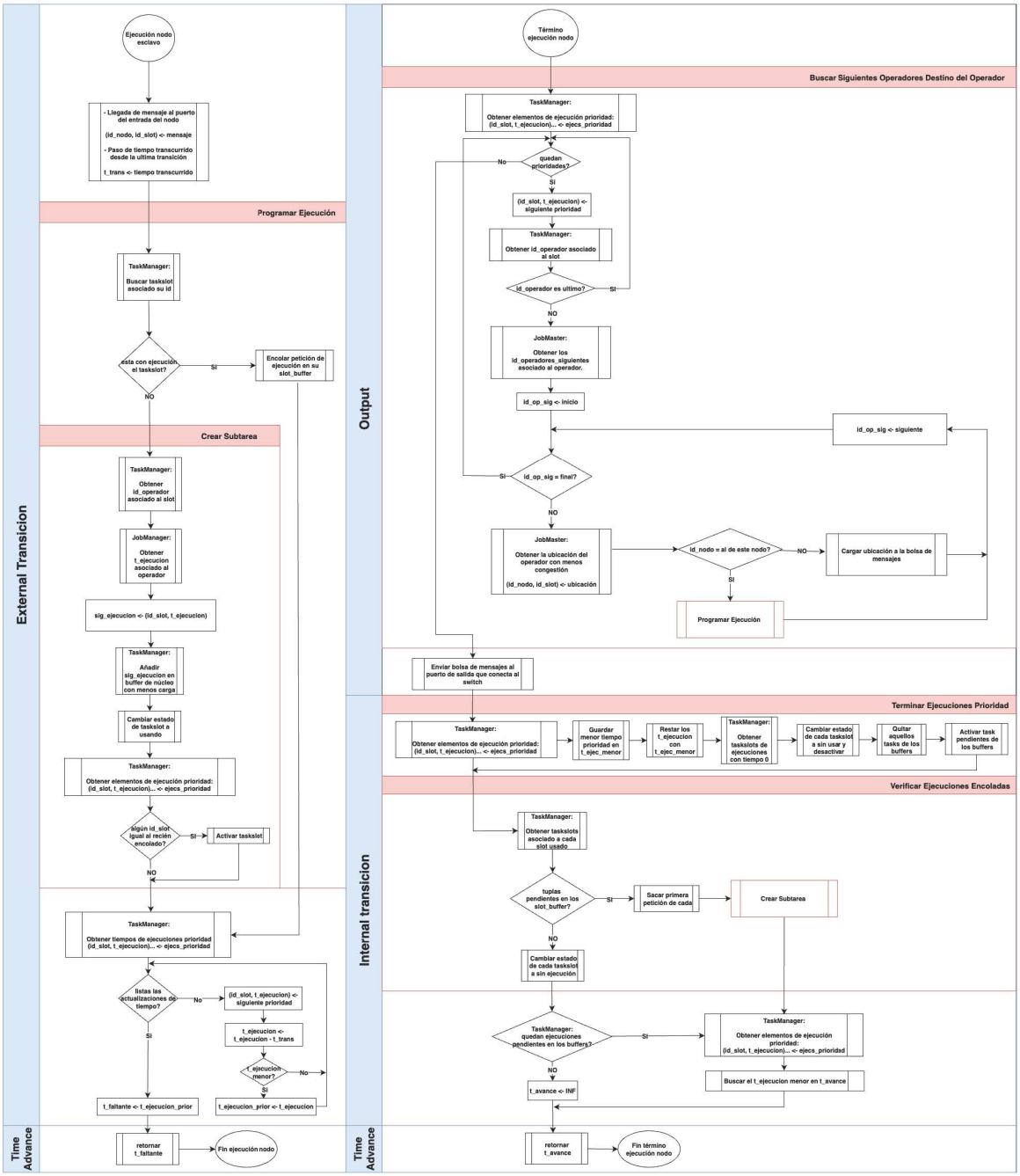

Figura 4.11: Diagrama de flujo proceso de llegada y salida de un mensaje hacia un nodo.

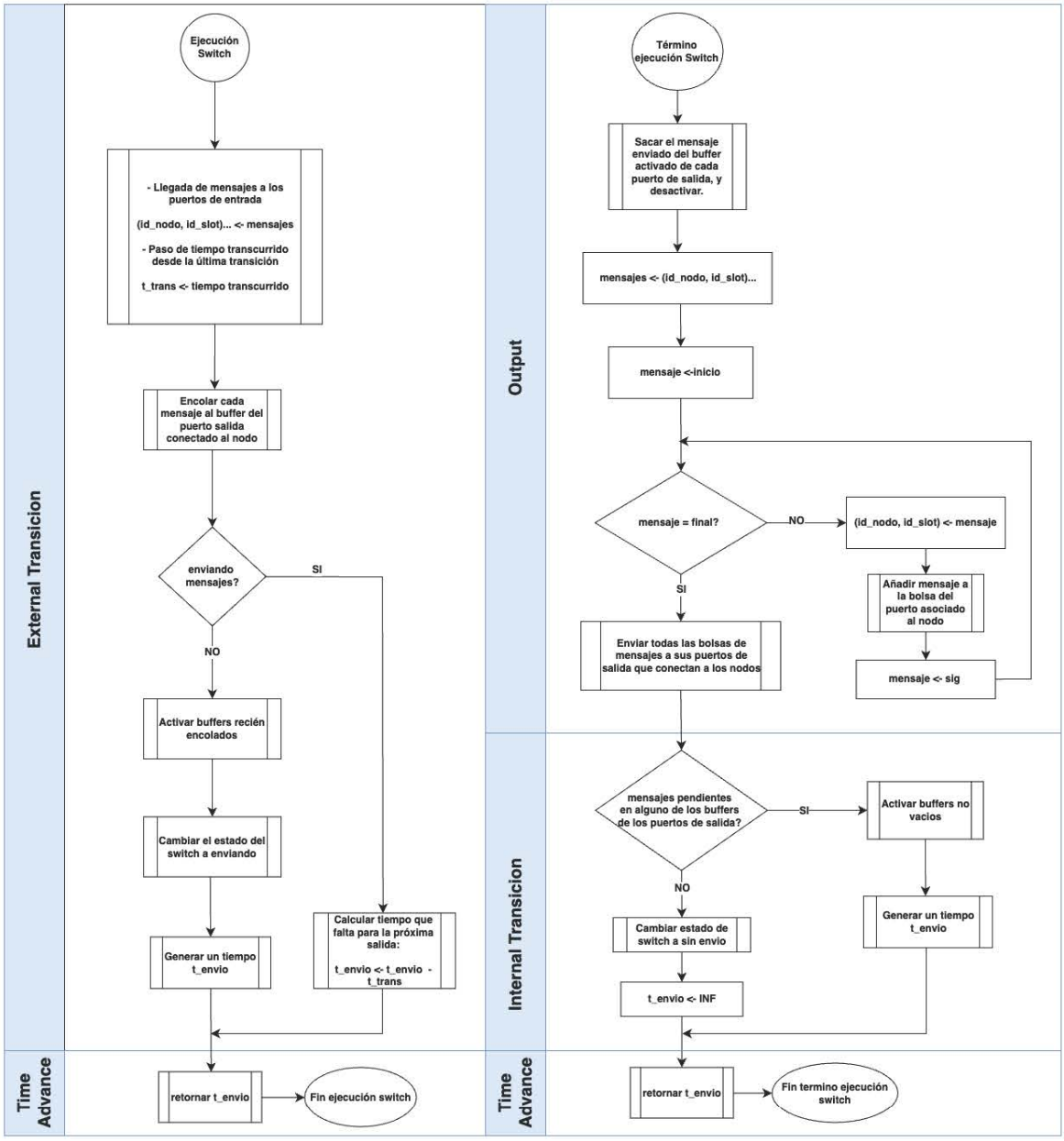

Figura 4.12: Diagrama de flujo proceso de llegada y salida de mensajes del switch.

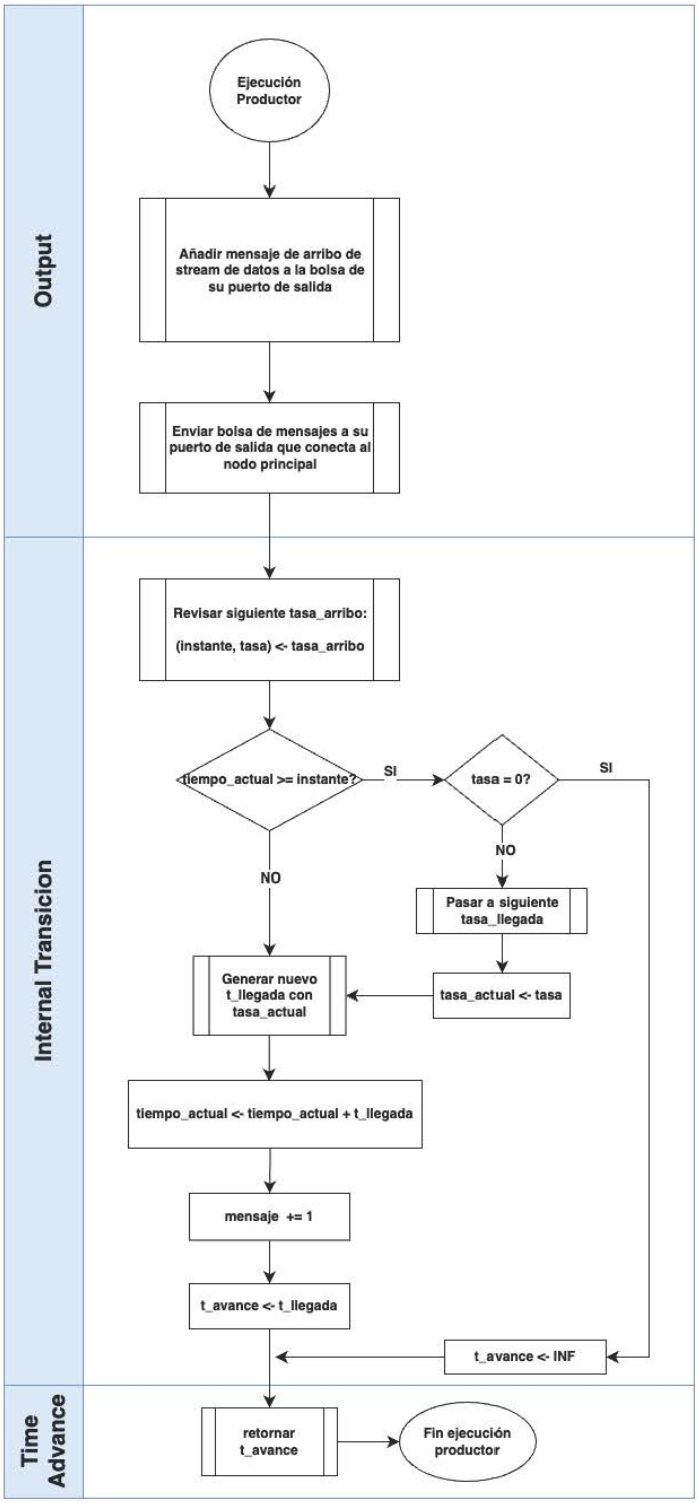

Figura 4.13: Diagrama de flujo proceso de generación de mensajes/eventos del productor.

4.3. Diseño de Datos

4.3.1. Parámetros y resultados

Como se aprecia en Figura 4.6 se hará uso de archivos para los parámetros de entrada y para los registros de la simulación. En Figura 4.14 se define la estructura de las configuraciones tanto para las tasas de llegada, la topología, operadores y hardware en sus archivos respectivos. Aquellos parámetros se consideran variables independientes de la simulación. Los registros de la simulación se estructuran como se visualiza en Figura 4.15, un archivo tanto para los registros de los nodos y para los operadores, ambos indicando el estado y la cantidad de requerimientos encolados en un instante de tiempo con una tasa de llegada determinado.

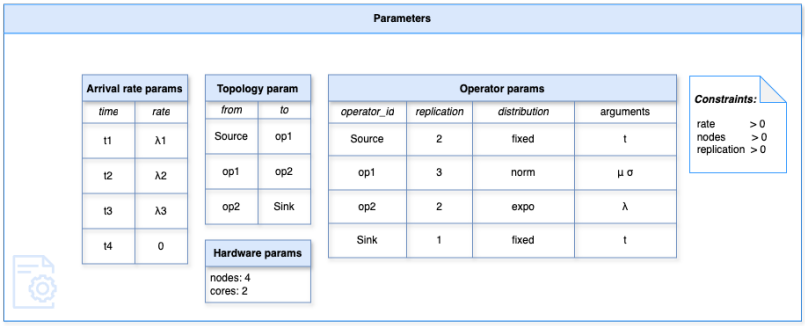

Figura 4.14: Estructura de archivos de configuración del modelo.

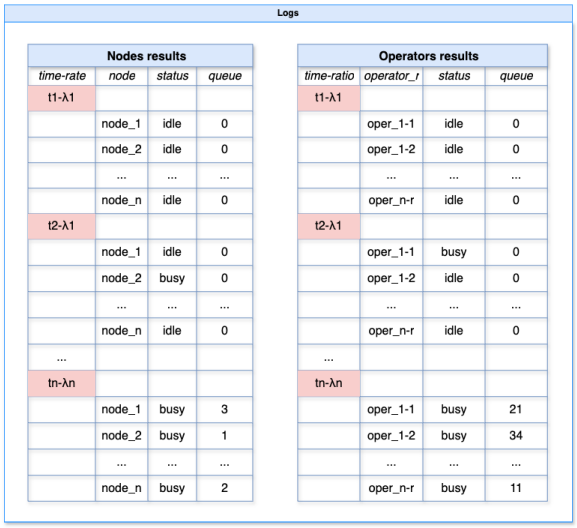

Figura 4.15: Registros de la simulación.

Con aquellos registros, se puede obtener métricas de interés como el nivel de utilización tanto para los nodos y los operadores, y el rendimiento (*Throughput*) en comparación con la tasa de arribo (*Arrival rate*) respecto al sistema completo. Las métricas se describen en la Tabla 4.3. La estructura de las métricas calculadas se muestra en Figura 4.16.

Tabla 4.3: Métricas de interés.

Métrica	**Fórmula**	**Descripción**
Utilización	$U = \frac{B}{T}$	El nivel de utilización es un porcentaje del tiempo que un recurso B estuvo activo bajo un período de observación T.
Arrival rate	$\lambda = \frac{A}{T}$	Es la cantidad de requerimientos A que llagan al recurso bajo un período de observación T.
Throughput	$X = \frac{C}{T}$	Es la cantidad de requerimientos C que el recurso ha completado bajo un período de observación T.

Metrics

Node Utilization Level

rate	*node_1*	*node_2*	...	*node_n*
λ1	0.35	0.05	...	0.032
λ2	0.3	0.096		0.321
...	...	...	...	...
λn	0.21	0.45	...	0.17

Operator Utilization Level

rate	*oper_1*	*oper_2*	...	*oper_n*
λ1	0.021	0.056	...	0.3
λ2	0.367	0.256		0.099
...	...	...	...	...
λn	0.11	0.078	...	0.31

Throughput vs Arrival Rate

rates	*requieriments*	*Finish time (s)*	Throughput (req/s)	*Time last arrival (s)*	Arrival rate (req/s)
[λ1, λ2, λ3, λ4]	277	0.007055	39262.93	0.006995	39599.71
[λ1, λ2, λ3, λ4]	306	0.007042	43453.56	0.006958	43978.15
...	...	...	...	...	...
[λ1, ..., λn]	329	0.007010	46932.95	0.006988	47080.71

Figura 4.16: Estructura de métricas calculadas.

4.4. Diseño de Pruebas

En esta sección se expondrá el diseño de las pruebas a realizar en el modelo de simulación para evaluar el correcto funcionamiento de todos los componentes de manera individual e íntegros.

4.4.1. Pruebas Unitarias

El formalismo DEVS permite la separación entre modelo y simulador, así como la estructura jerárquica y modular del formalismo, permitiendo realizar pruebas formales sobre las entidades en estudio. Esto incluye pruebas de componibilidad de subcomponentes y la verificación de los algoritmos de simulación, resultando en simuladores rigurosamente verificados [4]. Por lo tanto, es posible proceder con las pruebas unitarias, es decir, verificar el correcto comportamiento esperado de cada componente atómico. En Figura 4.17 expone el esquema de las pruebas unitarias, comenzando con el modelo productor, luego con el nodo y finalmente con el switch. Se observa que tanto el nodo como el switch se integran junto con el modelo productor, ya que es necesario recibir eventos para producir las transiciones externas. También, todos lo modelos de cada prueba unitarias están dentro de un componente acoplado superior, debido a que la librería Cadmium lo requiere dado los requisitos de implementación del motor [54]. Se observa también que en cada prueba unitaria hay un archivo, representando los mensajes de los puertos de salida correspondiente. Aquello es una utilidad que integra la librería Cadmium al momento de ejecutar pruebas mediante un componente 'logger' encargándose de registrar todos los eventos de salida de cada modelo atómico hacia un archivo.

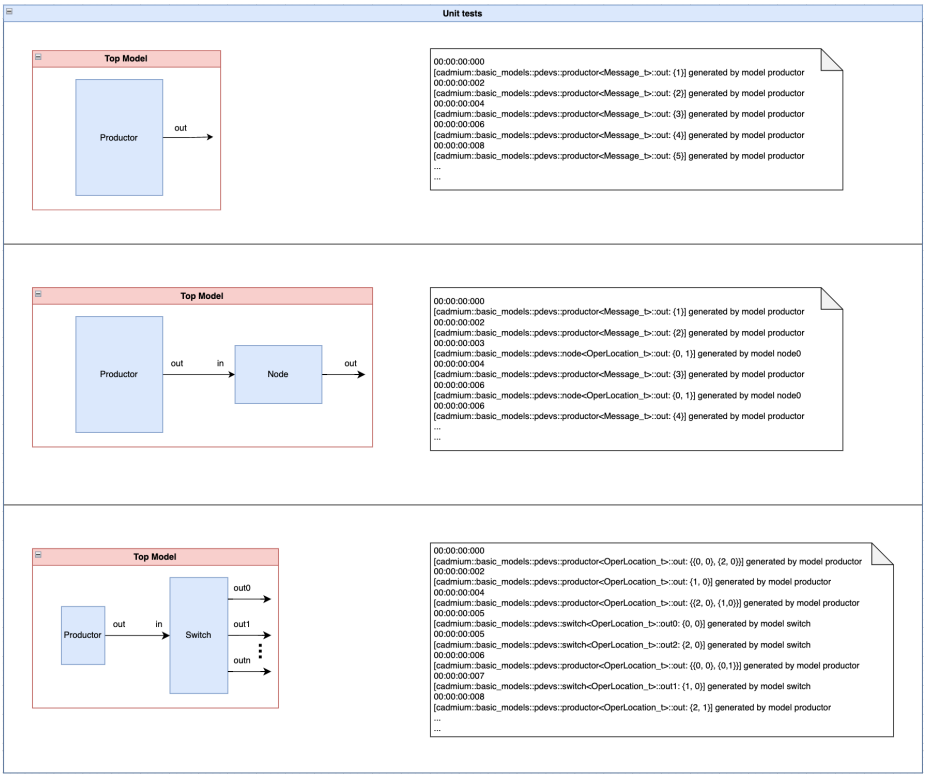

Figura 4.17: Esquema de pruebas unitarias.

4.4.2. Pruebas de Integración

Probado el comportamiento de cada componente atómico, se procede a probar de forma grupal, es decir, unir aquellas partes que establecen componentes mas amplios del software, con el propósito de probar la integración de diferentes módulos juntos e identificar los errores y problemas relacionados con ellos. La Figura 4.18, comprende el esquema de las pruebas de integración, iniciando con un prueba siendo similar a la segunda prueba unitaria de Figura 4.17, incluyendo ahora el modelo acoplado cluster que separa el conjunto de máquinas del productor: por una lado la fuente de datos y por otro el sistema distribuido que recibe y procesa flujos de datos. La segunda prueba de integración incluye el modelo atómico switch dentro del modelo cluster. Esta prueba da un paso a un modelo de simulación inicial de una plataforma de procesamiento de flujo de datos.

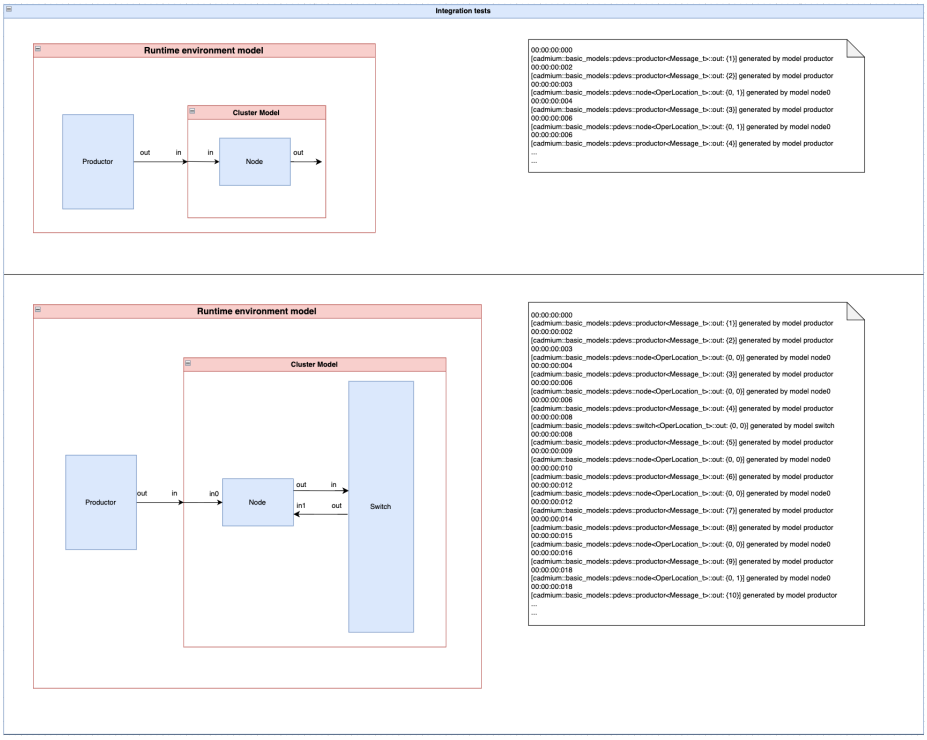

Figura 4.18: Esquema de pruebas de integración.

4.4.3. Pruebas de Sistema

Esta prueba contempla todo el sistema general del software, en este caso, se espera que el simulador se configure con mas de un nodo con el fin de probar su comportamiento considerado como un sistema de procesamiento distribuido.

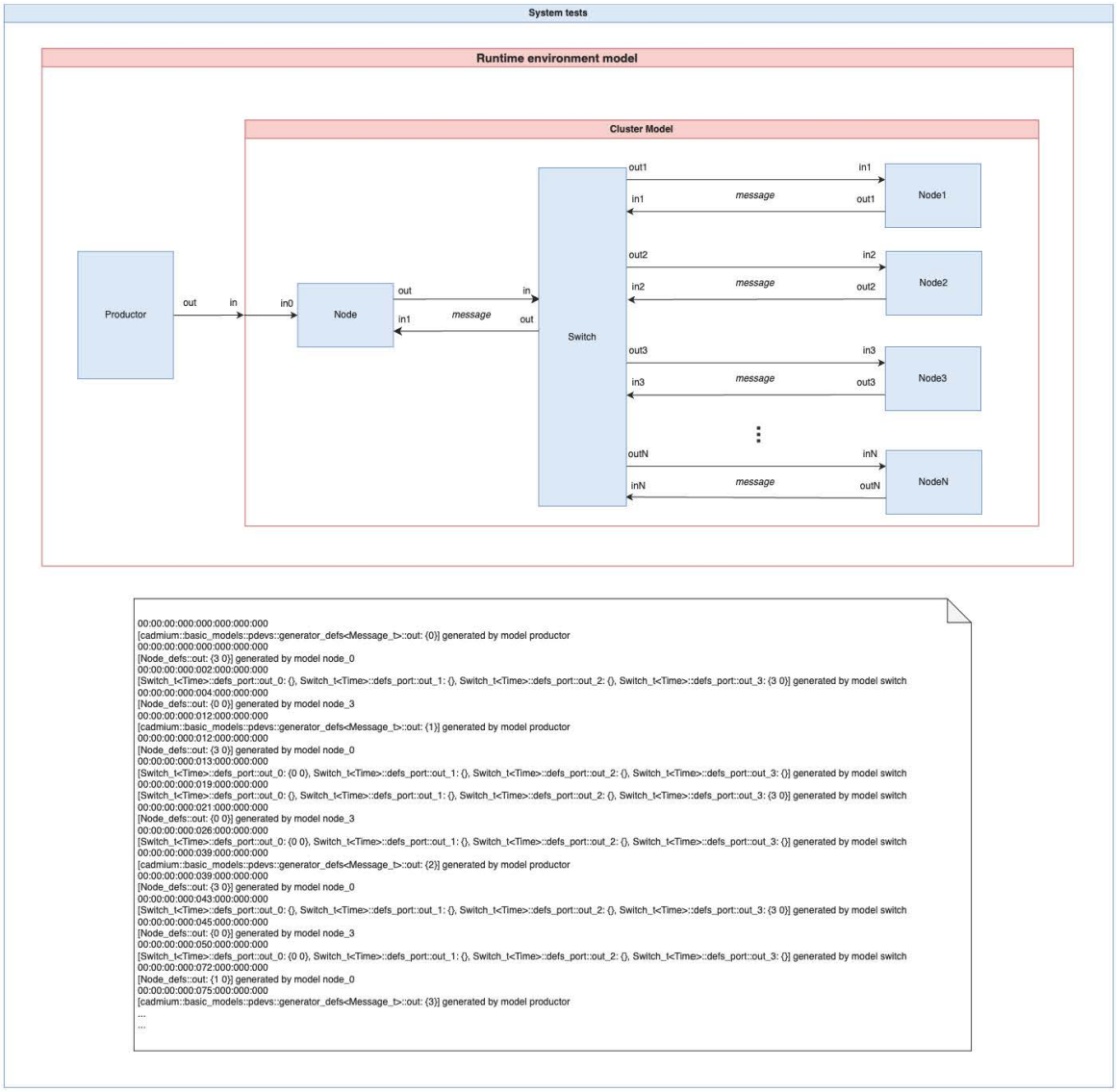

Figura 4.19: Esquema de pruebas de sistema.

4.4.4. Pruebas de Verificación

En Figura 4.20 describe los requisitos de comportamiento base para cada uno de los modelos atómicos para las pruebas de verificación.

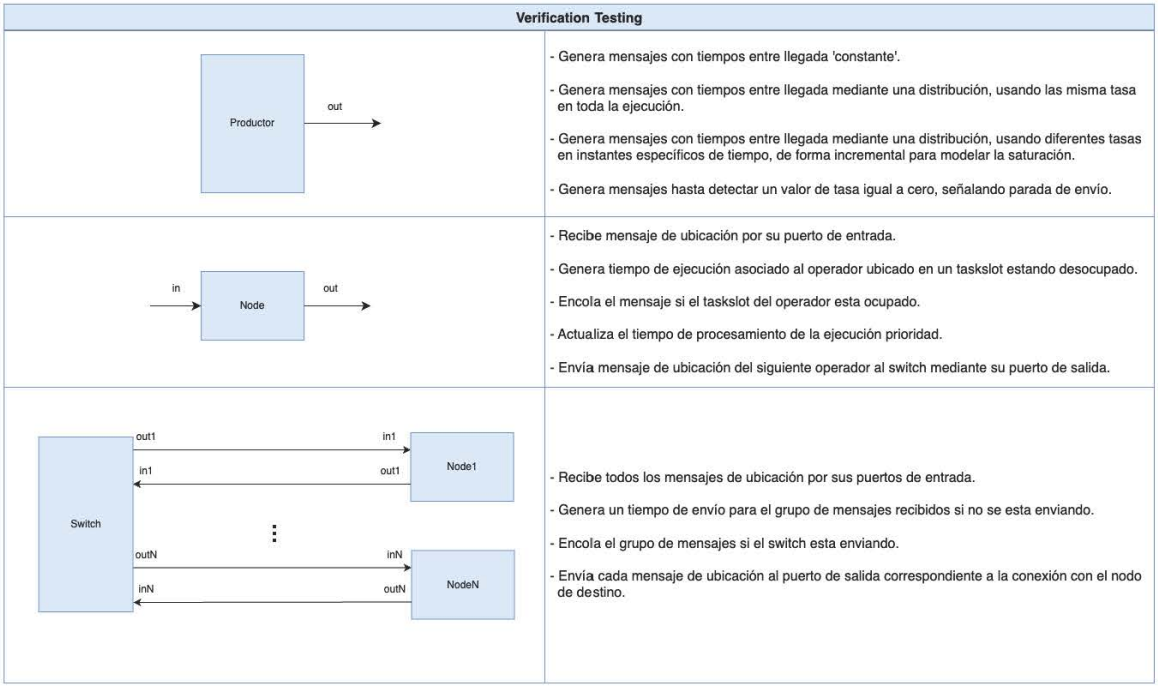

Figura 4.20: Esquema de pruebas de verificación.

4.4.5. Pruebas con Aceptación / Validación

El avance de esta etapa comprende reuniones con el maestro guía, y se considera la de mayor duración del proceso. Se deben tener en cuenta todos los encuentros previos, en los que se presentan y demuestran los cambios realizados sobre el desarrollo, permitiendo así un seguimiento continuo de los avances y validación.

Capítulo 5

Implementación

En este capítulo se presenta lo realizado en la fase de implementación. En la Sección 5.1 se da a conocer el hardware utilizado. En la Sección 5.2 se describe el software utilizado. El/los lenguaje/s de programación se justifican en Sección 5.3 y la estrategia de implementación en Sección 5.4.

5.1. Hardware utilizado

El equipo utilizado es un Apple chip M1 con las siguientes especificaciones:

- Arquitectura o conjunto de instrucciones ARM.
- CPU 8 núcleos (4 rendimiento, 4 eficiencia).
- GPU 7 núcleos.
- RAM 8 GB.

La Figura 5.1 presenta un esquema del sistema desde el punto de vista de hardware.

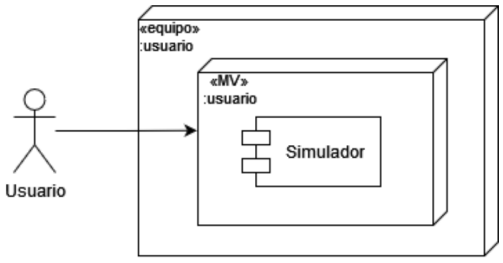

Figura 5.1: Esquema alto nivel hardware [5].

5.2. Software utilizado

Para la implementación del modelo DEVS se utilizaton las siguientes herramientas de software:

- **MacOS 15.4.1, Darwin Kernel Version 24.0.0**: Sistema operativo y Kernel que utiliza el equipo.
- **Zsh**: shell Z (Zsh) es un shell de Unix que se puede utilizar como un shell de inicio de sesión interactivo y como intérprete de comandos para scripts de shell. Zsh es un shell Bourne extendido con muchas mejoras, incluidas algunas características de Bash, ksh y tcsh. Aquella linea de comandos es la utilizada para la compilación del proyecto.
- **GCC/G++**: "GCC" es un término abreviado común para la Colección de compiladores GNU. Es el nombre más general del compilador y el nombre que se utiliza cuando el énfasis está puesto en la compilación de programas en C (la abreviatura antigua significaba "GNU C Compiler"). Cuando se hace referencia a la compilación de C++, es habitual llamar al compilador "G++"[58].
- **C++**: Lenguaje de programación de propósito general orientado a la programación de sistemas que: es una mejora de C, admite abstracción de datos, programación orientada a objetos y programación genérica [59]. Lenguaje principal utilizado para el desarrollo del simulador. El estandar 2020 de C++ es el utilizado para el desarrollo del modelo.
- **C++ STL**: La biblioteca de plantillas estándar proporciona un conjunto de componentes genéricos de C++ bien estructurados que funcionan juntos de manera perfecta [60]. Aquella biblioteca es utilizada en la mayoría de los componentes debido al alto nivel conceptual y a la separación entre interfaz e implementación, debido a la independencia entre contenedores, iteradores y algoritmos.
- **Cadmium**: Es una biblioteca de solo encabezado multiplataforma implementada en C++ para modelado y simulación de eventos discretos, basada en el formalismo DEVS. El simulador es implementado conforme a los requerimientos y las funcionalidades que entrega Cadmium mencionado en la guía de usuario [54].
- **Visual Studio Code**: Es un editor de código fuente desarrollado por Microsoft para Windows, Linux, macOS y Web. Se utilizo para la edición y desarrollo de la implementación del simulador.

- **Python**: Lenguaje de programación de alto nivel, interpretado y de propósito general, conocido por su sintaxis sencilla y legible. Es muy utilizado en desarrollo web, ciencia de datos, automatización, inteligencia artificial y más. La versión del interprete utilizado es Python 3.13.2.

- **Github**: Plataforma de desarrollo colaborativo basada en la web que permite almacenar, gestionar y controlar versiones de proyectos de software utilizando Git. Es ampliamente utilizada por desarrolladores para trabajar en equipo, revisar código, gestionar proyectos y publicar código abierto. También ofrece herramientas para la integración continua, documentación y seguimiento de errores.

En Figura 5.2 se presenta el flujo de alto nivel entre los componentes de software.

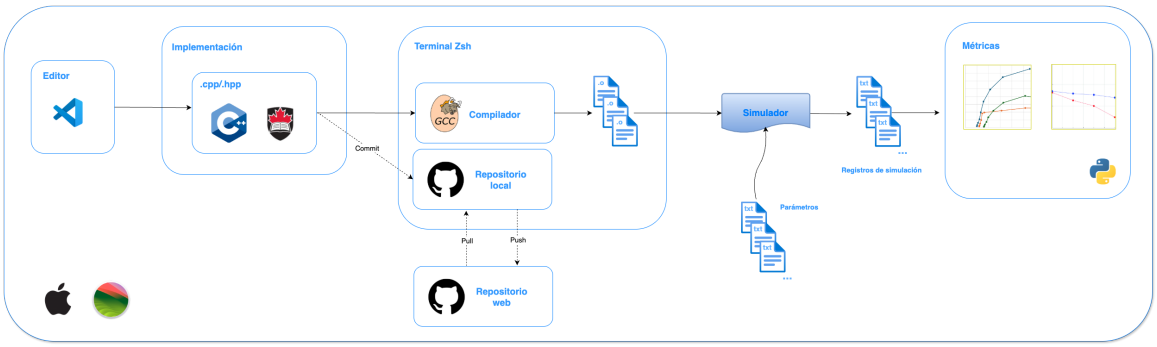

Figura 5.2: Esquema alto nivel de componentes de Software.

5.3. Lenguajes de programación

En la Sección 5.1 se mencionó dos lenguajes: C++ y Python. C++ es un lenguaje versátil, rico en tipado de datos y orientado al desarrollo de medio y bajo nivel con búsqueda de eficiencia en cuanto a tiempo de ejecución de los programas. Es un lenguaje conveniente para el desarrollo de simulaciones y aún más con modelos de gran escala. Es por ello, que el motor Cadmium fue implementado con C++ y los usuarios que desarrollan los modelos deben utilizarlo por compatibilidad con el motor y ediciencia. Por otro lado, Python es un interprete orientado a la aplicación de algoritmos y estructuras para la agilización del desarrollo de sofware. Integra una gran cantidad de funciones a disposición de propósito general que el usuario quiere aplicar. Bajo este aspecto, la utilización de este lenguaje se concentra en procesar los registros del simulador para el cálculo de métricas y visualización mediante gráficos que dispone Python en sus modulos.
En resumen, el desarrollo de código eficiente de modelos de simulación lo concentra C++ y

el proceso de registros y despliegue de gráficos para la interpretación de métricas se aplica con Python. Se puede encontrar más detalles sobre la implementación en el repositorio público DEVS-MODELS GitHub [61].

5.4. Estrategia de implementación

La estrategia comprende un trabajo con metodología incremental, esto es partiendo con la representación mental del sistema hasta la obtención de productos respecto a las pruebas unitarias y de integración.

5.4.1. Descripción de las etapas del proceso

Los siguientes puntos comprende las etapas del proceso incremental:

- Partir con una representación mental de un sistema de procesamiento de flujo de datos genérico, como se muestra en Figura 4.1.
- Representar los subcomponentes internos del sistema respecto a los conceptos de los componentes principales de la plataforma Flink, diseñado en Figura 4.5.
- Pasar la representación del sistema a un modelo conforme a la especificación DEVS, es decir, seleccionar el conjunto de componentes para los modelos atómicos y los componentes para modelos acoplados, como se muestra en Figura 4.7.
- Construir el modelo de datos de entrada a partir de los aspectos configurables, como la cantidad de nodos, propiedad de operadores, distribuciones, etc (Figura 4.14).
- Construir el modelo de datos principal conforme a la inicialización del sistema y su utilidad clave en el proceso del paso simulado de flujo entre operadores. (Tabla 4.2).
- Construir el diseño lógico del proceso formal DEVS, es decir la transición interna, externa, salida y avance de tiempo de cada modelo atómico (Subsección 4.2.4).
- Implementar las pruebas unitarias (Subsección 4.4.1), de integración (Subsección 4.4.2) y de sistema (Subsección 4.4.3).
- Verificar las pruebas conforme a los estados de los modelos mediante los registros de estado.
- Validar las pruebas conforme a las pruebas de aceptación, mediante los registros de mensajes y comportamiento del sistema a partir de las métricas.

5.4.2. Patrones de implementación

La implementación del modelo esta definido conforme a los requerimientos de Cadmium detallados en la guía de usuario [54], lo cual se considera los patrones respecto a los requerimientos y arquitectura interna del motor y por otro los patrones de diseño que han surgido por el lado del usuario:

- **Respecto a Cadmium**
  - **Clase plantilla para modelos atómicos**: Cadmium define una estructura de clase plantilla para los modelos atómicos. El parámetro plantilla representa el tipo de dato asociado al tiempo, con el propósito de operar con un tipo que el usuario desea para representar el tiempo. La definición de la clase comprende las siguientes estructuras y métodos internos:
    - **Puertos**: Estructuras plantilla para puertos de entrada y salida. Por cada puerto, se define una estructura que hereda de un clase plantilla *'port'*, tal que el parámetro plantilla es el tipo de dato que el puerto usa para el mensaje en emisión o recepción.
    - **Mensaje**: Estructura con parámetro plantilla de los puertos. Cadmium requiere que la estructura defina sobrecarga de operadores de flujo de entrada y salida para el registro de mensajes.
    - **Métodos**: Funciones de transiciones interna y externa, función de salida, función de avance de tiempo y operador salida para registrar los estados del modelo.
    - **Estado**: Estructura con datos que representa el estado del modelo.
  - **El motor está implementado con metaprogramación mediante plantillas**: las interfaces de programación de la librería requiere parámetros de tipo al momento de construir el modelo del sistema, debido a que el motor especializa su código por cada tipo en tiempo de compilación (ver Figura 5.3), gracias a la implementación genérica que los autores aplicaron mediante plantillas. Por lo tanto, el motor se auto configura en base a la necesidad del usuario. La metaprogramación se satisface mediante los siguientes patrones identificados:
    - **CRTP (herencia estática)**: El patrón de plantillas curiosamente recurrente, es un patrón donde una clase base plantilla toma como parámetro a una clase derivada, lo cual significa que una clase hereda de una clase plantilla especializándose sobre si mismo. Permite implementar polimorfismo en tiempo de compilación, evitando llamadas virtuales y su overhead.
    - **Especialización total**: Define una versión específica y completa de una plantilla para un conjunto exacto de parámetros. Sirve para cambiar por

completo el comportamiento de la plantilla cuando los parámetros son exactamente los que se especializaron.

- **Especialización parcial**: Define una versión modificada de la plantilla base para un subconjunto de los parámetros o una condición más general que no necesita ser exacta. Sirve para personalizar el comportamiento de la plantilla según algunos de los parámetros, pero no todos. Permite más flexibilidad que la especialización total.
- **Composite**: el motor tiene definiciones de clase plantillas llamados 'simuladores' y 'coordinadores', los cuales son conceptualizados como 'engine<>' a través de CRTP, el cual un simulador se encarga de la ejecución de un modelo atómico y un coordinador de la gestión de un modelo acoplado. Según DEVS, un modelo acoplado puede contener modelos atómicos y acoplados, entonces lo coordinadores pueden coordinar coordinadores que a su vez pueden coordinar mas coordinadores hasta llegar a las hojas siendo simuladores, definiendo un patrón de composición.

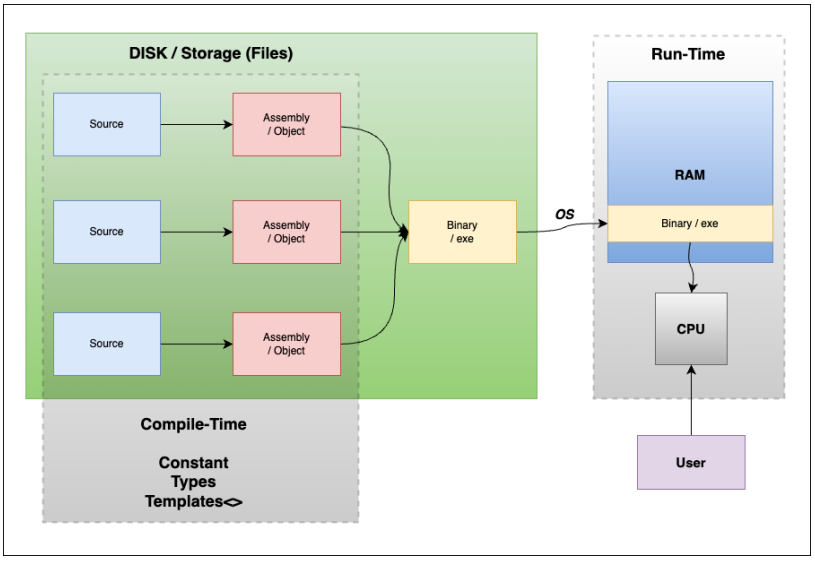

Figura 5.3: Trabajo en tiempo de compilación.

- **Respecto al usuario**
  - **Herencia**:
    - **Nodo maestro y esclavo es un nodo**: Existe cierta diferencia respecto a los eventos externos hacia el nodo master y hacia un nodo esclavo, ya que el maestro comprueba eventos externos de un productor o del switch, y un

nodo esclavo solo eventos externos del switch, por ende la implementación de las funciones de transición debe ser polimórfica, como se aprecia el diagrama izquierdo en Figura 5.4.

- **Modelo productor es un generador**: Cadmium provee un modelo básico que solo produce salidas, un generador, siendo definida como una clase con interfaz pura el cual se debe heredar para definir el tipo de mensaje de envió y la distribución entre envíos. En Figura 5.4 en la parte derecha representa la herencia que consiste en que el modelo productor es conceptualmente un generador de flujo de datos.

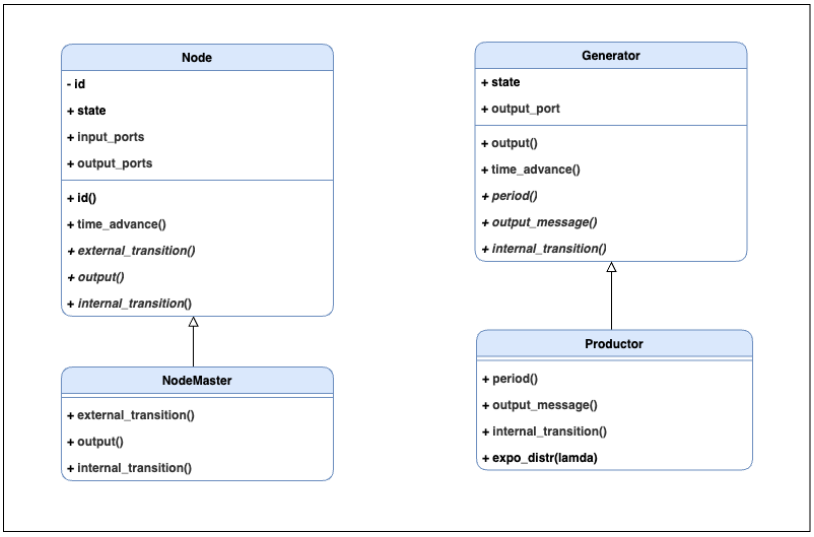

Figura 5.4: UML herencia de clases.

Capítulo 6

Pruebas

En este capítulo, se presenta el estado de la aplicación de las pruebas indicadas en la Sección 4.4.

6.1. Pruebas Unitarias

Esta sección muestra el cumplimento del comportamiento esperado de las pruebas unitarias para cada modelo atómico. En la Tabla 6.1, Tabla 6.2 y Tabla 6.3 describe las pruebas unitarias para el modelo productor, nodo y switch, respectivamente.

ID	Descripción	Estado
PUP1	Construcción: Agregación de tasas de arribo.	Correcto.
PUP2	Transición interna: Cambio a siguiente tasa de arribo cuando el instante de tiempo lo indique, y apaciguar el modelo cuando la indicación de tasa sea 0.	Correcto.
PUP3	Salida: envía el estado o número de mensaje actual del modelo.	Correcto.
PUP4	Avance de tiempo: envía el tiempo entre llegada de la tasa actual, o infinito si el modelo está apaciguado.	Correcto.

Tabla 6.1: Pruebas unitarias modelo Productor.

ID	Descripción	Estado
PUN1	Construcción: Agregación del JobManager.	Correcto.
PUN2	Transición externa: recibe mensaje, programa nueva ejecución, revisa ejecución pendiente, actualiza el tiempo de ejecución pendiente, actualiza el estado: procesando y marca del paso de tiempo del próximo evento	Correcto.
PUN3	Salida: buscar el/los siguiente/s operador/es, programar ejecución si el/los operador/es esta/n en el mismo nodo o enviar mensajes al switch si el/los operador/es están en otros nodos.	Correcto.
PUN4	Transición interna: terminar la ejecución actual, verificar ejecuciones pendientes, si hay pendientes actualizar marca de tiempo al tiempo de ejecución prioridad y activar el estado a procesando, de lo contrario no procesando.	Correcto.
PUN5	Avance de tiempo: enviar el tiempo de ejecución actual si esta procesando, de lo contrario infinito si el modelo está apaciguado.	Correcto.

Tabla 6.2: Pruebas unitarias modelo Nodo.

ID	Descripción	Estado
PUS1	Construcción: Generación de puertos de entrada y salida de acuerdo a la cantidad de nodos configurados.	Correcto.
PUS2	Transición externa: recibe mensajes, añade los mensajes a sus buffers respectivos, si no esta transmitiendo activa los buffers no activos y genera un nuevo tiempo de envió de la distribución y activa la transmisión, de lo contrario actualiza el tiempo de envío actual.	Correcto.
PUS3	Salida: envía la prioridad de mensaje de cada buffer activo y desactiva los buffers .	Correcto.
PUS4	Transición interna: activa lo buffers con mensajes de envio pendientes. Si no existen mensajes pendientes, desactiva la transmisión. De lo contrario genera un nuevo tiempo de envió de la distribución y activa la transmisión.	Correcto.
PUS5	Avance de tiempo: enviar el tiempo de envío actual si esta transmitiendo, de lo contrario infinito si el modelo está apaciguado.	Correcto.

Tabla 6.3: Pruebas unitarias modelo Switch.

6.2. Pruebas de Integración

Esta sección muestra el cumplimento del comportamiento esperado de las pruebas de integración para cada modelo acoplado. En la Tabla 6.4 y Tabla 6.5 describe las pruebas de integración para el modelo acoplado Productor-Nodo y Productor-Nodo-Switch, respectivamente.

ID	Descripción	Estado
PIPN1	Envío de mensajes del productor	Correcto.
PIPN2	Paso del mensaje del productor al modelo cluster mediante interfaz IC.	Correcto.
PIPN3	Paso del mensaje del productor al modelo nodo mediante interfaz EIC.	Correcto.
PIPN4	El nodo programa ejecuciones a la medida que llegan mensajes del productor.	Correcto.

Tabla 6.4: Pruebas de integración modelo Productor-Nodo.

ID	Descripción	Estado
PIPNS1	Envío de mensajes del productor	Correcto.
PIPNS2	Paso del mensaje del productor al modelo cluster mediante interfaz IC.	Correcto.
PIPNS3	Paso del mensaje del productor al modelo nodo mediante interfaz EIC.	Correcto.
PIPNS4	El nodo programa ejecuciones a la medida que llegan mensajes del productor, o envía mensajes al switch si se debe ejecutar en otros nodos.	Correcto.
PIPNS5	Paso del mensaje del nodo al modelo switch mediante interfaz IC.	Correcto.

Tabla 6.5: Pruebas de integración modelo Productor-Nodo-Switch.

6.3. Pruebas de Sistema

Esta sección muestra el cumplimento del comportamiento esperado de las pruebas del sistema para el modelo acoplado superior, descritas en la Tabla 6.6.

ID	Descripción	Estado
PSPNS1	Envío de mensajes del productor	Correcto.
PSPNS2	Paso del mensaje del productor al modelo cluster mediante interfaz IC.	Correcto.
PSPNS3	Paso del mensaje del productor al modelo nodo mediante interfaz EIC.	Correcto.
PSPNS4	El nodo maestro programa ejecuciones a la medida que llegan mensajes del productor, o envía mensajes al switch si se debe ejecutar en otros nodos.	Correcto.
PSPNS5	Paso del mensaje del nodo al modelo switch mediante interfaz IC.	Correcto.
PSPNS6	El switch programa un envió para los mensaje que le ha llegado.	Correcto.
PSPNS7	El switch envía los mensajes a los nodos mediante sus puertos respectivos.	Correcto.
PSPNS8	Paso de mensajes del switch a cada modelo nodo mediante las interfazes IC respectivas.	Correcto.
PSPNS9	Cada nodo programa ejecuciones a medida que llegan mensajes del switch y envía mensajes al switch si las siguientes ejecuciones son en otros nodos.	Correcto.

Tabla 6.6: Pruebas de integración modelo Productor-Nodo-Switch.

6.4. Pruebas de Verificación

En esta sección se presenta la verificación de los requisitos de comportamiento base de los modelos descritos en la Sección 4.4.4. En la Tabla 6.7, Tabla 6.8 y Tabla 6.9 describe la verificación del modelo productor, nodo y switch, respectivamente.

ID	Descripción	Estado
PVP1	Genera mensajes con tiempos entre llegada 'constante'.	Correcto.
PVP2	Genera mensajes con tiempos entre llegada mediante una distribución, usando las misma tasa en toda la ejecución.	Correcto.
PVP3	Genera mensajes con tiempos entre llegada mediante una distribución, usando diferentes tasas en instantes específicos de tiempo, de forma incremental para modelar la saturación.	Correcto.
PVP4	Genera mensajes hasta detectar un valor de tasa igual a cero, señalando parada de envío.	Correcto.

Tabla 6.7: Pruebas de verificación modelo Productor.

ID	Descripción	Estado
PVN1	Recibe mensaje de ubicación por su puerto de entrada.	Correcto.
PVN2	Genera tiempo de ejecución asociado al operador ubicado en un taskslot estando desocupado.	Correcto.
PVN3	Encola el mensaje si el taskslot del operador esta ocupado.	Correcto.
PVN4	Actualiza el tiempo de procesamiento de la ejecución prioridad.	Correcto.
PVN5	Envía mensaje de ubicación del siguiente operador al switch mediante su puerto de salida.	Correcto.

Tabla 6.8: Pruebas de verificación modelo Nodo.

Id	Descripción	Estado
PVS1	Recibe todos los mensajes de ubicación por sus puertos de entrada.	Correcto.
PVS2	Genera un tiempo de envío para el grupo de mensajes recibidos si no se esta enviando.	Correcto.
PVS3	Encola el grupo de mensajes si el switch esta enviando.	Correcto.
PVS4	Envía cada mensaje de ubicación al puerto de salida correspondiente a la conexión con el nodo de destino.	Correcto.

Tabla 6.9: Pruebas de verificación modelo Switch.

6.5. Pruebas de Aceptación / Validación

Durante las reuniones junto al maestro guía, se ha explicado y presentado el avance del modelo de procesamiento de flujo usando el caso de estudio Apache Flink. Hasta ahora, y en general se acepta el comportamiento deseado de la simulación con el nivel actual de afinidad del modelo.

Para esta evaluación, se simula una aplicación de prueba inicial presentada en la Tabla 6.10, junto a las propiedades de cada operador que establecen la distribución de tiempos de operación como se aprecia en Tabla 6.11. La aplicación seleccionada es un ejemplo inspirado en un patrón clásico de procesamiento de eventos por clave y ventana, muy común en casos de uso de análisis en tiempo real. No está basada en una aplicación de Flink específica del repositorio oficial, pero representa una estructura típica, realista y sencilla que se puede encontrar en muchos escenarios. Su topología está basada en patrones comunes descritos en la documentación oficial de Apache Flink [14], así como en casos representativos del Nexmark benchmark [62] y literatura académica [63] [64]. Esta estructura incluye operaciones básicas como transformación, filtrado, agrupamiento por clave y agregación por ventanas, típicas de sistemas de procesamiento de flujo de eventos.

From	To
source	map
map	filter
filter	keyby
keyby	window
window	sink

Tabla 6.10: Topología.

Id	Distribución	Argumentos
source	expo	0.02
map	norm	80 10
filter	fix	25
keyby	expo	0.01
window	norm	180 20
sink	expo	0.015

Tabla 6.11: Propiedad de operadores.

6.5.1. Primer alcance

Como fase inicial, se requirió comprobar una aceptación de acercamiento entre la Tasa de arribo (requerimientos entrantes por unidad de tiempo) y Throughput (requerimientos completados por unidad de tiempo) del sistema. Para ello, se inicia una prueba con una configuración de un solo nodo y operadores no replicados con tiempos de procesamiento mínimos fijos para modelar este primer alcance. El comportamiento se aprecia en

Figura 6.1, donde el eje horizontal son la ejecuciones con distintas tasas de llegada de requerimientos al sistema de manera incremental usando una distribución exponencial. Este incremento por ejecución implica una cantidad mayor de llegadas de requerimientos, como se aprecia en la curva naranja. Se comprueba que la curva azul representando el Throughput es muy parecido a la Tasa de arribo, lo cual se acepta debido a los tiempos mínimos fijos de procesamiento de cada operador.

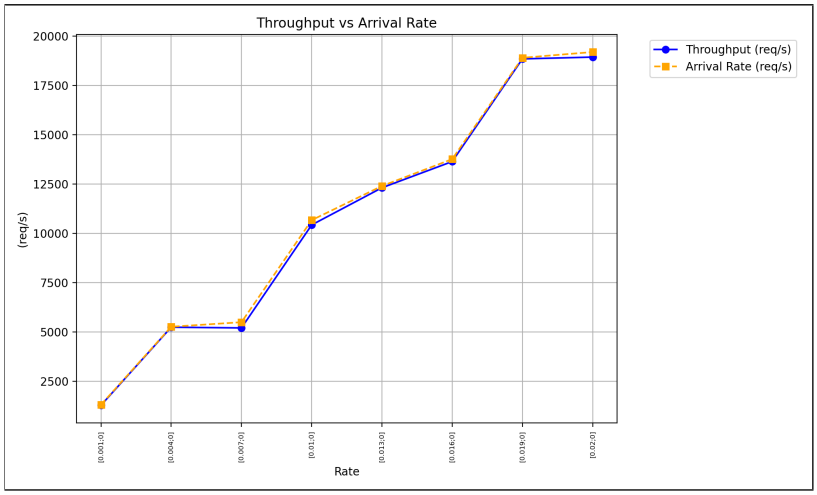

Figura 6.1: Throughput vs Tasa de arribo con 1 nodo y tiempos de procesamiento mínimos fijos en los operadores.

Si los tiempos de los operadores se configuran con distribuciones de generación de números aleatorios (en este caso el tiempo de operación), como se establece en la Tabla 6.11, se visualiza que el Throughput se aleja drásticamente de la curva de la Tasa de arribo en Figura 6.2.

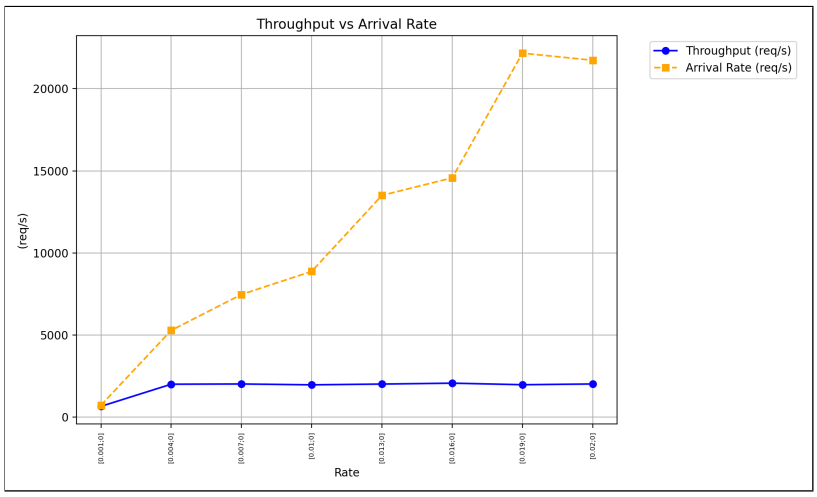

Figura 6.2: Throughput vs Tasa de arribo con 1 nodo y tiempos de procesamiento mediante distribución en los operadores.

La Figura 6.3 muestra la misma configuración de la prueba anterior pero cada operador replicado 8 veces. Se observa que el Throughput no aumenta debido a que la simulación solo considera un procesador por nodo. Sin embargo, en Figura 6.4 se aprecia un pequeño aumento de Throughput con una configuración de 4 nodos y en Figura 6.5 un aumento de rendimiento aceptable con una cantidad de 8 nodos.

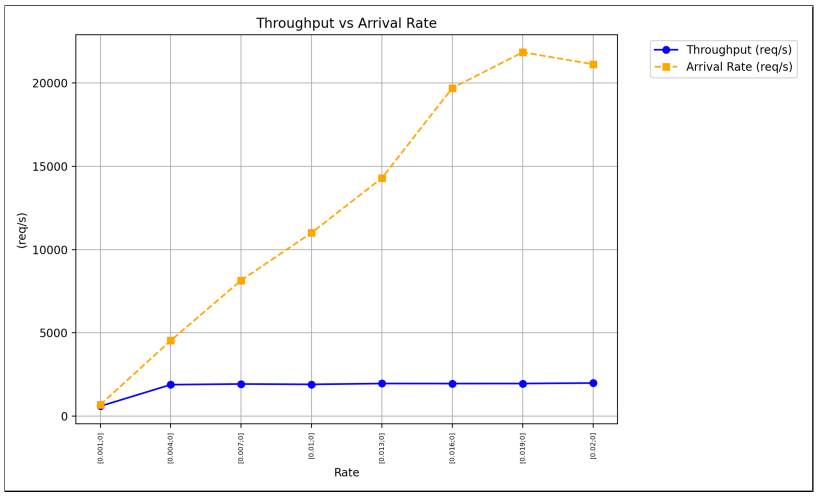

Figura 6.3: Throughput vs Tasa de arribo con 1 nodo y tiempos de procesamiento con distribución en los operadores replicados 8 veces.

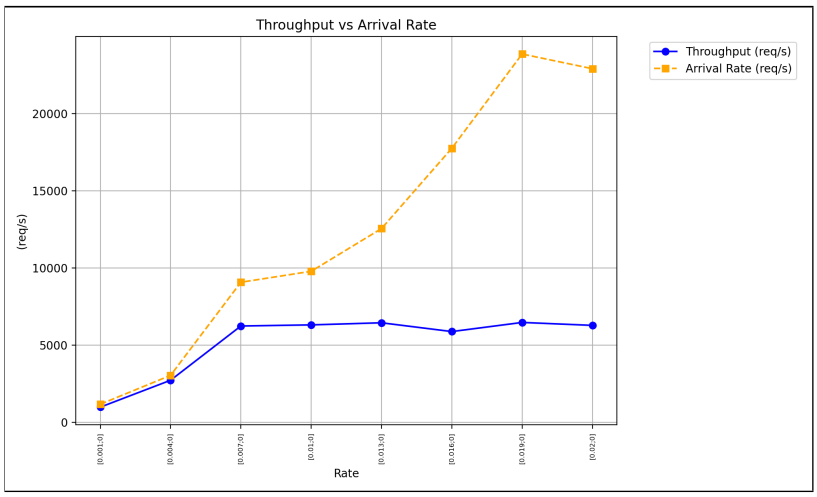

Figura 6.4: Throughput vs Tasa de arribo con 4 nodos y tiempos de procesamiento mediante distribución en los operadores replicados 8 veces.

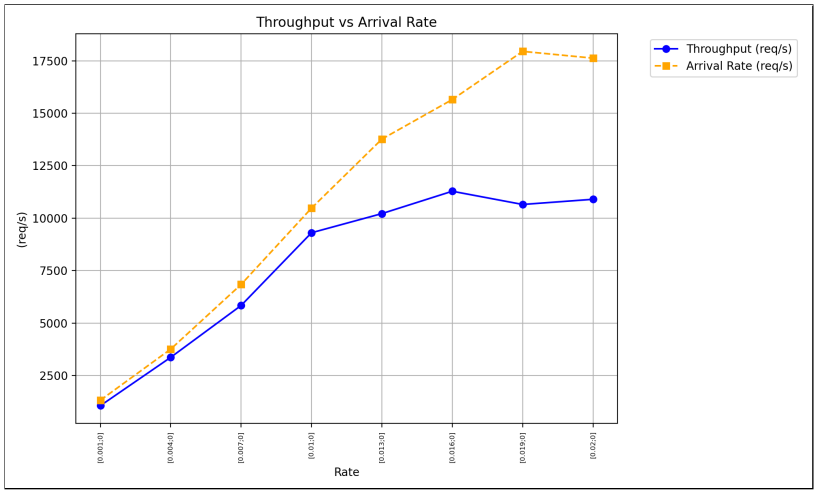

Figura 6.5: Throughput vs Tasa de arribo con 8 nodos y tiempos de procesamiento mediante distribución en los operadores replicados 8 veces.

Las primeras pruebas presentaron un funcionamiento esperado tanto en obtener un máximo Throughput cuando los operadores operan casi de manera instantánea, como cuando es necesario aumentar nodos de procesamiento para un mayor rendimiento. También al intentar aumentar el rendimiento replicando los operadores desplegados en una sola maquina no hubo mejora, debido a que la simulación no fue configurada con varios núcleos.

6.5.2. Escenarios y resultados

Se procedió a ejecutar los siguiente escenarios de simulación en correspondencia con la aplicación de la Tabla 6.10 y sus propiedades de operador en Tabla 6.11. A continuación, la Tabla 6.12 resume los resultados obtenidos de los escenarios simulados. Cada escenario consiste en cambios de configuración en cuanto a la cantidad de nodos, núcleos y réplicas de operadores, y 30 ejecuciones para el cálculo de métricas como el promedio y desviación estándar para valores operacionales siendo la Tasa de arribo y Throughput del sistema. En Figura 6.6 se visualiza como el Throughput cambia conforme a las configuraciones de cada escenario, acercándose a la tasa de arribo lo cual significa un rendimiento deseado a lograr.

Id scenario	**N° nodes**	**N° cores**	**Operator replicas**						**Arrival rate**		**Throughput**	
			so	**ma**	**fi**	**ke**	**wi**	**si**	**avg**	**std**	**avg**	**std**
1	1	1	1	1	1	1	1	1	11162	435	1973	19
2	6	1	1	1	1	1	1	1	11259	358	5163	44
3	6	1	1	5	1	5	5	1	11297	455	7078	175
4	6	1	5	5	5	5	5	5	11113	404	8215	173
5	6	2	5	5	5	5	5	5	11206	359	10865	344

Tabla 6.12: Resumen de resultados.

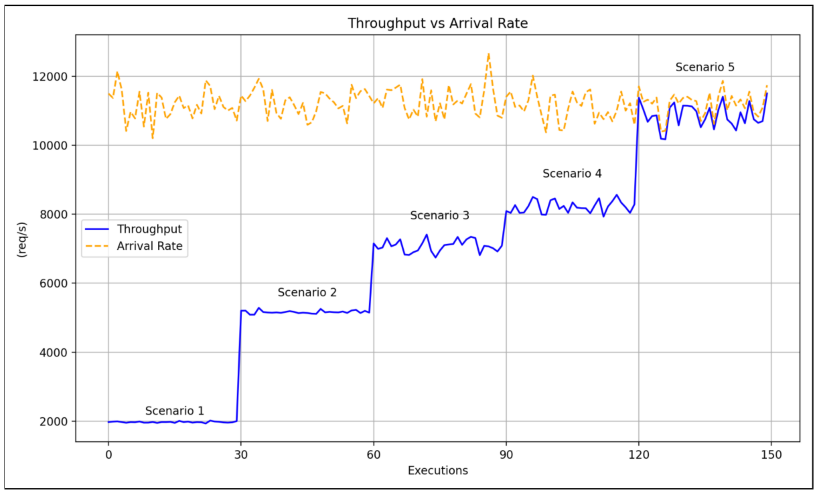

Figura 6.6: Throughput vs Tasa de arribo.

A continuación, se describen los escenarios analizando la métrica de utilización (relación entre el tiempo ocupado respecto a un tiempo de observación) de nodos y operadores junto con el Throughput. Un punto conceptual importante de la utilización es que indica cuánto se está aprovechando cada nodo u operadores durante el procesamiento simulado de flujo. En el caso que algún operador tenga una alta utilización, puede indicar que está operando muchos datos, lo cual puede ser bueno ya que está aprovechando el recurso de procesamiento, o malo dado que se convierte en un cuello de botella manteniendo ocupado el recurso de proceso disponible. Por lo tanto, este análisis permite entender dónde optimizar o replicar componentes para mejorar el rendimiento del sistema.

Las gráficas de utilización de cada escenario es la captura de alguna de las 30 ejecuciones. El eje horizontal son los cambios incrementales de las tasas de llegada de eventos al sistema que el modelo productor utiliza (la última tasa igual a cero indica fin de eventos entrantes y descarga de requerimientos pendientes en el sistema). El eje vertical es la tasa de utilización.

Escenario 1: Este escenario partió ejecutando la simulación con 1 nodo, 1 núcleo y operadores sin réplicas. La Figura 6.8 muestra que los niveles de utilización de los operadores están por debajo del 40 %, aunque un nivel más demandante por parte del operador 'windows', ya que es el que requiere más tiempo de operación dada su distribución establecida. De todas formas, la utilización en general es baja, y eso es debido a que hay un único nodo a cargo del procesamiento para los 6 operadores, observando su alta utilización en Figura 6.7. Con esta configuración, se aprecia que el rendimiento siendo el valor promedio del Throughout de 1973 req/s es demasiado alejado respecto al promedio de la Tasa de arribo de 11162 req/s.

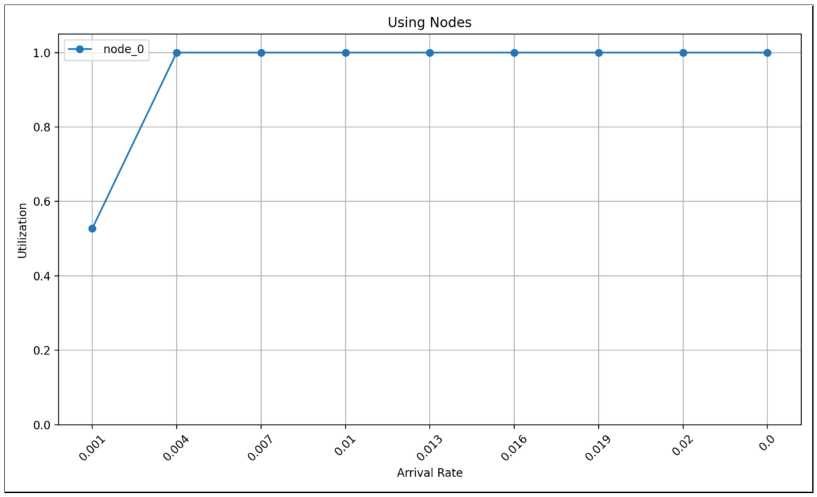

Figura 6.7: Utilización de un nodo con 1 núcleo.

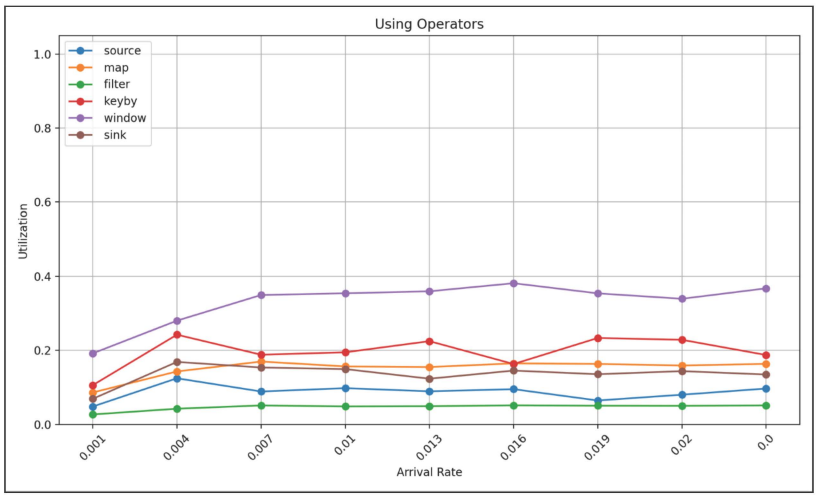

Figura 6.8: Utilización de operadores sin réplica.

Escenario 2: Para observar más la utilización, se procedió a una prueba ejecutada con 6 nodos. En Figura 6.10 se observa que la tasa de cada operador aumenta, coincidiendo con la tasa de los nodos apreciados en Figura 6.9, ya que cada operador fue asignado a cada máquina debido al balance de carga de la gestión de asignación del modelo. Por ende, cada nodo se encarga de ejecutar un operador, lo cual el tiempo ocupado respecto al tiempo observado en cada intervalo coincide en ambos. Con esta configuración se aprecia que el rendimiento aumenta a un Throughput promedio de 5163 req/s, sin embargo, sigue estando por debajo de la mitad de la Tasa de arribo promedio. Surge entonces la cuestión de saber si es necesario adquirir más recursos o generar más réplicas de operadores. Apreciando que algunos nodos no están siendo suficientemente utilizados durante altos niveles de carga de llegadas de eventos al sistema, indica que hay procesamiento disponible para la existencia de réplicas, y en este caso para los operadores *map*, *keyby* y *window* ya que son los más utilizados.

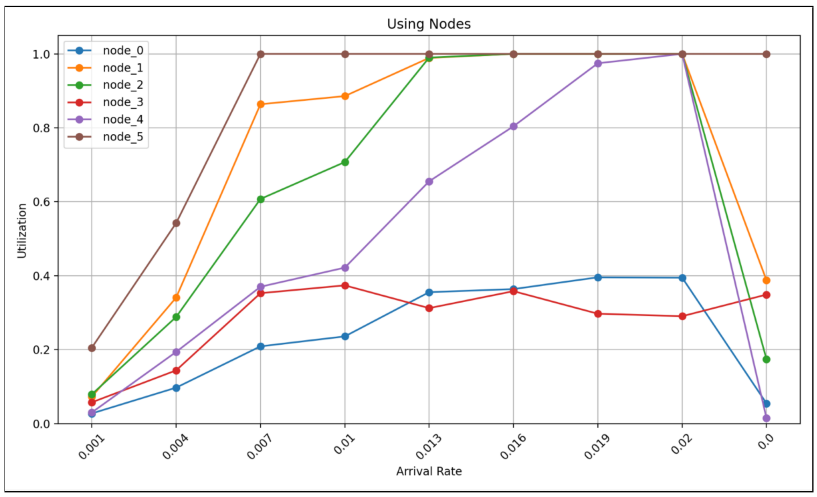

Figura 6.9: Utilización de 6 nodos con 1 núcleo c/u.

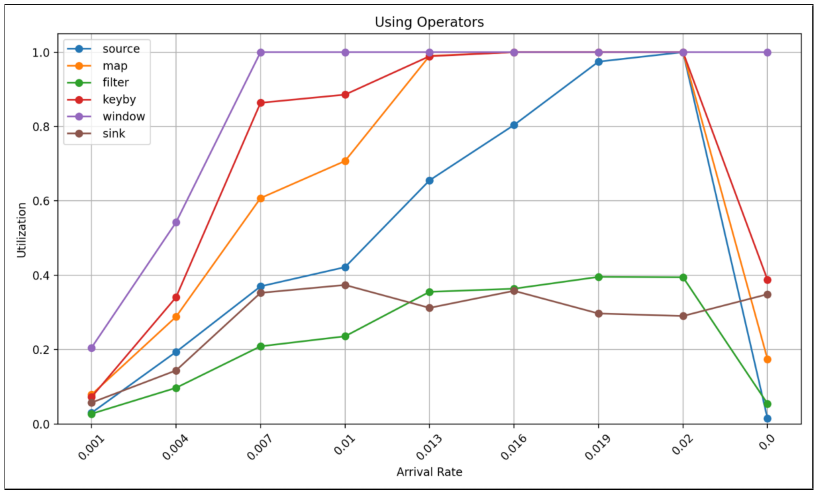

Figura 6.10: Utilización de operadores sin réplica.

Escenario 3: En este escenario se procedió a ejecutar el modelo con algunos operadores replicados mencionados en el escenario anterior, con un número de 5 réplicas manteniendo la cantidad de nodos y núcleos. En Figura 6.12 se aprecia que los operadores replicados ya no están saturados debido al nivel de paralelismo asignado, y esto a su vez repercutió en la disminución de utilización de otros operadores por ejemplo el 'source', ya que no es el único que se estuvo ejecutando en el nodo asignado, o también un aumento como lo fue en el operador 'sink' dada la cantidad de datos entrantes de la salida de los operadores 'window' replicados. No obstante, en efecto, se consiguió una ganancia del Throughput a

2000 req/s más aproximadamente y también como se visualiza en Figura 6.11, los nodos se encuentran más ocupados en comparación a niveles anteriores, pero aun así se aprecia que es posible exigir más el sistema replicando operadores, ya que algunos nodos no se encuentran en un estado de saturación significativo.

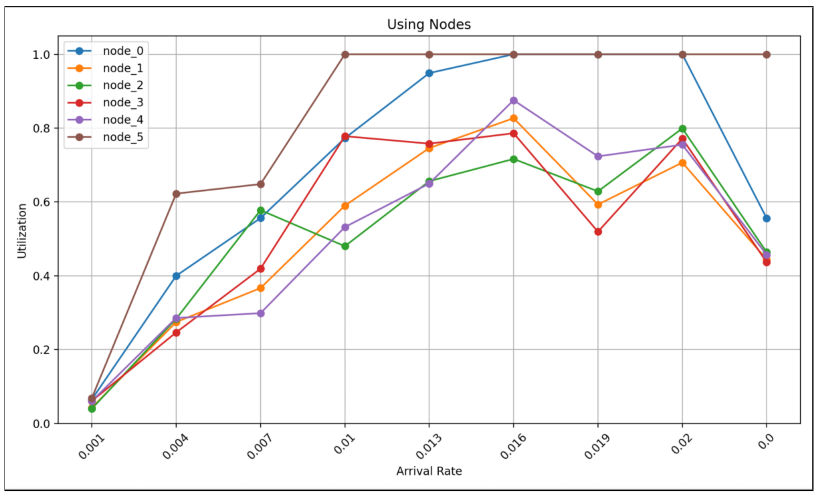

Figura 6.11: Utilización de 6 nodos con 1 núcleo c/u.

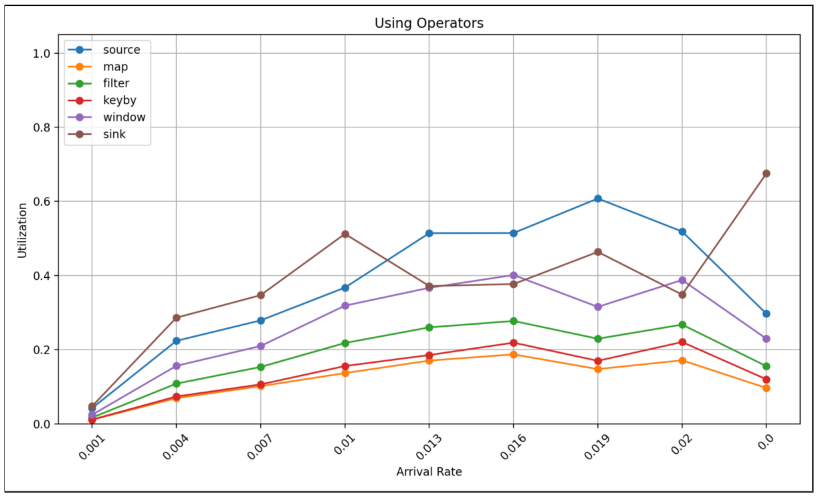

Figura 6.12: Utilización de operadores con replicación = 5 para *map*, *keyBy* y *windows*

Escenario 4: En este escenario se procedió a replicar los operadores restantes: 'source', 'filter' y 'sink'. Con un número de réplica igual a 5 en todos los operadores desplegados en 6 nodos con 1 núcleo c/u, en Figura 6.14 se observa los nuevos niveles de utilización

de operadores y se aprecia que el sistema se encuentra en un punto de saturación como lo muestra Figura 6.13. El Throughput efectuado fue de 8215 req/s completados en promedio, el cual se considera un prudente acercamiento a la tasa media de requerimientos entrantes al sistema de 11113 req/s. En esta instancia, se determina que los recursos que dispone el sistema se encuentran mayormente ocupados durante altas tasas de eventos entrantes, y respecto la utilización de los operadores, se observa que vuelven a ser similares a las obtenidas inicialmente sin replicación ejecutadas en una sola máquina (ver Figura 6.8), pero ahora estando en un caso de 6 máquinas con 5 replicas, consiguiendo una ganancia de rendimiento. Siguiendo el mismo procedimiento, ganar aún más rendimiento si es posible, procediendo a configurar el número de núcleos de los que puede disponer cada nodo.

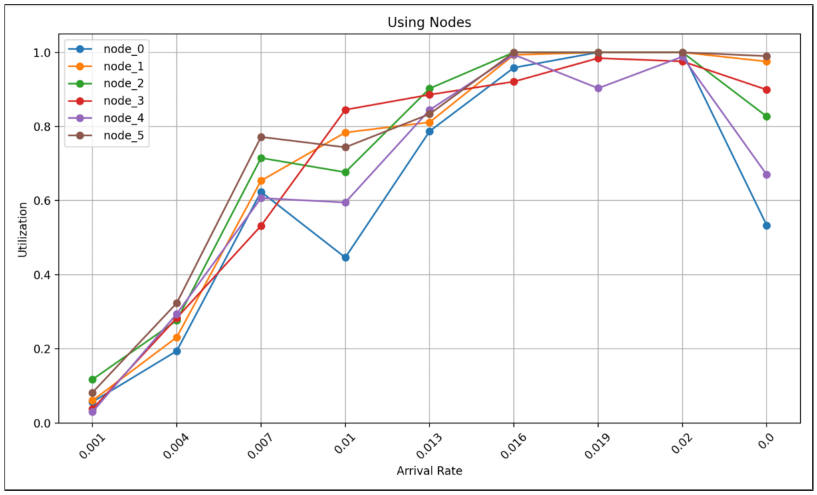

Figura 6.13: Utilización de 6 nodos con 1 núcleo c/u.

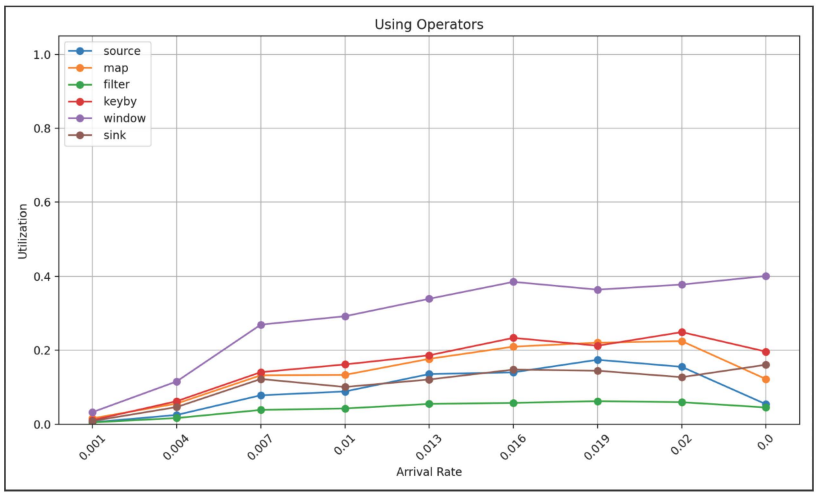

Figura 6.14: Utilización de operadores, todos con nivel de replicación = 5.

Escenario 5: Se procedió a ejecutar la simulación configurada con 2 núcleos en cada una de las 6 máquinas del sistema, manteniendo la cantidad de réplicas de operaciones. La utilización de operadores aumenta dado que existen más procesadores disponibles para las réplicas como se aprecia en Figura 6.16, y en Figura 6.15 se observa que la utilización en los nodos no incrementa tan apresuradamente en comparación con el escenario anterior en Figura 6.13, indicando que el sistema es más lento en saturarse. Con la adición de recursos en cuanto a núcleos por máquinas, se logra obtener un rendimiento promedio considerable de 10865 req/s completados, y otro punto importante es que la desviación estándar del Throughput es consistente (con diferencia mínima) respecto la desviación de la Tasa de arribo visto en la Tabla 3 de este escenario en contraste con las desviaciones de los escenarios anteriores.

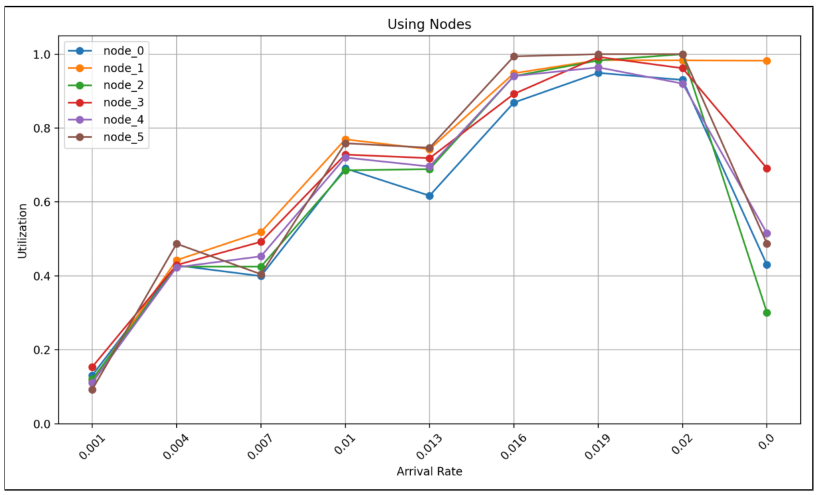

Figura 6.15: Utilización de 6 nodos con 2 núcleos c/u.

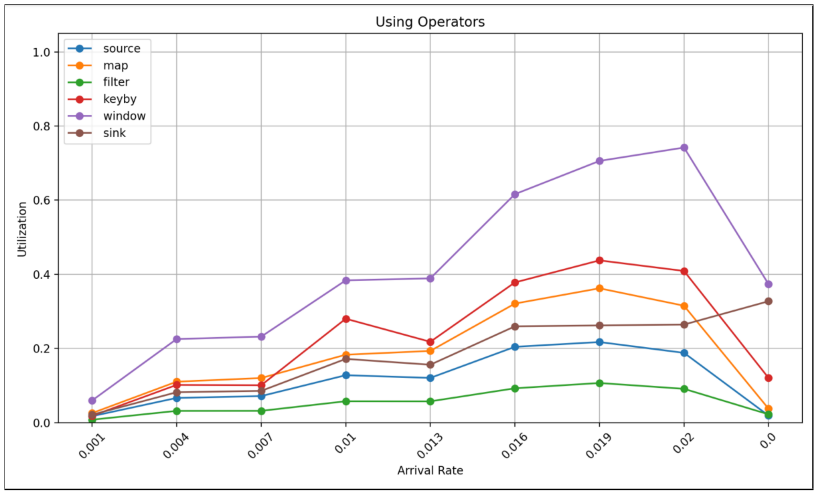

Figura 6.16: Utilización de operadores, todos con nivel de replicación = 5.

Capítulo 7

Implantación

Los detalles de este capítulo relacionados con los requerimientos del sistema, preparación del ambiente, documentación y manual de usuario se encuentran disponibles en el repositorio público de la implementación del modelo de simulación desarrollado en este trabajo [61]. Este repositorio incluye un manual de usuario completo, scripts de automatización y documentación técnica que permiten replicar y extender los experimentos realizados.

Capítulo 8

Conclusiones

Este trabajo destaca la importancia de preveer los recursos computacionales necesarios para ejecutar aplicaciones de procesamiento de datos en tiempo real de manera eficiente, evitando tanto la sobreestimación de capacidad como el uso innecesario de recursos y tiempo en pruebas sobre el sistema real. Para ello, se propone un modelo de simulación que permite evaluar el rendimiento de dichas aplicaciones sin recurrir a su ejecución directa en infraestructura productiva.

El desarrollo implicó, en primer lugar, comprender los sistemas de procesamiento distribuido en tiempo real, considerando como base un cluster genérico —estructura común donde operan plataformas como Apache Flink—. Luego, se profundizó en el caso de estudio de Flink para identificar los componentes arquitectónicos y operativos clave que deben reflejarse en el modelo.

Actualmente, el modelo incorpora elementos esenciales del sistema, como la cantidad de máquinas y núcleos de procesamiento, y simula el comportamiento general de la plataforma. A futuro, se requiere validar estos resultados frente a ejecuciones reales y ajustar la granularidad del modelo cuando sea necesario, tanto en los aspectos de hardware como en los de plataforma.

Finalmente, construir modelos computacionales que imitan la realidad exige adoptar una mentalidad experimental: iterar, probar, aprender y refinar continuamente. Este proceso no solo mejora la fidelidad del modelo frente al sistema real, sino que también permite desarrollar soluciones más robustas y significativas para la planificación de infraestructura en entornos de procesamiento de datos en tiempo real.

Bibliografía

[1] S. Kamburugamuve and G. Fox, "Survey of distributed stream processing," *Bloomington: Indiana University*, p. 153, 2016.

[2] A. S. Foundation, "Learn flink: Hands-on training," Online, 2011, available: https://nightlies.apache.org/flink/flink-docs-release-1.20/docs/learn-flink/overview/.

[3] ——, "Arquitectura flink," Online, 2011, available: https://nightlies.apache.org/flink/flink-docs-release-1.20/docs/concepts/flink-architecture/.

[4] G. A. Wainer, *Discrete-event modeling and simulation: a practitioner's approach*. CRC press, 2017.

[5] J. Santana, A. Inostrosa-Psijas, F. Moreno, M. Oyarzún, and G. Wainer, "Devs modeling and simulation of the loading and hauling process in open pit mines," in *2023 Winter Simulation Conference (WSC)*. IEEE, 2023, pp. 3082–3093.

[6] 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)*. IEEE, 2023, pp. 3130–3141. [Online]. Available: https://doi.org/10.1109/WSC60868.2023.10407840

[7] 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://doi.org/10.1007/s00778-023-00819-8

[8] C. E. Blanco, "Modelado y simulación de aplicaciones stream processing con petri nets," Master's thesis, Universidad Nacional de San Luis, San Luis, Argentina, 2024.

[9] A. S. F. LinkedIn, "Apache kafka," Online, 2011, available: https://kafka.apache.org/.

[10] J. Kreps, "What is the relationship between kafka, the writer, and apache kafka, the distributed messaging system?" 2014. [Online]. Available: https://www.quora.com/What-is-the-relation-between-Kafka-the-writer-and-Apache-Kafka-the-distributed-messaging-system/answer/Jay-Kreps

[11] A. S. F. Twitter, "Apache storm," Online, 2011, available: https://storm.apache.org/.

[12] A. W. Services, "Amazon kinesis," Online, 2013, available: https://aws.amazon.com/es/kinesis/.

[13] Google, "Pub/sub," Online, 2016, available: https://cloud.google.com/pubsub?hl=es-419.

[14] A. S. Foundation, "Apache flink," Online, 2011, available: https://flink.apache.org/.

[15] A. W. Services, "What is batch processing?" Online, 2023, available: https://aws.amazon.com/what-is/batch-processing/.

[16] U. Çetintemel, D. Abadi, Y. Ahmad, H. Balakrishnan, M. Balazinska, M. Cherniack, J.-H. Hwang, S. Madden, A. Maskey, A. Rasin *et al.*, "The aurora and borealis stream processing engines," *Data stream management: processing high-speed data streams*, pp. 337–359, 2016.

[17] W. Thies, M. Karczmarek, and S. Amarasinghe, "Streamit: A language for streaming applications," in *Compiler Construction: 11th International Conference, CC 2002 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8–12, 2002 Proceedings 11*. Springer, 2002, pp. 179–196.

[18] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo, "Spade: The system s declarative stream processing engine," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, 2008, pp. 1123–1134.

[19] Q. Anderson, *Storm real-time processing cookbook*. Packt Publishing, 2013.

[20] 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.

[21] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas, "State management in apache flink®: consistent stateful distributed stream processing," *Proceedings of the VLDB Endowment*, vol. 10, no. 12, pp. 1718–1729, 2017. [Online]. Available: https://doi.org/10.14778/3137765.3137777

[22] A. S. Foundation, "Apache samza," Online, 2013, available: https://samza.apache.org/.

[23] 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.

[24] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter heron: Stream processing at scale," in *Proceedings of the 2015 ACM SIGMOD international conference on Management of data*, 2015, pp. 239–250.

[25] T. Buddhika and S. Pallickara, "Neptune: Real time stream processing for internet of things and sensing environments," in *2016 IEEE international parallel and distributed processing symposium (IPDPS)*. IEEE, 2016, pp. 1143–1152.

[26] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle, "Millwheel: Fault-tolerant stream processing at internet scale," *Proceedings of the VLDB Endowment*, vol. 6, no. 11, pp. 1033–1044, 2013.

[27] A. Basak, K. Venkataraman, R. Murphy, and M. Singh, *Stream Analytics with Microsoft Azure: Real-time data processing for quick insights using Azure Stream Analytics*. Packt Publishing Ltd, 2017.

[28] G. S. Bedi and A. Singh, "Big data analysis with dataset scaling in yet another resource negotiator (yarn)," *International Journal of Computer Applications*, vol. 92, no. 5, 2014.

[29] P. Saha, A. Beltre, and M. Govindaraju, "Exploring the fairness and resource distribution in an apache mesos environment," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, 2018, pp. 434–441.

[30] A. S. Foundation, "Apache hadoop yarn," Online, 2012, available: https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/YARN.html.

[31] Google, "Kubernetes," Online, 2014, available: https://kubernetes.io/.

[32] A. S. Foundation, "Flink: Glosary," Online, 2011, available: https://nightlies.apache.org/flink/flink-docs-release-1.20/docs/concepts/glossary/.

[33] ——, "Flink: Jobmanager high availability," Online, 2011, available: https://nightlies.apache.org/flink/flink-docs-release-1.20/docs/deployment/ha/overview/.

[34] ——, "Flink: Task chaining and resource groups," Online, 2011, available: https://nightlies.apache.org/flink/flink-docs-release-1.20/docs/dev/datastream/operators/overview/#task-chaining-and-resource-groups.

[35] J. S. Bardi, *The calculus wars: Newton, Leibniz, and the greatest mathematical clash of all time*. Hachette UK, 2009.

[36] L. Lapidus and G. F. Pinder, *Numerical solution of partial differential equations in science and engineering*. John Wiley & Sons, 1999.

[37] D. Goldsman and P. Goldsman, "Discrete-event simulation," in *Modeling and Simulation in the Systems Engineering Life Cycle: Core Concepts and Accompanying Lectures*. Springer, 2015, pp. 103–109.

[38] R. E. Nance, "A history of discrete event simulation programming languages," in *History of programming languages—II*. NA, 1996, pp. 369–427.

[39] B. P. Zeigler, A. Muzy, and E. Kofman, *Theory of modeling and simulation: discrete event & iterative system computational foundations*. Academic press, 2018.

[40] 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.

[41] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," *ACM SIGCOMM computer communication review*, vol. 38, no. 4, pp. 63–74, 2008.

[42] K. Jensen and K. Jensen, *Coloured petri nets*. Springer, 1997, vol. 1.

[43] M. Westergaard and H. M. W. Verbeek, "Cpn tools," Online, 2018, available: https://cpntools.org/.

[44] M. Serrate, "Analysis of twitter streams with kafka and storm," Online, 2016, available: https://serrate.net/.

[45] V. Gil-Costa, E. Tapia, and M. Marin, "Asynchronous approximate simulation algorithm for stream processing platforms (wip)," in *Proceedings of the Summer Computer Simulation Conference*, 2016, pp. 1–6.

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

[47] W. A. Higashino, M. A. Capretz, and L. F. Bittencourt, "Cepsim: A simulator for cloud-based complex event processing," in *2015 IEEE International Congress on Big Data*. IEEE, 2015, pp. 182–190.

[48] V. Gil-Costa, A. Inostrosa-Psijas, and M. Marin, "Elastic and real-time capacity planning for web search engines," in *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE, 2020, pp. 331–338.

[49] A. Inostrosa-Psijas, V. Gil-Costa, M. Marin, and G. Wainer, "Semi-asynchronous approximate parallel devs simulation of web search engines," *Concurrency and Computation: Practice and Experience*, vol. 30, no. 7, p. e4149, 2018.

[50] G. A. Muñoz, V. Gil-Costa, and M. Marin, "Efficient simulation of natural hazard evacuation for seacoast cities," *International Journal of Disaster Risk Reduction*, vol. 81, p. 103300, 2022.

[51] R. Franceschini, P.-A. Bisgambiglia, L. Touraille, P. Bisgambiglia, and D. Hill, "A survey of modelling and simulation software frameworks using discrete event system specification," in *2014 Imperial College Computing Student Workshop*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2014.

[52] Y. Van Tendeloo and H. Vangheluwe, "An evaluation of devs simulation tools," *Simulation*, vol. 93, no. 2, pp. 103–121, 2017.

[53] G. Wainer, "Cd++: a toolkit to develop devs models," *Software: Practice and Experience*, vol. 32, no. 13, pp. 1261–1306, 2002.

[54] C. Ruiz-Martin and G. Wainer, "Cadmium. a tool for devs modeling and simulation. user's guide," *A Tool for DEVS Modeling and Simulation. User's Guide*, 2019.

[55] G. Quesnel, R. Duboz, and É. Ramat, "The virtual laboratory environment–an operational framework for multi-modelling, simulation and analysis of complex dynamical systems," *Simulation Modelling Practice and Theory*, vol. 17, no. 4, pp. 641–653, 2009.

[56] J.-S. Bolduc and H. Vangheluwe, "A modeling and simulation package for classic hierarchical devs," *MSDL, School of Computer McGill University, Tech. Rep*, 2002.

[57] Y. Van Tendeloo and H. Vangheluwe, "The modular architecture of the python (p) devs simulation kernel," in *Proceedings of the 2014 Symposium on Theory of Modeling and Simulation-DEVS*, 2014, pp. 387–392.

[58] GNU, "Using the gnu compiler collection (gcc)," Online, 2024, available: https://gcc.gnu.org/onlinedocs/gcc-3.3.6/gcc/G_002b_002b-and-GCC.html.

[59] B. Stroustrup, "The c++ programming language," Online, 2024, available: https://www.stroustrup.com/C++.html.

[60] D. C. Schmidt, "The c++ standard template library," *unknow*, 2014.

[61] E. Zúñiga, "Devs-models," Online, 2025, available: https://github.com/NZAE000/DEVS-MODELS.git.

[62] N. B. Contributors, "Nexmark: A benchmark for streaming query processing," https://github.com/nexmark/nexmark, 2016, accessed: 2025-06-03.

[63] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl, "Benchmarking distributed stream data processing systems," in *2018 IEEE 34th international conference on data engineering (ICDE)*. IEEE, 2018, pp. 1507–1518.

[64] F. Hueske and V. Kalavri, *Stream processing with Apache Flink: fundamentals, implementation, and operation of streaming applications*. O'Reilly Media, 2019.