UNIVERSIDAD DE SANTIAGO DE CHILE

FACULTAD DE INGENIERÍA

Departamento de Ingeniería Informática

Algoritmos de predicción y distribución de carga para el grafo lógico en los sistemas de procesamiento de *stream*

Daniel Pedro Pablo Wladdimiro Cottet

Profesor guía: Nicolás Hidalgo Castillo
Profesor co-guía: Erika Rosas Olivos

Tesis de grado presentada en
conformidad a los requisitos para
obtener el grado de Magíster en
Ingeniería Informática

Santiago – Chile
2015

Gracias a la vida que me ha dado tanto
Me ha dado la risa y me ha dado el llanto,
Así yo distingo dicha de quebranto
Los dos materiales que forman mi canto
Y el canto de ustedes que es el mismo canto
Y el canto de todos que es mi propio canto.

AGRADECIMIENTOS

Agradezco a

TABLA DE CONTENIDO

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

ÍNDICE DE ALGORITMOS

RESUMEN

resumenBlaBlaBla

Palabras Claves: SPS;Elasticidad;Algoritmos reactivos

ABSTRACT

abstractBlaBlabla

Keywords: SPS;Elastic;Algorithm reactive;Algorithm predictive

CAPÍTULO 1. INTRODUCCIÓN

1.1 ANTECEDENTES Y MOTIVACIÓN

La gran contribución de información en la Internet se ha debido al origen de la Web 2.0, donde ésta se caracteriza por la participación activa del usuario, siendo reflejado en el auge de blogs, redes sociales u otras aplicaciones web (Oberhelman, 2007). Debido a lo anterior, se crean sistemas de procesamiento para grandes cantidades de información generadas por la interacción entre los usuarios.

Es así como con el tiempo se han ido creando distintas aplicaciones de *streaming*, debido al interesante funcionamiento que poseen, las que se caracterizan por ser capaces de procesar grandes flujos de datos en tiempo real (Chen & Zhang, 2014). La necesidad de procesar información en tiempo real surge dado que muchas aplicaciones, donde sus usuarios requieren de respuestas rápidas y actualizadas que le permitan tomar decisiones en períodos cortos de tiempo. Dentro de los ejemplos existentes se encuentran; análisis de sentimientos de los mensajes de usuarios, análisis de los precios de la bolsa de valores, recopilación de información en caso de emergencia, entre otros. Las distintas aplicaciones que se han creado se volvieron críticas para sus usuarios, debido que sustenta la toma de decisiones de empresas o instituciones (Wenzel, 2014).

Un ejemplo de esto, es aplicaciones que analizan las redes sociales en caso de un desastre natural, donde grandes cantidades de información son procesadas, procesando esta información lo más cercano al tiempo real para obtener información que sea relevante para la situación (Andrade et al., 2014). De esta manera, se puede construir un sistema distribuido que pueda procesar los datos realizando análisis de sentimiento, búsqueda de palabras claves o filtros de búsqueda, ya sea por idioma, país o género, realizando un procesamiento lo más cercano al tiempo real. Con esta información, se puede realizar análisis de sectores críticos, búsqueda de personas o aviso de alertas, lo cual sería crucial

para tomar decisiones en estos momentos.

Por otra parte, también es utilizado estos sistemas de procesamiento para predicciones en la bolsa de comercio, de esta manera, se crean sistemas de procesamiento de los datos que vayan llegando en el día, de tal manera con estos datos existan modelos matemáticos que predicen el comportamiento para el siguiente día. Con estos sistemas, la ganancia que existe por parte de las personas interesadas puede aumentar considerablemente, por lo que ha generando un alto interés en el desarrollo e investigación en esta área con el transcurso del tiempo.

También se aplica en casos de seguridad, dado que se realiza un monitoreo de la actividad que surge por parte de los usuarios que interactúan en una red específica. Esto es útil para empresas o ministerios que poseen información privilegiada, y en caso que alguien desee realizar respaldos o eliminar información sin consentimiento de los encargados, puede detectarse la persona y generarse una alarma de preventiva a las autoridades. Como la información es procesada en tiempo real, la cantidad de datos que pueden irse procesando hace que el retraso de la información procesada sea baja, de esta manera, ayuda a detectar a tiempo las posibles acciones de usuarios maliciosos.

Entre los sistemas actuales de procesamiento de *stream* se encuentran S4 (Neumeyer et al., 2010), Storm (Storm, 2014), Samza (Samza, 2014), entre otros, los cuales son los más utilizados como arquitectura de procesamiento en la confección de distintas aplicaciones de *streaming*. Aunque poseen bastante flexibilidad para la creación de un sistema, por la facilidad de crear distintas topologías, no lo tiene para adaptarse en el tiempo, debido a que las topologías de procesamiento generadas son estáticas, por lo que dada la naturaleza dinámica de las interacciones pueden surgir problemas de sobrecarga.

El problema de sobrecarga conlleva a una baja en el rendimiento, produciendo una pérdida de recursos, tiempo o información. Abortar este problema es crítico, puesto que implica una mejora en la exactitud y disminución en el tiempo de procesamiento, debido que al tener mayor cantidad de datos, menor tiempo de procesamiento, se mejora la información entregada. Un ejemplo

de esto, es que se posee un tiempo t para procesar n datos, de disminuir el tiempo de procesamiento total de los datos, se tendrá que en el mismo tiempo t se procesarán una cantidad $n + m$ de datos, donde m son los datos adicionales a analizar debido a la mejora del rendimiento. Como existe una mejora en la cantidad de datos para analizar, la información de salida es más exacta, debido que tiene más datos con que comparar. De esta manera, se efectúa una mejora en los recursos utilizados, debido a la disminución del tiempo de procesamiento, y una mejor calidad en la información entregada al usuario.

1.2 DESCRIPCIÓN DEL PROBLEMA

Los SPS (Sistemas de Procesamiento de *Streaming*) modelan sus aplicaciones como un grafo cuyas vértices son operadores y las aristas son flujo de datos entre los operadores. Dada su representación, puede existir sobrecarga del sistema a producto de factores físicos o lógicos. El factor físico se define como los componentes que posee la máquina, los cuales pueden ser limitantes para el sistema alojado. En cambio, el lógico se concentra en los componentes del grafo, por lo que existe una limitante en la cantidad de operadores o la cantidad de flujo existente entre los operadores.

Debido a lo anterior, existe un problema en el sistema a raíz de la sobrecarga que puede darse debido a los distintos factores lógico, como la cola de cada operador, por la falta de flexibilidad del SPS en los operadores más demandados. Esto sucede dada la condición estática del grafo, es decir la topología del grafo no cambia con el tiempo, por lo que no existe una forma para adaptarse el tráfico de manera dinámica que permita variar la carga y reducir las colas, de tal manera de mejorar el rendimiento del sistema y obtener información más precisa y en tiempo real.

1.3 SOLUCIÓN PROPUESTA

La solución propuesta consiste en el diseño de algoritmos de predicción y distribución de carga a nivel de la lógica del grafo, los cuales adaptan el grafo a las variaciones del grafo. Por lo que se propone implementar cuatro módulos que componen la estructura del sistema de distribución de carga: monitor de carga, analizador de carga, predictor de carga y administrador de réplicas.

El monitor de carga está encargado de recuperar el nivel de carga de cada uno de los operadores. Esta información es entregada a los módulos de analizador y predictor de carga , los cuales están encargados de procesarla de tal manera de ver si existe sobrecarga en el operador. Cada uno de éstos trabaja de forma independiente y tiene distintos métodos, uno proactivo y otro reactivo, de tal manera de poseer mayor exactitud en la detección de una sobrecarga.

El analizador de carga consiste en un método reactivo, el cual analiza el tráfico de los operadores en el tiempo actual, y cuantifica su carga. La sobrecarga de cada operador depende de un umbral, por lo que según ésto se envía al administrador de réplica el tráfico de cierto operador de ser necesario una replicación.

El predictor de carga consiste en un método proactivo, el cual analiza la carga de los distintos operadores según una ventana de tiempo, y predice la carga según un método predictivo. De esta manera, se determina la posible carga que existe en cierto período de tiempo futuro, donde según un umbral y un margen de error se envía el tráfico de carga de un operador al administrador de réplicas, y así analizar si es necesario una replicación.

El administrador de réplicas se alimenta de la información entregada por los dos módulos anteriores, y así toma una decisión de la administración de cada una de las réplicas de los distintos operadores. Por lo tanto, verifica cuántas réplicas son necesarias según la cantidad de tráfico de cierto operador.

Finalmente, el sistema de procesamiento constantemente está realizando un *feedback* al sistema de optimización, de tal manera que pueda admi-

nistrar las réplicas necesarias. De esta manera, se poseerá un sistema procesa información de manera más rápida, a través de este sistema de optimización con bajo *overhead*.

1.4 OBJETIVOS Y ALCANCE DEL PROYECTO

1.4.1 Objetivo general

Diseño, construcción y evaluación de un algoritmo de predicción y un algoritmo de distribución de carga para sistemas de procesamiento de *stream*.

1.4.2 Objetivos específicos

1. Diseñar e implementar un algoritmo de predicción que permita estimar la carga de los operadores.
2. Diseñar e implementar un algoritmo de distribución que permita la administración de los operadores del grafo de procesamiento de forma elástica.
3. Diseñar y construir experimentos que permitan validar la hipótesis formulada.
4. Evaluar y analizar el rendimiento del sistema a través de aplicaciones generadas sobre sistemas de procesamiento de *stream*.

1.4.3 Alcances

Dentro de los alcances y limitaciones que se tienen en el proyecto son:

- La evaluación de la solución presentada se implementará sobre un solo sistema de procesamiento de *stream*.

- Se evalúo con al menos una aplicación bajo escenarios simulados utilizando datos reales.
- La distribución de flujo de datos es a nivel de operadores y no de nodos físicos, por lo que no se analizó la carga de estos últimos.
- Los algoritmos propuestos no incluyen técnicas que garanticen el procesamiento de todo el flujo de datos.
- En la evaluación de los algoritmos propuestos se consideró el costo de comunicación de manera igualitaria para todos los operadores.

1.5 METODOLOGÍA Y HERRAMIENTAS UTILIZADAS

1.5.1 Metodología

Dado el carácter de investigación de la propuesta de tesis, se propone utilizar el método científico para la realización de ésta. Dentro de las etapas propuesta por (Hernández Sampieri et al., 2010) están:

1. Formulación de la hipótesis: “La utilización de un modelo híbrido de paralelización permitirá mejorar la distribución de carga entre los operadores de manera dinámica, logrando reducir los tiempos de procesamiento y pérdida de eventos”.
2. Elaboración del marco teórico: Exponer las investigaciones que existen sobre problemas de sobrecarga en los operadores de SPS. Así mismo, los conceptos fundamentales de estos sistemas.
3. Seleccionar el diseño apropiado de investigación: Diseñar el experimento para el problema de balance de carga a nivel lógico en un SPS, vale decir, los algoritmos de predicción y distribución. Cada ejecución de los experimentos se basan según los principios de un SPS.

4. Analizar los resultados: De deberá analizar los resultados según las estadísticas entregadas y el modelo propuesto.
5. Presentar los resultados: Elaborar el reporte de investigación y presentar los resultados en gráficos y tablas.
6. Concluir en base a los resultados de la investigación.

1.5.2 Herramientas de desarrollo

Para el procesamiento de *stream* se utilizó Apache S4 0.6.0, por lo que fue necesario para su configuración Java SE Development Kit 7. Dentro esto, el lenguaje de programación de cada una de las estructuras del sistema desarrollado fue en Java, por lo que se trabajó sobre el IDE Eclipse Standard 4.4.2, y para el prototipo del modelo matemático se utilizó MATLAB 2014a. De forma complementaria, se utilizó Texmaker 4.1 para la confección de los distintos informes requeridos y la documentación correspondiente al trabajo.

1.6 RESULTADOS OBTENIDOS

Pam pam !

1.7 ORGANIZACIÓN DEL DOCUMENTO

Pam pam pam !

CAPÍTULO 2. MARCO TEÓRICO

2.1 STREAMING

Streaming es una técnica para la transferencia de datos de forma continua, de tal manera que sea temporal y secuencial, cuyo funcionamiento se basa en el envío de datos por parte de un ente externo a un sistema de procesamiento de información, donde en caso de estar ocupado el servicio, se dejan los datos en cola. Generalmente, esto es utilizado en la interacción con la Web, como redes sociales o reproducción *online* de contenido multimedia. En la Figura 2.1 se muestra un servidor que emana un flujo de datos que llega a distintos clientes, donde cada uno de ellos procesa la información entrante, y en caso de estar ocupado el procesamiento, se guarda en un *buffer* los datos para posteriormente ser procesados.

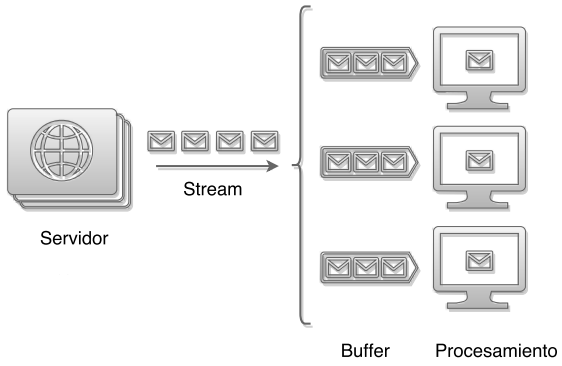

Figura 2.1: Flujo de datos entre el servidor y los clientes.

Este tipo de técnica es útil cuando se desea procesar información en tiempo real, siendo relevante la temporalidad de los datos, como la reproducción *online* de material multimedia. Los datos emanados por el *streaming* pueden ser utilizados para el análisis y procesamiento de un SPS (Sistema de Procesamiento de *Stream*). Un ejemplo de esto, es el *Streaming API* proporcionada por Twitter, donde esta información se puede utilizar para estudiar los *trending topic* o los *hashtag* más utilizados para casos específicos, como campañas electorales o desastres naturales.

2.2 STREAM PROCESSING

Stream processing es un paradigma de programación, el cual está orientado al procesamiento de un flujo de datos en tiempo real. Se centra en la programación de aplicaciones que puedan procesar la información en el momento, utilizando los recursos del sistema de forma paralela o distribuida para cumplir su objetivo, de tal manera que su procesamiento sea lo más cercano al tiempo real (Chakravarthy & Jiang, 2009).

Dentro de las aplicaciones existentes en el procesamiento de *stream*, están el monitoreo de signos vitales, detección de fraudes, reproducción de videos *online*, y para cada uno de ellos es necesario cumplir con ciertas características para el funcionamiento correcto del sistema. Para ello, se han propuesto ciertos requerimientos para el procesamiento continuo de datos (Andrade et al., 2014), los cuales serán desglosados a continuación:

- **Procesamiento de grandes cantidades de datos**: esto significa que al tratar de procesar los datos, no se puede guardar en una base de datos y luego procesarlos, como en general lo realizan los *bash processing*, por lo tanto es necesario otro mecanismo que pueda procesarlos mientras va llegando la información entrante. Por lo que utilizar *stream processing* soluciona este problema, dado que la información entrante es procesada a medida que van llegando los datos.
- **Limitaciones de ancho de banda y latencia**: se refiere a la comunicación que existe por parte del proveedor de datos, de tal manera que no sea una limitante en el procesamiento de los datos el ancho de banda o la latencia que existe. Esto es importante, dado que no sirve un sistema de estimación de la bolsa del mercado que producto de la latencia existente, envíe datos obsoletos. Siempre se debe mantener una baja latencia, para poseer los datos lo más cercano al tiempo real.
- **Procesamiento de datos heterogéneos**: en su mayoría, los datos poseen distintos formatos, contenidos y niveles de ruido, por lo que es necesario

realizar una normalización de estos, de tal manera de estandarizar el procesamiento.

- **Proporcionar alta disponibilidad a largo plazo**: es importante poseer un constante flujo de información, que sea estable y persistente en el tiempo, de tal manera que esté procesando constantemente los datos para el propósito designado. Por ejemplo, si se posee un sistema de análisis de partículas en el espacio, es necesario que posea una tolerancia a falla, para que en el caso que exista una anomalía, siga manteniéndose una disponibilidad por parte del sistema, y no se pierda el objetivo de observar en tiempo real y procesar esa información entrante.

2.3 SISTEMAS DE PROCESAMIENTO DE STREAM

Entre los diferentes motores de procesamiento de datos masivos, existen los sistemas de procesamiento de *stream*, los cuales reciben grandes cantidades de datos que deben procesar de forma distribuida y en tiempo real, de ahora en adelante hablaremos de procesamiento *online* para hacer referencia al tiempo real. Para realizar esto, se requiere un cambio en el paradigma tradicional de *bash processing*, el cual almacena los datos en una base de datos, los que posteriormente son procesados de forma *offline* (Hawwash & Nasraoui, 2014), a uno que procese de forma *online*.

El paradigma utilizado se basa en grafos de procesamiento como muestra la Figura Figura 2.2, donde los operadores corresponden a las vértices del grafo, como por ejemplo analizadores de sentimientos, filtros de palabras o algún algoritmo en particular, y las aristas corresponden a los flujos de datos entre un operador y otro (Shahrivari, 2014). Además de esto, los datos proporcionados son originados por un ente externo, ya sea *streaming* de datos de redes sociales, estadísticas del monitoreo de un sistema u otra información deseada en tiempo *online*, la cual entrega los datos iniciales a los primeros operadores del grafo

(Appel et al., 2012).

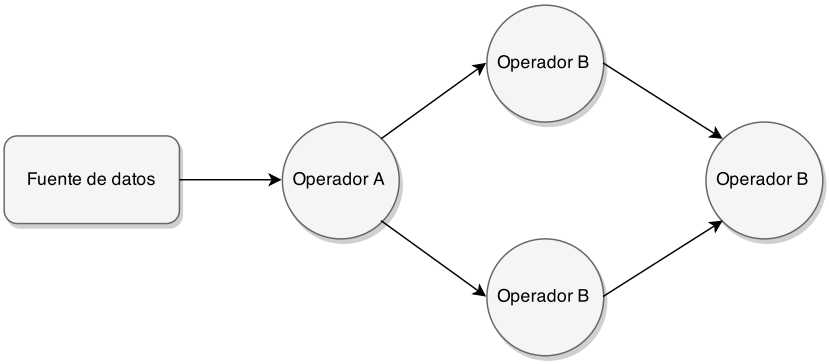

Figura 2.2: Ejemplo de modelo de SPS.

Cabe destacar que al ser distribuido los SPS, cada uno de los vértices del grafo son alojados en un nodo físico disponible en el ambiente en que se aloja el sistema, ya sea un *cluster*, un *grid* o un *cloud*. Por lo tanto, se debe realizar la comunicación entre los distintos nodos, para realizar el envío del flujo de datos de un operador a otro.

Las principales aplicaciones que se le dan a estos SPS son orientados en el manejo de grandes cantidades de datos, los cuales deben ser procesados para obtener información o estadísticas, como es el caso de detección de fraudes, recolección de información en caso de desastres o análisis de la interacción en las redes sociales. Para efectuar un procesamiento en tiempo real de los datos, (Stonebraker et al., 2005) propone los siguientes requerimientos:

- **Baja latencia**: este concepto está asociado con la comunicación fluida entre los distintos nodos que estén trabajando en el sistema, de tal manera que no existan *delay* o retrasos en el procesamiento del sistema.
- **Consultas SQL**: poder realizar consultas a una base de datos, sin perder las propiedades del SPS, como el procesamiento distribuido. Para esto, se debe realizar un cambio en la forma de ejecutar las consultas, debido que no sólo es necesario realizar la consulta, sino también realizar un *merge* de las respuestas, por lo que es necesario diseñar un sistema que cumpla con operadores adicionales a los utilizados en las consultas tradicionales por sistemas centralizados.

- **Manejo de fallas en el flujo de dato**: es importante poseer sistemas que no se preocupen de la pérdida en los datos, debido que se posee como premisa que se van a perder datos en el procesamiento de estos, ya sea por las colas, *delays* u otro problema asociado al procesamiento o la fuente de datos. Por lo tanto, al modelar la aplicación no es necesario lidiar con este tipo de fallas.

- **Generar resultados predecibles**: cuando se realizan consultas en el sistema, existe la posibilidad que sean correctas sólo por un período de tiempo, debido a alguna falla en el sistema que genere una pérdida en el estado del operador. Por lo tanto, es necesario garantizar que el resultado sea predecible y persistente en el tiempo, ya sea respaldando la información u otro mecanismo, de tal manera que si se realiza una consulta, el resultado sea consistente u homólogo con el transcurso del tiempo.

- **Integrar almacenamiento y flujo de datos**: en general, cuando se trabaja con procesamiento de datos, es importante guardar estados en el sistema, de tal manera que los datos entrantes vayan verificando, modificando o eliminado la información que se posea. En un operador que cuente palabras, es necesario soportar variables que guarden las estadísticas de la información entrante. Otro tema importante es la uniformidad de los datos, como se había presentando en el tópico anterior de *Streaming*, siempre se va a trabajar con datos heterogéneos, por lo que se requiere estandarizarlos para su procesamiento, de esta manera, no existirá una discordancia en la información procesada.

- **Garantizar la seguridad y disponibilidad de los datos**: este requerimiento está orientado en poseer mecanismos de *checkpoint*, técnica utilizada para respaldar el estado del operador cada cierto período de tiempo, y tolerancia a falla, por lo que en caso de existir alguna anomalía, pueda volver el sistema a estar disponible y sin perder una cantidad considerable de información, ya sea en las estadísticas o estados del sistema.

- **Partición y escalabilidad automática de las aplicaciones** Es importante también distribuir la carga entre distintos procesadores o máquinas, desean-

do idealmente una escalabilidad incremental, esto significa que el flujo de datos sea entregado a los distintos recursos que se posean y en caso de necesitar más recursos, incrementar lo que se poseen (Tanenbaum & van Steen, 2007). Si bien no sucede siempre, se espera que esto sea automático y transparente.

- **Procesamiento y respuesta instantánea**: cuando se plantea el uso de los SPS, se apuesta por un sistema que entregue respuestas en un tiempo lo más cercano al real, este requerimiento hace necesario lidiar posibles sobrecargas de los operadores, las cuales afectan al rendimiento del sistema. Por lo tanto, se hace necesario abordar estos posibles escenarios proveyendo una solución con una optimización de bajo *overhead*, esto quiere decir con bajo costo de implementación o recursos necesarios para su funcionamiento, aumentando la eficiencia y rendimiento del sistema.

Cada sistema de procesamiento de *streaming* está basado en un modelo de procesamiento en particular. Por ejemplo, S4 está basado en el modelo de procesamiento *push* (Neumeyer et al., 2010), y Storm en el modelo *pull* (Storm, 2014).

El primer modelo consiste en el envío de datos desde el operador. La ventaja de este modelo empleado por S4 radica en la abstracción en el envío de datos, sin embargo no asegura el procesamiento de estos, debido a que no existe un mensaje de respuesta al ser entregado al operador. En la Figura 2.3 se puede ver el Operador A como envía los datos al Operador B, donde en caso que esté procesando un dato el Operador B, éste lo guardará en cola. No existe algún mecanismo para asegurar que llegue efectivamente el dato, puede darse el caso que por falla de la red no se haya enviado u otro motivo, por lo que existe una abstracción en el envío de los datos.

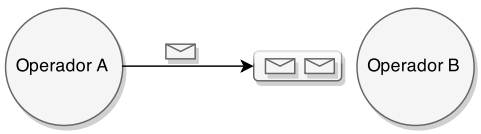

Figura 2.3: Modelo push de procesamiento.

Por otra parte, el segundo modelo se basa en la petición de datos a un operador, por lo que son enviados solo si son requeridos. Si bien este modelo asegura procesamiento de los datos, genera una menor abstracción al programador, dado que en el primer modelo sólo se indica a que operador deben ir los datos, en cambio en el segundo se debe indicar quién lo envía y quién lo recibe. En la Figura 2.4 se puede ver que existen dos operadores, donde en la parte (a) se solicita por parte del Operador B el envío de un dato para ser procesado, donde en la parte (b) el Operador A envía el dato para que posteriormente sea procesado por el Operador B.

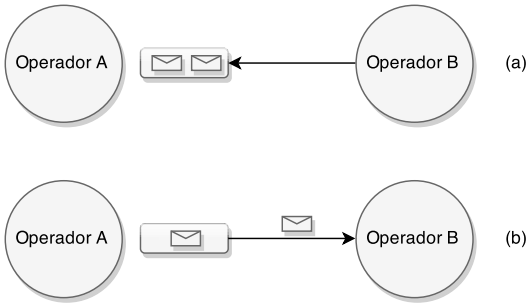

Figura 2.4: Modelo pull de procesamiento.

2.3.1 Simple Scalable Streaming System (S4)

S4 (Neumeyer et al., 2010) es un sistema de propósito general, distribuido y escalable que permite que aplicaciones puedan procesar flujos de datos de forma continua y sin restricciones. S4 está inspirado en MapReduce (Lin & Dyer, 2010), y fue diseñado en el contexto de minería de datos y algoritmos de aprendizaje de máquina en Yahoo! Labs para sistemas de publicidad *online*. Cada evento en S4 es descrito como un par (clave, atributo), cuyos pares pueden ir agregándose a medida que sea necesario. La unidad básica son los elementos de procesamiento (PEs, por sus siglas en inglés) y los mensajes que son intercambiados entre ellos. Los PEs pueden emitir o pueden publicar resultados y son alojados en servidores llamados nodos de procesamiento, llamados PNs.

Los PNs son responsables de escuchar eventos, rutear eventos a los PEs del nodo y despachar eventos a través de la capa de comunicación. Los eventos son encaminados usando una función de *hashing* sobre los valores de los atributos hacia el PE apropiado. Por otro lado, la capa de comunicación utiliza Zookeeper (Hunt et al., 2010), el cual provee manejo de clusters y reemplazo automático de nodos que fallan. S4 usa encaminamiento estático, es parcialmente tolerante a fallas, y no posee mecanismos de balanceo dinámico de carga.

2.3.2 Storm

Storm (Leibiusky et al., 2012) es una plataforma similar a S4, la cual es publicada como una API para la computación con flujos de datos en tiempo real y de forma escalable. El modelo de programación está basado en dos primitivas básicas para la transformación de flujos de datos que deben ser implementados de acuerdo a la lógica de las aplicaciones: *Spouts* y *Bolts*. Un *Spout* es una fuente de flujo de datos y un *Bolt* hace una transformación de un solo paso sobre el flujo de datos, creando un nuevo flujo basado en la entrada que recibe. Transformaciones complejas requieren múltiples *Bolts*, los cuales crean topologías o grafos, el nivel más alto de abstracción en Storm. La plataforma soporta tolerancia a fallas a través de un proceso maestro llamado Nimbus (Miao et al., 2014), el cual garantiza el procesamiento de todos los mensajes a través del uso de una base de datos para el almacenamiento. Sin embargo, esta base de datos es su mayor desventaja respecto de S4 puesto que no es completamente distribuido. Storm define diferentes técnicas para el particionamiento de *streams* de datos y para la paralelización de *Bolts*, por lo tanto la asignación de máquinas para alguna actividad, deberá efectuarse de forma manual. Sin embargo, se complejiza el desarrollo de aplicaciones. Al igual que S4, Storm usa Zookeeper (Hunt et al., 2010) en la capa de comunicación.

2.4 ELASTICIDAD

La propiedad de elasticidad en el área de *Cloud Computing* o *SPS*, está relacionado con la capacidad que el sistema tiene de adaptarse dinámicamente a las condiciones cambiantes del sistema, como por ejemplo el tráfico. Esto quiere decir que aumente o disminuya los recursos que se utilicen, para que funcione de manera eficiente.

En el caso de *Cloud Computing*, existen estudios que han trabajado con esta propiedad como (Gong et al., 2010; Nguyen et al., 2013; Lehrig et al., 2015), donde el sistema se comporta de forma elástica, determinando dinámicamente la cantidad de máquinas virtuales necesarias en el sistema. Por otra parte, en los SPS, existen trabajos como (Gedik et al., 2014; Ishii & Suzumura, 2011; Schneider et al., 2009; Madsen et al., 2014; Gulisano et al., 2012), en que el sistema de forma dinámica determina la cantidad de operadores necesarios para realizar una tarea en específico, como se ve representando en la Figura 2.5, donde la cantidad de operadores B cambia dinámicamente según el rendimiento del sistema.

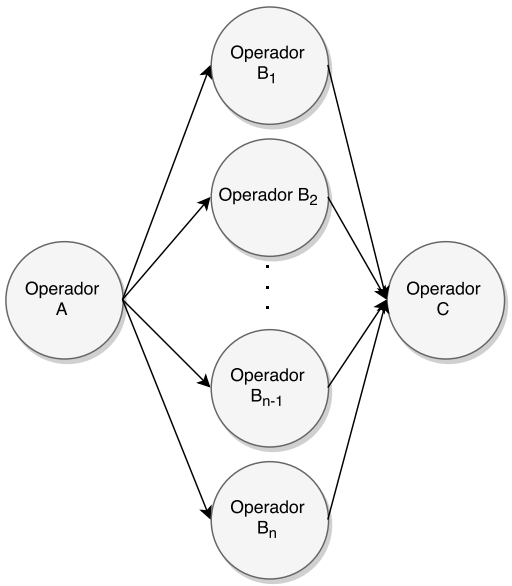

Figura 2.5: Elasticidad en un SPS.

Un ejemplo práctico de elasticidad es el supermercado, donde se debe

considerar la cantidad de cajas necesarias para atender de manera eficiente los n clientes que van llegando en un período de tiempo. Si se estudia el período de la mañana, en general, es tiene un bajo flujo de personas que acude al supermercado, en comparación con la tarde, pero alto con la medianoche. Por lo tanto, en los horarios de la tarde es necesario poseer una mayor cantidad de cajas disponibles que en la tarde, disminuyendo la cantidad cuando el horario borde la media noche, adaptándose de forma elástica la cantidad de cajas disponibles en el supermercado.

En el trabajo realizado, se propone un sistema elástico según la necesidad del sistema dada la demanda de los distintos operadores. De esta manera, según el tráfico existente aumenta o disminuye los operadores, de tal manera que trabaje de manera dinámica y de forma óptima.

2.5 PROCESOS ESTOCÁSTICOS

Se define proceso estocástico como una colección de variables aleatorias X_t, con $t \epsilon T$, las cuales están determinadas por algún comportamiento en el tiempo t. Esto significa que cada variable estará tratada de forma discreta en el tiempo, sin poseer un proceso determinístico entre sus variables, es decir, que las variables dependan de la historia (Taylor & Karlin, 2014).

Por lo tanto, se puede definir un estado como el posible comportamiento que puede tener una variable aleatoria en el sistema. Un ejemplo de esto es un modelo que contemple tres estados: estable, inestable y ocioso, y que según el valor de la variable aleatoria vaya cambiando de un estado a otro. Un caso de estudio utilizando el concepto de estados son las cadenas de Markov, las cuales consideran distintos estados que representan un comportamiento del sistema, habiendo una probabilidad de cambiar de un estado a otro (De Sapio, 1978).

Como se mencionó anteriormente, las cadenas de Markov son procesos estocásticos, las cuales se utilizaron para el predictor de carga en el sistema modelado. De esta manera, se definió estados que fueran

independientes entre si, cuyas muestras para definir los estados no dependieran con el tiempo. Esto fue pensando con el fin de realizar análisis a futuro, tomando en consideración los datos *a priori*.

2.5.1 Cadena de Markov

Sea X_t el valor de una variable aleatoria X en un tiempo t. El conjunto de todos los valores posibles para X se llama espacio de estado (Ching & Ng, 2006). La variable aleatoria es un proceso de Markov si las probabilidades de transición entre dos estados cualquiera de Ω (definido como el universo de posibles estados), sólo depende del estado actual, como se denota en la Ecuación 2.1 y gráficamente en la Figura 2.6. Cabe destacar que este tipo de proceso es un caso específico de los procesos estocásticos.

$$P_r(X_{t+r} = S_j | X_0 = S_k; X_1 = S_l; ...; X_t = S_i) = P_r(X_{t+1} = S_j | X_t = S_i) \tag{2.1}$$

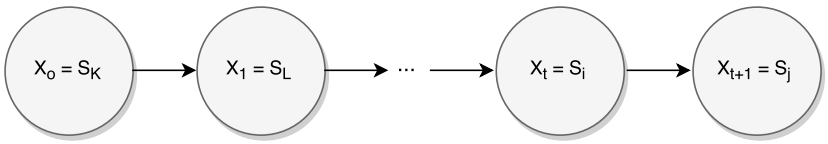

Figura 2.6: Proceso de Markov.

Una cadena de Markov es una secuencia de variables aleatorias generadas por un proceso de Markov, como se denota en la Ecuación 2.2.

$$(X_0, X_1, X_2, ..., X_{n-1}, X_n) \tag{2.2}$$

La Ecuación 2.3 se define por sus probabilidades de transición. En la Figura 2.7 se muestra un ejemplo de la transición del estado i al estado j, dada la probabilidad P_{ij}.

$$P_{ij} = P_r(i \to j) = P_r(X_{t+1} = S_j | X_t = S_i) \tag{2.3}$$

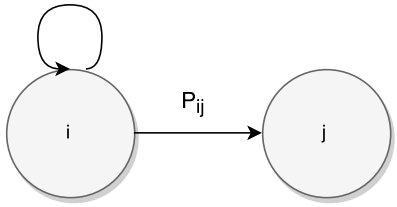

Figura 2.7: Cadena de Markov.

En la Ecuación 2.4 se presenta una matriz de transición de finitos estados, donde la probabilidad de pasar de un estado a otro está determinado por una posición de la matriz, tomando en consideración que la suma de todas las transiciones de un estado debe ser igual a 1.

$$P = \begin{bmatrix} P_{1,1} & P_{1,2} & \cdots & P_{1,n} \\ P_{2,1} & P_{2,2} & \cdots & P_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ P_{n,1} & P_{n,2} & \cdots & P_{n,n} \end{bmatrix} \qquad \sum_{j=1}^{n} P_{ij} = 1; \forall i \tag{2.4}$$

En la Figura 2.8 se muestra un ejemplo de una cadena de Markov simple, donde se analiza la probabilidad del clima de mañana dado el clima de hoy día. Como se puede observar, no se considera la historia del clima en la semana, sólo en el caso actual, lo cual es aplicado en los procesos estocásticos. Dada las probabilidades que transite de un clima a otro, se puede ver en la Ecuación 2.5 la matriz de transición resultante.

$$P = \begin{bmatrix} \frac{1}{2} & \frac{1}{4} & \frac{1}{4} \\ \frac{1}{2} & 0 & \frac{1}{2} \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{2} \end{bmatrix} \tag{2.5}$$

Si se desea saber la probabilidad que la cadena esté en el estado S_i en el tiempo $t+1$, está dada por la ecuación de Chapman-Kolmogórov (Papoulis, 1984):

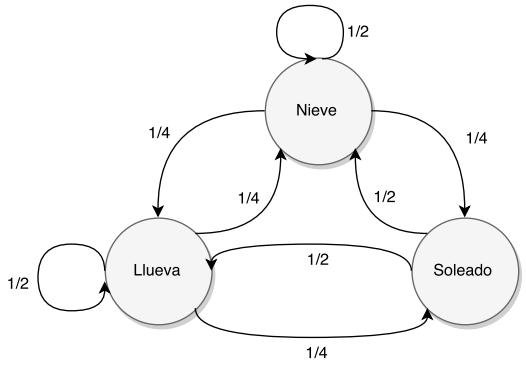

Figura 2.8: Ejemplo de cadena de Markov.

$$\begin{aligned}\Pi_i(t+1) &= P_r(X_{t+i} = S_i) \\ &= \sum_k P_r(X_{t+i} = S_i / X_t = S_k) P_r(X_t = S_k) \\ &= \sum_k P_r(X_{t+i} = S_i / X_t = S_k) \Pi_k(t)\end{aligned} \tag{2.6}$$

En notación matricial:

$$\begin{aligned}\Pi_{(t+1)} &= \Pi_{(t)} P \\ \begin{bmatrix}\Pi_1 & \Pi_2 & \Pi_3\end{bmatrix}_{(t+1)} &= \begin{bmatrix}\Pi_1 & \Pi_2 & \Pi_3\end{bmatrix}_{(t)} \begin{bmatrix} P_{1,1} & P_{1,2} & \cdots & P_{1,n} \\ P_{2,1} & P_{2,2} & \cdots & P_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ P_{n,1} & P_{n,2} & \cdots & P_{n,n} \end{bmatrix}\end{aligned} \tag{2.7}$$

Usando recurrencia, se puede calcular la distribución estacionaria como se muestra la ecuación 2.8, la cual indica el comportamiento a futuro de la cadena de Markov, dado los estados y transiciones que éste posee.

$$\begin{aligned}\Pi(t) &= \Pi(t-1)P \\ &= \Pi(t-2)P^2 \\ &= \Pi(0)P^t; \Pi(0) : \text{distribución inicial}\end{aligned} \tag{2.8}$$

2.5.2 Trabajo relacionado

Existen modelos predictivos que están basados en modelos matemáticos, los cuales simulan el comportamiento del sistema, ya sea del flujo o de la carga de un operador, de tal manera que pueda predecir como será su estado en un tiempo futuro. En general, para poder realizar una predicción se analiza las variables deseadas en una ventana de tiempo, para posteriormente aplicar un modelo matemático que prediga la variación del sistema en la próxima ventana de tiempo que se tiene estipulada.

Dentro de las aplicaciones que se han realizado con modelos predictivos, se encuentra PRESS (Gong et al., 2010). En este sistema orientado a *Cloud Computing* (Birman, 2012), analiza la cantidad de recursos disponibles, ya sea la memoria disponible o el uso promedio de CPU en las máquinas virtuales que se dispone en el *Cloud*. Para realizar la predicción del estado del sistema, se aplica un modelo basado en cadenas de Markov, tomando sus estados como ventanas de tiempo en un determinado período. De esta manera, se analiza el estado del sistema en un tiempo en específico, para analizar si posee correlación con algún estado de la cadena de Markov, para posteriormente ver la transición de ese estado a otro y generar la matriz de transición. Posteriormente, con la ecuación de Chapman-Kolmogorov, se calcula la distribución estacionaria de la matriz estacionaria, de tal manera de saber en que estado estará en la próxima ventana de tiempo, para finalmente analizar si es necesario algún cambio en el sistema.

Dentro de la misma línea de modelos predictivos, existe el sistema AGILE (Nguyen et al., 2013) para *Cloud Computing*, que modifica las máquinas virtuales de forma elástica en un *Cloud*. Lo que se realiza en este trabajo es aplicar la transformada de Fourier (Falk et al., 2012) a la carga de CPU en una ventana de tiempo determinada, donde la función resultante se analiza con distintas frecuencias, de tal manera de solicitar la predicción de la próxima ventana de tiempo a cada una de las funciones creadas. De esta manera, se sintetizan todas predicciones realizadas por cada función, para analizar el

comportamiento del sistema en la próxima ventana de tiempo, y ver si es necesario aumentar o disminuir recursos de éste.

2.6 TEORÍA DE COLAS

La teoría de colas se centra en el estudio matemático de las colas existentes en un sistema, cuyo caso de estudio era el desbordamiento de peticiones por parte del cliente al servidor (Breuer & Baum, 2005). En la Figura 2.9 se muestra un ejemplo de un sistema basado en teoría de colas, donde existe n productores que envían cierto flujo de datos a los m servidores disponibles, y en caso de no estar disponibles, se genera una cola de espera en el sistema.

- **Productor**: es quién provee la fuente de entrada para el servidor, de tal manera que procese según la necesidad que se posea.
- **Cola** o línea de espera, la cual está encargada de almacenar la información emanada por el productor en caso que los servidores estén ocupados, para que posteriormente sean procesados.
- **Servidor**: es quién procesa la información disponible en la cola, de tal manera que entre una fuente de salida con los datos o información deseada.

Además de esto, se tienen ciertos componentes importantes en el sistemas, definidos a continuación:

- **Tasa de llegada**: denotado λ, es la cantidad de datos, eventos o información que van llegando por un determinado período de tiempo, la cual está determinada por los productores que existan en el sistema.
- **Tasa de procesamiento**: denotado μ, es la cantidad de datos, eventos o información que salen del sistema, producto del servicio provisto por cada servidor.

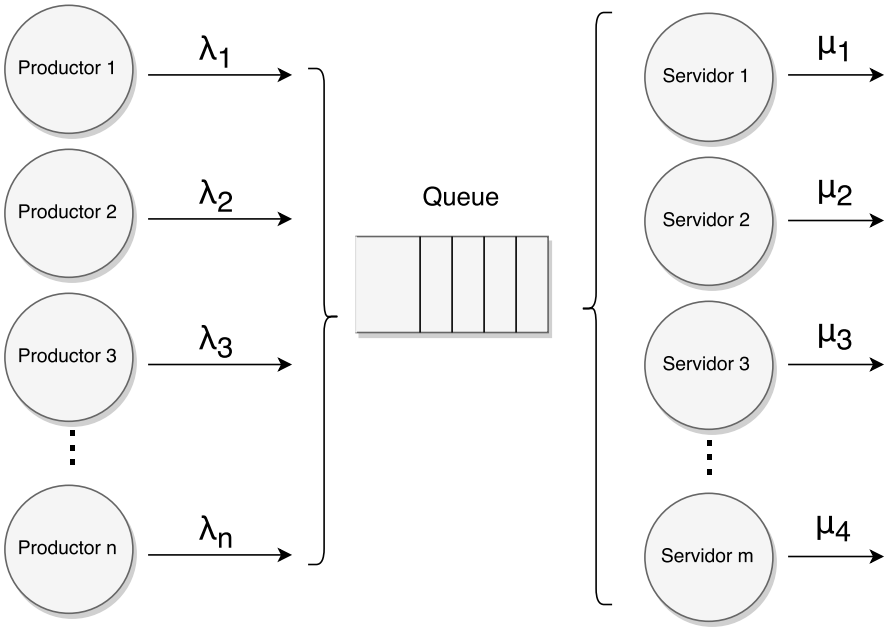

Figura 2.9: Ejemplo de un sistema basado en teoría de colas.

- **Tasa de rendimiento**: denotado ρ, es el porcentaje de utilización del sistema, donde $\rho = \frac{\lambda}{\mu}$, siendo un sistema estable si $\rho < 1$, dado que la capacidad de procesamiento es mayor que la tasa de llegada.

- **Disciplina de la cola**: significa el método utilizado para extraer los datos encolados en el sistema, para esto puede aplicarse los métodos *FIFO*, *LIFO*, *RSS*, entre otros.

Este tipo de modelos se puede aplicar en los SPS, debido que la fuente de datos es el productor y cada operador es un servidor del sistema. Por lo que existe un problema interesante a analizar, dado el dinamismo de los datos a procesar, los cuales pueden generar la sobrecarga en algún operador. Un factor importante para el desempeño de este tipo de modelos en los SPS, es que al normalizar los datos, realiza un análisis más certero de las estadísticas, debido que puede ser que la tasa de llegada λ sea demandante para un operador que posee una tarea con un alto costo de cómputo, pero para otro operador no es tan demandante, por lo que lo importante no son las tasas de llegada o procesamiento, sino el rendimiento que posea cada operador.

Los distintos problemas de sobrecarga en los operadores, se deben a que la tasa de procesamiento es menor a la tasa de llegada, creando colas en el sistema. Por ejemplo, si se posee una tasa de llegada λ y una tasa de servicio μ, donde $\mu < \lambda$, se tendrá un sistema inestable, debido que se procesa más lento de lo que llegan los datos. Como existen colas, es necesario un aumento del rendimiento del sistema, debido que $\rho > 1$, donde se define $\rho = \frac{\lambda}{s\mu}$, siendo s la cantidad de servicios disponibles. Tomando este tipo de consideraciones, se consideró que en caso que el operador posea un $\rho > 1$ el sistema se considera inestable, por lo que es necesario realizar una modificación en la cantidad de réplicas existentes de ese operador en el SPS, de tal manera de mejorar el rendimiento del sistema.

CAPÍTULO 3. BALANCE DE CARGA EN SPS

3.1 PERSPECTIVAS DE BALANCE DE CARGA

Dentro de la literatura se han encontrado distintas perspectivas al problema de balance de carga en un SPS, las cuales consideran los recursos físicos como fuente del problema de la sobrecarga del sistema, y otro enfoque que considera el recurso como el foco de éste.

3.1.1 Recursos físicos

En esta perspectiva se toma en consideración la sobrecarga del sistema dado las limitantes físicas que éste posea, ya sea por condiciones de los recursos disponibles o por el ambiente de desarrollo. Para esto, se consideran distintos parámetros como umbrales, los cuales si son sobrepasados debe aplicarse alguna estrategia para aliviar la carga del sistema. Estos umbrales pueden ser el nivel *Service Level Objective* (SLO) (Sturm et al., 2000), porcentaje de CPU utilizada o disponibilidad de la memoria (Dong & Akl, 2006).

Una de las soluciones con la perspectiva anterior es la propuesto en Borealis (Xing et al., 2005), donde considera la cantidad de carga de los nodos en ventanas de tiempo pre-determinadas, las cuales serán manejadas por un coordinador centralizado. Este coordinador se encarga de analizar los recursos del sistema, y en caso que se sobrepase el umbral propuesto, se deberán migrar los operadores que estén en ese nodo, para luego ser enviados a otro nodo candidato con menor cantidad de carga, por lo no soluciona el problema de rendimiento del sistema, sino de la máquina, debido que sólo mueve el programa de una máquina a otra. Para elegir al nodo candidato, se realiza un análisis de la cantidad de correlación que existe entre el operador y el nodo candidato, de esta manera, no necesariamente va a ser enviado a otro nodo con menor sobrecarga,

sino también a uno que posea menor cantidad de mensajes. Dentro de los problemas que pueden existir en el sistema es la conexión entre los distintos nodos, por lo que para las pruebas se considera un buen ancho de banda, de tal manera que aparente una red sin limitaciones de este tipo.

Otra de las soluciones que se han propuesto es lo realizado por Flood (Alves et al., 2010), la cual propone un DPS (*Distributed data stream processing*) que considera ciertos factores físicos para agregar o eliminar máquinas virtuales que se provee del *Infrastructure-as-Service*, como Amazon EC2. Para esto, se posee un administrador que considera las estadísticas en tiempo de ejecución como la cantidad de CPU utilizada, latencia o memoria disponible, las cuales considera para ver en que rango está de los umbrales establecidos, y posteriormente agregar o eliminar recursos de manera elástica al sistema.

3.1.2 Recursos lógicos

A diferencia de la perspectiva de recursos físicos, en esta perspectiva lógica se consideran los componentes lógicos del sistema, es decir, el foco está en los operadores y su carga de trabajo. Las distintas soluciones que se presentan, analizan componentes como el flujo de datos o el tamaño de la cola de un operador, tomando esos parámetros como umbrales en los algoritmos implementados para realizar mejoras en el sistema.

Dado esta perspectiva, se han presentando dos tipos de enfoques: el estático y el dinámico (Gupta & Bepari, 1999). El primer enfoque está centrado en un modelo definido y fijo antes de la inicialización del sistema, sin considerar el estado del mismo. En cambio, el segundo enfoque está basado en un modelo que analizará el sistema según su estado en el transcurso de su ejecución.

3.1.3 Enfoque estático

Este enfoque se ha implementando en distintos sistemas de procesamiento de *stream*, donde no se depende del estado del sistema (Storm, 2014; S4, 2014). De esta manera, no existe una interrupción en la ejecución o un cambio debido al estado del sistema (Casavant & Kuhl, 1988). Por lo tanto, no se considera variables como la carga o cola del operador, sólo se aplican técnicas que administren el flujo de los datos en el sistema.

Storm utiliza distintas técnicas de distribución de las tuplas en los operadores según la política que se desee, todas tomando el enfoque estático (Storm, 2014). Dentro de las políticas que existen están *Shuffle grouping*, *Fields grouping*, *Partial Key grouping*, *All grouping*, *Global grouping*, *None grouping*, *Direct grouping* y *Local grouping*.

La política de *Shuffle grouping* se enfoca en distribuir las tuplas de forma homogéneas en los n operadores que se encuentren en el grafo, utilizando la planificación *Round-Robin* (Brucker, 2004), de esta manera la cantidad de tuplas se distribuye de forma homogénea en el sistema. Una de las principales fallas es que la tasa de procesamiento de las tuplas no siempre es la misma, por lo tanto puede existir una sobrecarga en un operador en particular, dado que éste recibe una mayor cantidad de tuplas y requiera de un mayor tiempo de procesamiento. Otra de las políticas utilizada es *Fields grouping*, la cual determina ciertas llaves a un operador determinado. Por ejemplo, se contiene un flujo de datos que se determinan por el identificador de los usuarios, de ser así, desde cierto rango de letras corresponden a cierto operador, de tal manera de dividir equitativamente según la cantidad de caracteres existentes. Si bien genera un determinismo en el procesamiento de las llaves, puede existir una sobrecarga de un operador en particular, debido a que una llave se repite con mayor frecuencia que otras (Leibiusky et al., 2012).

Por otra parte, se encuentra el funcionamiento de S4, cuya política es similar a la de *Fields grouping* de Storm, la diferencia es que un operador no le corresponde un conjunto de llaves, sino que posee una llave única. Esto quiere

decir que cada llave se le asigna un operador, y en caso de no existir un operador para el valor de esa llave, se creará un nuevo operador de manera automática. Debido a la infinidad de combinaciones de llaves que pueden generarse, S4 recomienda aplicar una función *consistent hashing* (X11, 2011), de esta manera el valor de la función determina el operador que procesará el dato y disminuirá la cantidad de operadores que deben estar disponibles, debido que al existir menor cantidad de combinaciones, se deben crear menor cantidad de operadores al existir una nueva llave con la función *hash* planteada anteriormente. Esta técnica provee dinamismo en la cantidad de operadores en el sistema, pero al igual que la *Fields grouping* puede sobrecargar un operador, debido que una llave posee mayor frecuencia que las otras, como se expresa en la ley de potencia, debido que un porcentaje de llaves es más usada que otras, como es el caso de las letras utilizadas en el diccionario (Rushton, 2010).

Una ventaja del enfoque estático es el bajo costo de la implementación de los métodos, lo cual es beneficioso para sistemas con bajos recursos. Por otra parte, una desventaja existente es la sobrecarga de operador, debido que no asegura que la cantidad de flujo sea repartido de forma homogénea. Si bien, no es una solución óptima, es un buen complemento para un modelo con el enfoque dinámico.

3.1.4 Enfoque dinámico

Este enfoque está basado en el estado del sistema, siendo esto el parámetro base para optimizar su rendimiento (Casavant & Kuhl, 1988). Esto significa que si el sistema posee una anomalía, como una sobrecarga o alta latencia entre nodos, es necesario realizar un cambio en la lógica del sistema con el fin de solucionar el problema. En este contexto se consideran dos modelos: reactivo y predictivo.

Reactivo : este modelo está basado en la detección de sobrecargas en el sistema a través de un monitor (Gulisano et al., 2012), el cual recibe periódicamente las variables de cada uno los operadores, y en caso que sobrepase un umbral, se aplica una técnica para aumentar el rendimiento bajo una métrica dada. El umbral puede estar basado en el tiempo de procesamiento, el tamaño de la cola u otra variable del operador (Bhuvanagiri et al., 2006). Por ejemplo, para realizar una optimización en el rendimiento general del sistema, se considera la tasa de procesamiento de cada operador, por lo que en caso de existir congestión en un operador, se procede a realizar una paralelización de éste, de tal manera que exista un operador adicional que pueda recibir un flujo de datos y realizar la misma operación que el operador sobrecargado en paralelo (Schneider et al., 2009).

Si bien estas soluciones en su mayoría son eficientes y poseen buen rendimiento, uno de los principales problemas es que no analiza el comportamiento a futuro, debido que sólo analiza y resuelve la situación en el momento. Otro problema son los falsos positivos, debido que puede ser que en un momento exista un *peak* de tráfico, pero esto era sólo un caso particular del instante, por lo que llevar a cabo la replicación del operador es un costo innecesario.

Predictivo : este modelo está basado en modelos matemáticos que calculan o estiman el comportamiento a futuro del sistema, dada cierta información que se posee del sistema, como flujo entrante o carga de la CPU. Si bien no existen modelos predictivos para SPS, si los existen en otras áreas, como se presentó anteriormente en la subsección 2.5.2.

3.2 TÉCNICAS DE BALANCE DE CARGA

Existen distintas técnicas de balance de carga que utilizan alguno de los dos modelos presentados anteriormente, las cuales están enfocados a mejorar el rendimiento del sistema en caso de existir una sobrecarga (Hirzel

et al., 2013). Dentro de las técnicas existentes se encuentran la planificación determinista (Xu et al., 2014; Dong et al., 2007), *load shedding* (Sheu & Chi, 2009), migración (Xing et al., 2005) y fisión (Gulisano et al., 2012; Ishii & Suzumura, 2011; Gedik et al., 2014; Fernandez et al., 2013), si bien existen más, sólo se trataron estas porque se consideran las relevantes y que han sido trabajadas en la literatura al problema.

3.2.1 Planificación determinista

La planificación determinista se centra en los conocimientos *a priori* del sistema, esto significa que se consideran las variables del entorno que se poseen y respecto a esto se toma una decisión de como debe actuar el sistema.

En el área de *Stream processing*, se han realizado diferentes análisis de la estimación de frecuencia de *data stream* en el sistema. Para poder realizar esto, se han considerado modelos matemáticos, tomando ventanas de tiempo de la frecuencia predicha y la real, para posteriormente generar con los datos una función que represente la frecuencia estimada del operador, es decir, el tráfico esperado que llegará al operador en la ejecución del sistema (Ganguly, 2009). Pero no sólo se han considerado modelos matemáticos, sino también algoritmos que determinan la frecuencia del sistema dado el flujo de datos que se podría poseer (Bhuvanagiri et al., 2006).

En otras áreas, como red de sensores, se utiliza esta técnica en el envío de estadísticas de dispositivos móviles, los cuales manejan información *a priori* de donde están los sensores, de tal manera de determinar según la intensidad de la frecuencia, localización o clima, a dónde debe enviar la señal para que se recolecte la información correspondiente (Dong et al., 2007).

Una de las limitaciones de la técnica, es que si bien realiza una predicción determinista de la frecuencia, no necesariamente es correcta a futuro. Esto se debe a qué puede analizarse respecto al promedio, pero pueden surgir procesos anómalos en el transcurso de la ejecución que pueden generar una

sobrecarga en el sistema. Por lo tanto, la estimación al realizarse *a priori*, sólo podrá considerar el inicio del sistema o ventanas de tiempo, por lo que puede existir un porcentaje de error considerable. Por otra parte, se considera que posee mejor rendimiento esta técnica si es que la frecuencia o función analizada es estacionaria, caso que raramente ocurre en el tráfico de datos (Karp et al., 2003).

3.2.2 Load Shedding

En los SPS también se utiliza la técnica de *load shedding*, que consiste en descartar eventos del sistema en caso de existir un comportamiento anómalo, ya sea un máximo en el tamaño de la cola u otro factor. En la Figura 3.1 se puede ver que existe un operador A, el cual recibe datos en un período de tiempo, debido a la cola que existe por parte del sistema, se considera utilizar un operador denominado *Shedding*, que en caso de existir un flujo de datos mayor al umbral propuesto, va a descartar los eventos que excedan el umbral. Por ejemplo, en la transmisión de *video streaming*, al enviar el flujo de información existe un administrador que está analizando el contenido a procesar, por lo que en caso de llegar datos de baja calidad, serán descartados por éste. De esta manera, al existir menor cantidad de datos que procesar, el sistema posee un mejor rendimiento, además de tener una mejor calidad en la visualización de los videos, dado que en su mayoría se procesan datos de alta calidad (Sheu & Chi, 2009).

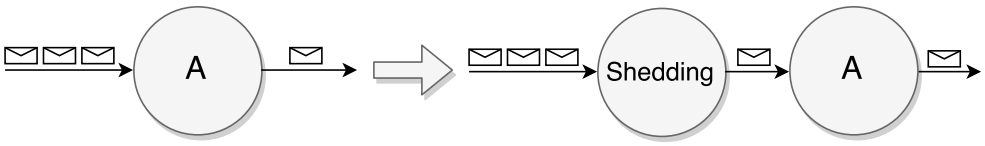

Figura 3.1: Load shedding en un SPS.

En el mundo de los SPS, varios poseen este tipo de estrategia, como por ejemplo S4 (S4, 2014), donde se establece una cota superior de eventos en cola, y en caso que su cola sea igual al límite establecido, los eventos entrantes

serán descartados. Otro sistema que aplica esta técnica es Aurora (Abadi et al., 2003), el cual se basa en procesamiento de datos por ventanas de tiempo, por lo que en caso de existir una ventana de tiempo con una mayor cantidad de eventos de lo estipulado, se descarta el exceso de eventos.

Si bien esta técnica es simple y de bajo costo, siendo pensada para la disminución rápida de carga, existe una baja en la precisión y fiabilidad del procesamiento de los datos. Por ejemplo, en el caso de la transferencia de video no es trascendental, dado que son pocos los pixeles perdidos, pero en una recopilación y análisis de estadísticas, esto dará una menor precisión de los datos procesados por el sistema, dado que puede perderse información que indique comportamientos de los datos estudiados.

3.2.3 Migración

La técnica de migración está basada en el traspaso de un operador de un nodo a otro, según el estado del sistema. En la Figura 3.2 se puede apreciar dos nodos, los cuales poseen tres y dos operadores respectivamente, pero debido a una sobrecarga del nodo 1, se realiza una migración de un operador al nodo 2, ya que este se encuentra con menor carga, de esta manera, se reparten homogéneamente la carga. Si bien no existe alguna implementación que utilice la perspectiva según los recursos lógicos, si existe una que utiliza la perspectiva según los recursos físicos como es el caso de Borealis, el cual fue explicado anteriormente (Xing et al., 2005). Una de las principales críticas que se realiza a esta técnica, es el costo asociado a la transferencia de datos, por lo que existe una menor tolerancia a fallos, debido que al transferir desde una máquina a otra, debe realizarse una comunicación por un medio, el cual puede poseer fallas. Debido a lo anterior, se propone el uso de *buffer* que tengan respaldos de la información, aumentando los costos del sistema (Pittau et al., 2007).

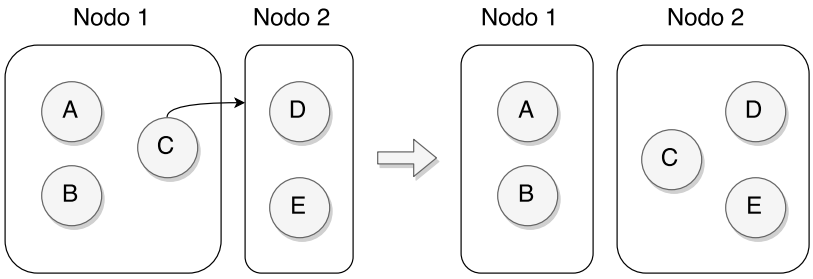

Figura 3.2: Técnica de migración en un SPS.

3.2.4 Fisión

Otra técnica utilizada en el balance de carga es la fisión, o también llamada replicación, particionamiento o paralelismo, la cual consiste en crear una réplica paralela del operador, sin perder el funcionamiento y estado, en caso de existir una sobrecarga en el operador. En Figura 3.3 se presenta un operador A, el cual en primera instancia recibe un flujo de entrada q_1 y un flujo de salida q_2, sin embargo, dado que este operador se sobrecargo, se crea dos operadores extras, los cuales pasarán por una fase de *Split* y posteriormente por una fase de *Merge*. Estas fases son necesarias para distribuir y juntar la información respectivamente. En ciertos SPS se posee el planteamiento que el *split* y el *merge* son operadores que deben ser realizados por el programador, y no de forma automática por el sistema, como S4 o Storm. Una de las características que se posee de esta técnica es que permite generar un sistema elástico, donde se puede aumentar o disminuir la cantidad de operadores según los requerimientos del sistema.

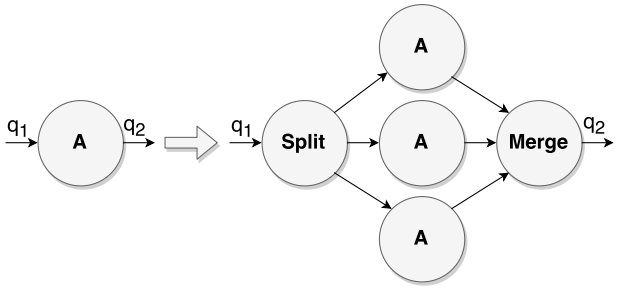

Figura 3.3: Técnica de fisión en un SPS.

Una aplicación que utiliza la técnica de fisión bajo el enfoque estático,

es la paralelización de tareas de Storm (Leibiusky et al., 2012). Aquí se debe indicar en la topología del grafo la cantidad de operadores necesarios para realizar una tarea. De esta manera, por cada tarea se asigna un proceso, el cual tiene a su disposición n hebras según la cantidad de operadores que se desea para cumplir dicha tarea.

Otro sistema que utilizan esta técnica, sin embargo bajo un enfoque dinámico, es *StreamCloud* (Gulisano et al., 2012). Aquí según la cantidad de consultas realizadas al sistema se aumenta o disminuye la cantidad de operadores que cumplen las tareas que se solicitan. Como se había mencionado anteriormente, existe la necesidad de poseer un operador que realice la distribución de los datos, denominado *split*, y la unión de la información al replicar los operadores, denominado *merge*, por lo que este sistema propone sólo la utilización de operadores que realicen funciones específicas, de tal manera que se realice de forma automática el proceso de *split* y *merge*. De esta manera, no existe problemas con los operadores con estado, como lo son los contadores y algoritmos de ordenamiento, dado que automáticamente realiza el procedimiento de separación y unión de los datos. Una de las características principales de este sistema, es aplicar el concepto de elasticidad, que aumenta y disminuye la cantidad de operadores según lo requerido por el sistema. Otros trabajos como (Gedik et al., 2014; Schneider et al., 2009) también aplican este método, y paralelizan las tareas de forma elástica, y con parámetros similares, sólo que su implementación es distinta, debido que este anterior propone utilizar *Cloud* para el uso del SPS, en cambio los otros propones *cluster*.

En (Fernandez et al., 2013), aplican fisión en el caso que exista un cuello de botella en un operador. Para la detección de estas situaciones, se posee un monitor, el cual está consultando en un período de tiempo corto el estado de cada uno de los operadores. De esta manera, se puede ver en cada operador si sobrepaso el umbral de carga propuesto, que en este caso se basa en la utilización de la CPU, se procede a replicar el operador sobrecargado. En la Figura 3.4 se puede ver un sistema, en el cual el operador *u* está enviando un flujo de datos al operador *o*. Una vez *o* se sobrecarga (cuello de botella), este se

debe replicar, hasta reducir la carga hasta un umbral deseado, en otras palabras, llegó a la cantidad de réplicas necesarios, y ya no es necesario replicar más.

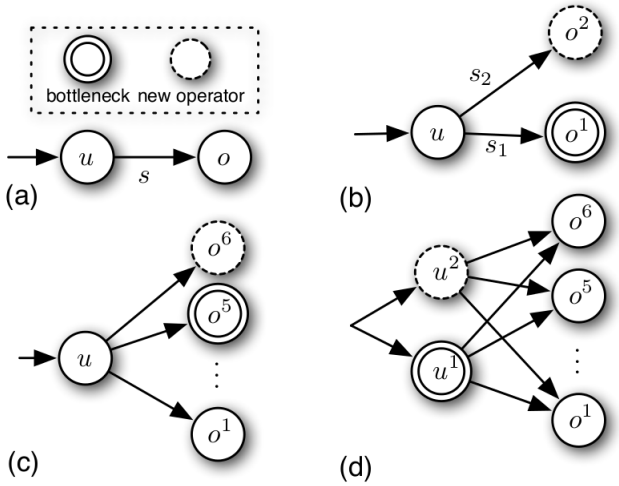

Figura 3.4: Ejemplo de replicación de los operadores (Fernandez et al., 2013).

CAPÍTULO 4. DISEÑO DEL SISTEMA DE DISTRIBUCIÓN DE CARGA

Como se hizo mención en el Capítulo **??**, el problema de la sobrecarga en un SPS está dado por la inflexibilidad existente en el diseño de grafo de procesamiento. Esto quiere decir, que en el momento que el sistema está en ejecución, no existen un mecanismo que permita adaptar los recursos a las necesidades actuales del sistema, por lo que se vuelve ineficiente, pudiendo generar sobrecargas en los operadores que lo componen.

Bajo un escenario como este, es necesario un sistema dinámico, que pueda adaptar su estructura, basándose en su carga y el flujo de datos recibidos . Con estas condiciones, se diseñó un sistema que optimiza el rendimiento, sin comprometer de manera significativa el rendimiento del sistema.

4.1 ANÁLISIS DEL SISTEMA DE DISTRIBUCIÓN DE CARGA

Dentro del análisis realizado en la arquitectura del sistema implementando, se consideró una perspectiva en base a los recursos lógicos según el enfoque dinámico, definido en las subsección 3.1.2 y 3.1.4 respectivamente, para el balance de carga de SPS. El presente trabajo no se enfoca en el análisis del comportamiento de cada uno de los nodos físicos del sistema, sino que más bien en el rendimiento que poseía cada uno de los operadores del grafo diseñado, siendo un problema de carácter lógico y no físico.

Respecto al estudio de las distintas técnicas implementadas, era necesario utilizar una que no tuvieran desventajas en cuanto a la pérdida de datos, inadaptabilidad con el tiempo y costo de implementación. Por lo tanto, se consideró que la mejor opción era utilizar la técnica de fisión, utilizando el mismo modelo de replicación que Fernández (Fernandez et al., 2013), donde basándose en el nivel de carga del operador se genera o no una réplica para este. Dentro de las hipótesis planteadas, se pensaba que el costo de un operador iba a ser menor

a la formación de las colas de datos en el sistema, lo cual podría variar según la arquitectura del SPS implementando.

En la Figura 4.1 se muestra un ejemplo de la replicación propuesta, donde en la parte (a) se presentan tres operadores, en uno de ellos (operador B) existe una sobrecarga, por lo que es necesario replicar el operador. En la parte (b) se presenta el operador ya replicado, pero todavía persiste la sobrecarga en éste, por lo que se vuelve a realizar el mismo procedimiento, hasta que finalmente converge a la cantidad óptima de réplicas deseadas en el sistema en el período de tiempo analizado, como se muestra en la parte (c).

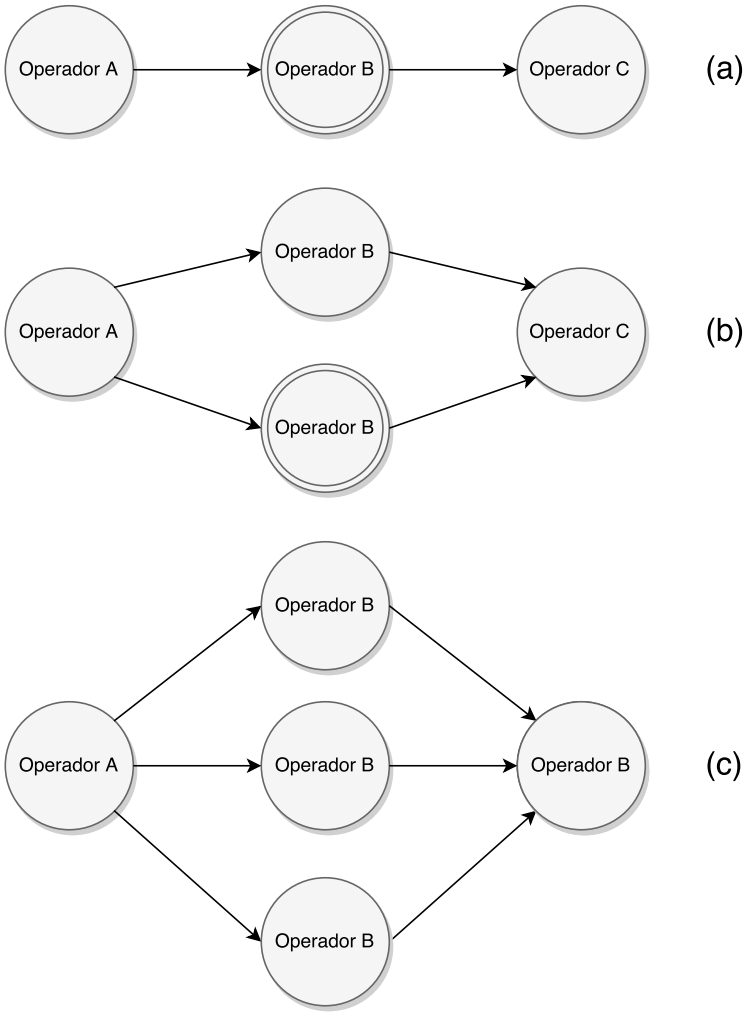

Figura 4.1: Ejemplo de replicación del sistema propuesto.

Para la detección del nivel de carga de un operador es necesario contar con un modelo basado en umbrales que permitan determinar cuando

está o no sobrecargado un operador. Para modelar esta situación se utilizaron conceptos de teoría de colas (Bose, 2013). Dado que los SPS se encuentran en el paradigma orientado a grafos, se puede obtener tanto la tasa de llegada (λ) como la tasa de servicio (μ) de cada uno de los operadores que lo componen, como se ve representando en la Figura 4.2. Aquí la tasa de procesamiento de un operador influye directamente en la tasa de llegada del siguiente operador en el grafo. Al utilizar estos conceptos, se cálculo la tasa de rendimiento (ρ), la cual está definida por la tasa de llegada, de procesamiento y la cantidad de réplicas del operador ($\rho = \frac{\lambda}{\mu\rho}$), cuyo valor representa el factor de utilización del sistema, donde se define un sistema estable si y sólo si $\rho < 1$.

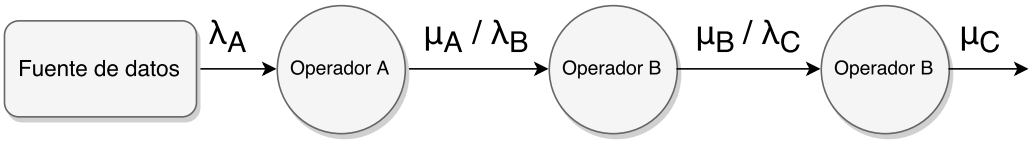

Figura 4.2: Enfoque de un SPS con conceptos de teoría de colas.

Tomando en consideración el tipo de enfoque en el algoritmo de balance de carga y la elasticidad que se pretendía por parte del sistema, es que se trataron tres posibles estados en el sistema: ocioso, estable e inestable. El primer estado corresponde a un exceso en la cantidad de recursos necesarios. El segundo está definido por el rendimiento óptimo del sistema. Y por último, el tercero hace referencia a un sistema sobrecargado, donde es necesario mayor cantidad de recursos por parte de éste. Definido los posibles estados de cada operador, es que se tomó esto como base para el análisis y predicción de la carga en el sistema de distribución de carga.

Para el sistema propuesto, se consideraron dos tipos de algoritmos: predictivo, enfocado en el futuro y la historia del operador, y reactivo, analizando el comportamiento del momento. Esto fue diseñado con el fin de analizar dos factores, los *peak* existentes en la historia del operador, dado el algoritmo predictivo, y otro que analice el comportamiento en el momento, haciendo uso del algoritmo reactivo, de tal manera de solucionar las anomalías que no son detectadas con la predicción.

Es importante denotar que dependiendo del tipo de caso es que un tipo de algoritmo va a funcionar mejor, por ejemplo si se posee una tasa de llegada dado una función exponencial, es necesario aumentar la cantidad de réplicas a medida que van aumentando la cantidad de réplicas en el sistema, por lo que en el predictivo podrá detectar este tipo de casos y aumentar la cantidad réplicas dado este patrón. Pero en el caso que no exista un patrón en la función, existe el algoritmo reactivo que analiza la situación en el momento para ver si es necesario o no cambiar la cantidad réplicas.

Además de lo anterior, se consideró que era necesario trabajar con los dos algoritmos en temporalidades distintas, es decir, en cierto período de tiempo se va a utilizar el reactivo y en otro el predictivo. Por ejemplo, el predictivo necesita n muestras para realizar el cálculo de predicción, por lo tanto, mientras tanto pueden existir k períodos que se realicen mientras tanto análisis en base al algoritmo reactivo. De esta manera, se utiliza el algoritmo reactivo mientras se consiguen la cantidad de muestra representativas para el análisis según el algoritmo predictivo.

Como se diseñaron dos tipos de algoritmos que se complementan, es necesario considerar un algoritmo que administre cual de los dos algoritmos se va a utilizar según el período analizado, como también la cantidad de réplicas que deben crearse o eliminar según el resultado del algoritmo utilizado.

Dado lo anterior, se diseño un sistema de distribución de carga con cuatro componentes: monitor de carga, analizador de carga, predictor de carga y administrador de réplicas, que se pueden apreciar en la Figura 4.3.

Monitor de carga : está encargado de recolectar las estadísticas del sistema, tanto para el algoritmo reactivo, como para el historial del algoritmo predictivo.

Analizador de carga : analiza la cantidad de carga de un operador en un período de tiempo determinado según el algoritmo reactivo, y respecto a esto se indica el estado del operador. Para esto, se consideró la tasa de rendimiento del operador, y según el valor que posea se determinará el estado, el cual podía ser ocioso,

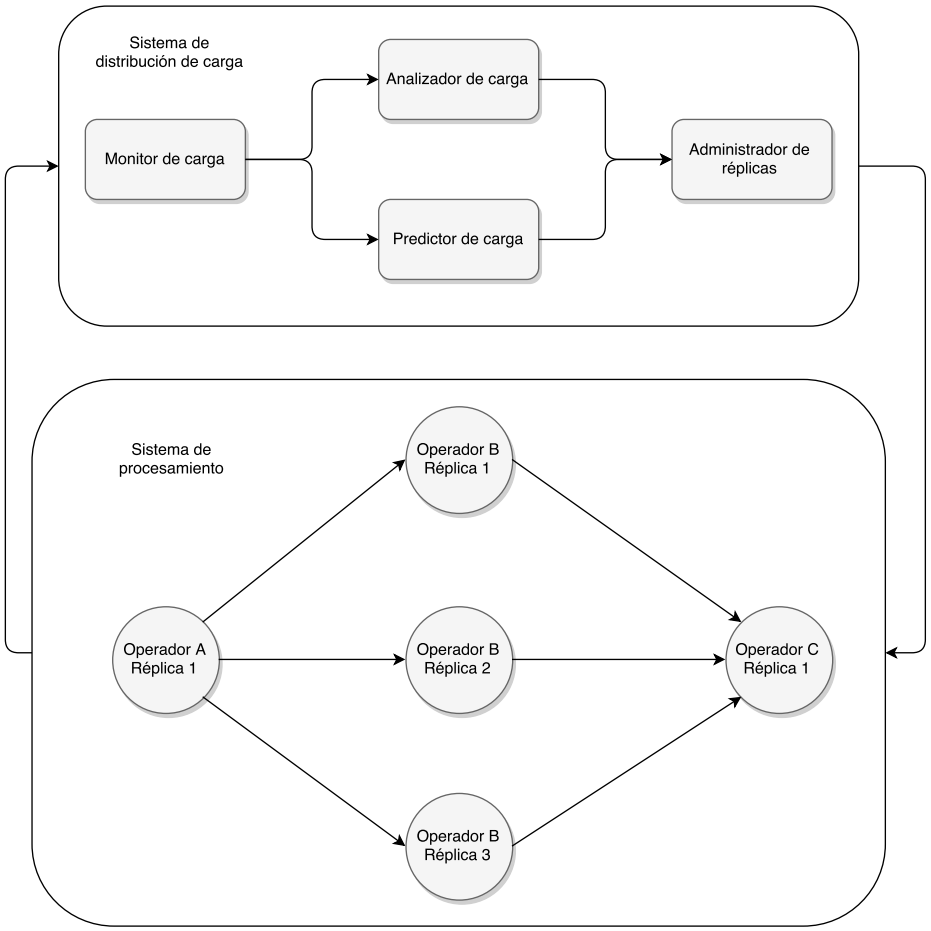

Figura 4.3: Estructura del sistema de distribución de carga.

estable o inestable.

Predictor de carga : es el módulo del algoritmo predictivo, que analiza la historia de un operador en una ventana de tiempo determinada, utilizando como muestra la tasa de rendimiento del operador, para posteriormente realizar una cadena de Markov según los posibles estados del sistema. Posteriormente, para la predicción de la carga del operador, se calculó la distribución estacionaria (Papoulis, 1984), el cual va a entregar la probabilidad que el operador se encuentre en cada uno de los posibles estados.

Administrador de réplicas : se encarga de determinar cual es el algoritmo a utilizar en cada período de tiempo, ya sea reactivo o predictivo, y la administración de las réplicas utilizando como entrada la información prevista por el analizador y

predictor de carga.

4.2 RECOLECCIÓN DE LOS DATOS

Como se había mencionado anteriormente, el monitor de carga está encargado de recolectar los datos necesarios para el funcionamiento del sistema de distribución de carga, tanto el historial para el algoritmo predictivo, como la tasa de rendimiento para el algoritmo reactivo. Para esto se consideraron ventanas de tiempo de un segundo para la recolección de muestras para el análisis predictivo, y cinco segundos para el análisis reactivo.

La recolección de muestras para el algoritmo predictivo, considera ventanas de tiempo de un segundo debido que se consideraba un período estable para determinar el comportamiento del operador. Con esto, se posee una muestra de cien datos cada vez que se realice el algoritmo predictivo, debido que cada cien segundo está planificado la ejecución del algoritmo. La cantidad de muestras fue determinado según la literatura, debido que se consideraba un número apropiado para realizar una predicción del operador (Ching & Ng, 2006), de tal manera de no existir una deficiencia en la cantidad de muestras para la predicción deseada.

La recolección de muestras para el algoritmo predictivo se realiza en ventanas de tiempo de un segundo, dado que cada un segundo se considera que es un tiempo suficiente para obtener una muestra representativa del operador. Con esto, se poseen 100 muestras cada vez que se realice el algoritmo predictivo, debido que cada 100 segundos está planificado la ejecución del algoritmo. La cantidad de muestras fue determinado según la literatura, debido que se consideraba un número apropiado para realizar una predicción del operador (Ching & Ng, 2006), de tal manera de no existir una deficiencia en la cantidad de muestras para la predicción deseada.

Por otra parte, para la obtención de muestras para el algoritmo reactivo, se consideraron muestras obtenidas en períodos de cinco segundos.

La muestra está compuesto por la tasa de rendimiento del operador en ese período, la cual es utilizada por el algoritmo reactivo para determinar el estado del operador según los umbrales propuestos. Dentro de las consideraciones realizadas para la recolección de datos para el algoritmo reactivo, fue considerar que la tasa de servicio (μ) es homogénea con el transcurso del tiempo, considerando que los datos procesados son homogéneos, por lo tanto, se considera el valor promedio y ese es considerado en el procesamiento de los datos.

Cabe destacar que cuando el algoritmo predictivo se ejecuta, no es necesario la recolección de los datos del período, debido que el algoritmo reactivo no se ejecuta en el mismo período que el predictivo. Sólo la recolección del historial es realizada en todo momento, dado que estás son guardadas para posteriormente ser analizadas por el algoritmo predictivo.

4.3 ALGORITMO REACTIVO

El diseño del algoritmo reactivo se basó en el tipo de análisis del estado del operador en un período determinado, siendo definido su estado por una variable del operador, el cual dependerá del rango en que se encuentre dado los umbrales que se poseen. En este diseño se analizó según la tasa de rendimiento (ρ), donde el estado del operador dependerá del valor que éste posea según los umbrales establecidos.

En el Algoritmo 4.1 se puede ver el análisis del estado de un operador según su tasa de rendimiento; que en el caso que sea mayor a 1, su estado es inestable, menor a 0.5, significa que está en estado ocioso, y en otro caso, significa que está estable. Estos datos posteriormente serán considerados por al administrador de réplicas, el cual analiza el comportamiento que debe tener el sistema según lo indicado por el algoritmo.

En la Figura 4.4 se puede analizar el estado del operador según la tasa de procesamiento de forma visual. En los primeros 90 segundos la tasa del

Algoritmo 4.1: Algoritmo reactivo del sistema de distribución de carga.

Entrada: Tasa de procesamiento ρ del operador ϕ.
Salida: Estado del operador, donde -1 significa estado ocioso, 0 estable y 1 inestable.

```
1: if ρ_φ > 1 then
2:     return 1
3: else if ρ_φ < 0.5 then
4:     return -1
5: else
6:     return 0
7: end if
```

operador es mayor al límite superior, lo cual indica que el sistema es inestable, es decir, el operador posee sobrecarga. Posteriormente, en el segundo 100 segundo, la tasa de rendimiento empieza a disminuir, ya sea por una optimización sobre los recursos lógicos o una disminución de la tasa de llegada, por lo que ahora el operador ya no se encuentra sobrecargado (inestable), sino se encuentra entre el límite inferior y superior, cuyo rango define al operador como un sistema estable.

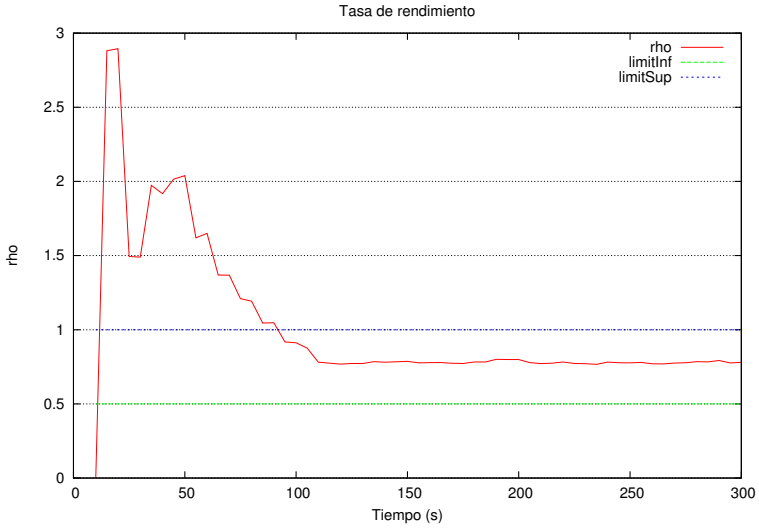

Figura 4.4: Comportamiento de la tasa de procesamiento de un operador.

4.4 ALGORITMO PREDICTIVO

Para la confección del algoritmo predictivo se realizó un análisis según las cadenas de Markov (Ching & Ng, 2006), por lo que se tuvieron que seguir las siguientes etapas:

- Definir muestras en tiempos discretos, las cuales cambiaran con el tiempo según un proceso estocástico. Las muestras se definieron como la tasa de procesamiento (λ) del operador, la cual dependiendo del valor que poseyera, iba a otorgar un estado al operador.
- Determinar los estados finitos que se van a utilizar para la conformación de la cadena, que en este caso sería los estados que puede encontrar el operador: ocioso, estable o inestable.
- Una cantidad representativa de muestras para la construcción de la cadena de Markov en el período analizado. Estas muestras serán independientes de un período y otro, por lo que los valores de la cadena de Markov irán cambiando en cada período de tiempo. Para la implementanción del algoritmo, se consideraron cien muestras por cada período, cuyos intervalos eran de cien segundos, tomando según lo analizado del estado del arte (Gong et al., 2010).

Tomando las bases anteriores, se diseño una cadena de Markov en base a tres posibles estados: ocioso, estable e inestable, como se demuestra en la Figura 4.5. Cada uno de los estados posee una probabilidad de transición hacia algún estado, cuyas probabilidades están definidas por las muestras obtenidas en el período de tiempo analizado.

Por lo tanto, para cada operador se construirá una cadena de Markov según el historial obtenido en la ventana de tiempo. Para la conformación de la cadena de Markov se consideraron las muestras de la historia, por lo que la transición de una muestra a otra presentaba un transición, las cuales dieron origen a la matriz de transición. En el Anexo A se puede ver el algoritmo que se

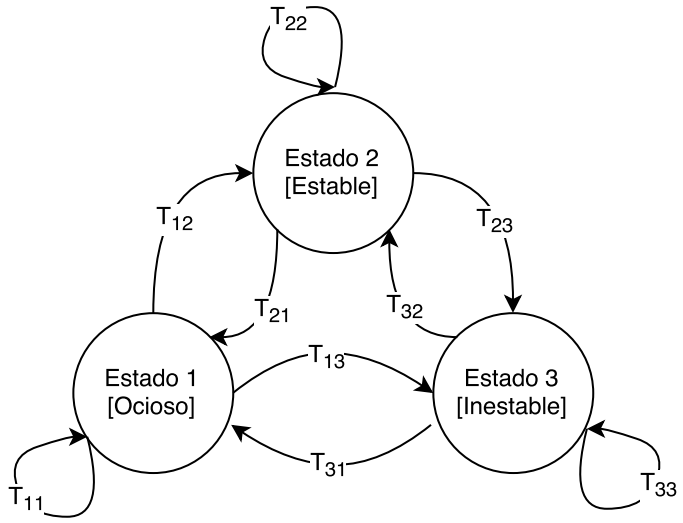

Figura 4.5: Cadena de Markov dado el modelo propuesto del sistema.

empleó para construir la matriz de transición. En la ecuación 4.1 se muestra la matriz de transición que se obtiene de la cadena de Markov de la Figura 4.5.

$$P = \begin{bmatrix} T_{1,1} & T_{1,2} & T_{1,3} \\ T_{2,1} & T_{2,2} & T_{2,3} \\ T_{3,1} & T_{3,2} & T_{3,3} \end{bmatrix} \tag{4.1}$$

Obtenida la matriz de transición se puede calcular la distribución estacionaria de la cadena de Marjov, la cual indica las probabilidades que en el futuro se encuentra el operador esté en cada uno de los posibles estados, ya sea ocioso, estable o inestable. Para el cálculo de esto, se utiliza la ecuación de Chapman-Kolmogórov (Papoulis, 1984) descrita en la subsección 2.5.1.

El Algoritmo 4.2 describe el cálculo de la distribución estacionaria, cuya entrada es la matriz de transición de un operador del SPS.

Antes de realizar el cálculo, es importante analizar si efectivamente existen transiciones en todos los estados, debido a que existía la posibilidad que no hubiera alguna transición a un estado en particular en un período de tiempo dado. Por ejemplo, podría ser que en cierto período nunca se ha encontrado ocioso el sistema, pero si estable e inestable. Como el cálculo de la distribución estacionaria requiere un estado de inicio, se verificó si efectivamente existía o no

el estado, y en caso no existir, el estado de inicio será alguno existente.

La cantidad de iteraciones que debe realizarse para el cálculo correspondiente, se proporcionó como entrada del algoritmo según lo que se estimaba necesario. Es importante destacar que entre mayor cantidad de iteraciones, mayor precisión en el valor de la predicción, pero esto implica a su vez un mayor tiempo de cómputo. Debido a esto, es que se trató de considerar un punto medio, de tal manera de tener un bajo margen de error, pero con bajo costo en el tiempo de ejecución. La cantidad de iteraciones escogida para la implementanción fue de 100.000.

Algoritmo 4.2: Cálculo de la distribución estacionaria de la cadena de Markov de un operador ϕ.

Entrada: Γ Matriz de transición del operador ϕ, υ cantidad de iteraciones deseadas y última muestra m de las muestras de ρ
Salida: Δ Distribución estacionaria de la cadena de Markov del operador ϕ.

```
1: i ← 0 //Estado inicial de iteración
2: if ρ_m < 0.5 then
3:    i ← 0
4: else if 0.5 ⩽ ρ_m ⩽ 0.5 then
5:    i ← 1
6: else
7:    i ← 2
8: end if
9: τ ← Arreglo[3] //Contador para la normalización de los datos
10: for k = 0 a υ do
11:    u = randomUniform(0, 1)
12:    σ = 0
13:    for j = 0 a 3 do
14:       σ = σ + Γ_{i,j}
15:       if u ⩽ σ then
16:          τ_j + +
17:          i ← j
18:          break
19:       end if
20:    end for
21: end for
22: Δ ← Arreglo[3] //Distribución estacionaria de la cadena de Markov del operador φ
23: for k = 0 a 3 do
24:    Δ_k ← τ_k/υ
25: end for
26: return Δ
```

Obtenida la distribución estacionaria, se procede a analizar las probabilidades obtenidas y como influye al operador. Para esto, se consideró que las probabilidades tuvieron entre ellas una desviación estándar superior a 0.25,

debido que si esto se cumple, la probabilidad mayor de la distribución estacionaria no posee incertidumbre (Soong, 2004), porque en caso que no supere la desviación estándar, puede ser que dos probabilidades sea muy parecidas y la probabilidad no sea un comportamiento determinante. En el Algoritmo 4.3 se describe el análisis que se realiza a la distribución estacionaria, siendo en primer lugar el análisis estadístico de las probabilidades, y segundo la obtención del estado con mayor valor de las probabilidades, retornando finalmente el estado del operador. Cabe destacar que el primer estado se consideró ocioso, el segundo estable, y el tercero inestable.

Algoritmo 4.3: Algoritmo predictivo del sistema de distribución de carga.

Entrada: Δ Distribución estacionaria de la cadena de Markov del operador ϕ.
Salida: Estado a futuro del operador, donde -1 significa estado ocioso, 0 estable y 1 inestable.

```
if σ(Δ1, Δ2, Δ3) > 0.25 //Desviación estándar de las probabilidades de la distribución estacionaria then
    i ← getStateMax(Δ) //Obtención del estado con mayor probabilidad
    if i = 1 then
        return -1
    else if i = 2 then
        return 0
    else
        return 1
    end if
end if
return 0
```

4.5 ADMINISTRACIÓN DEL SISTEMA

El último componente del sistema es el administrador de réplicas, cuya función es administrar la cantidad de réplicas en cada uno de los operadores según los recursos disponibles por parte del sistema y según el estado que adopte un operador, ya sea a futuro o en el momento.

Para esto, se diseño un administrador que ejecuta un algoritmo (reactivo o predictivo) según el período del ciclo que se encuentre el sistema. Cada ciclo posee 20 períodos, donde los primeros 19 corresponde al algoritmo reactivo y el último corresponde al algoritmo predictivo. Cada período posee un

intervalo de 5 segundos, de esta manera, cada ciclo tendrá un intervalo de 100 segundos, la cantidad necesaria para obtener las muestras para el algoritmo reactivo, suponiendo que cada muestra es obtenida en 1 segundo.

En el Algoritmo 4.4 está el procedimiento de administración, donde primero analiza que tipo de algoritmo debe ejecutar según el período del ciclo. En caso de realizarse el reactivo, se analiza si existen dos alertas consecutivas del mismo estado del operador, ya sea ocioso o inestable, y de ser así, realizar una modificación a la cantidad de réplicas del operador. Además, se cambió el estado del operador a estable, de tal manera de no considerar ese período para el próximo análisis reactivo, esto se consideró para dejar un margen de descanso para el algoritmo reactivo. Por otra parte, de ejecutarse el módulo predictivo, se analiza cual fue la predicción, por lo que si es ocioso disminuirá la cantidad de réplicas y si es inestable las aumentará. Como el proceso de predicción se realiza con menor frecuencia y posee mayor cantidad de cómputo, se considero que debía crear o remover mayor cantidad de réplicas que en el módulo reactivo, aprovechando así el análisis de la historia del operador.

Dentro de las consideraciones que se tuvieron para el diseño del administrador, fue la cantidad máxima de réplicas que podían realizarse. Dado que una de las limitantes de este trabajo fue que sólo se utilizó una máquina, la cantidad de recursos son limitados, por lo que aumentar la cantidad de réplicas indefinidamente iba a generar una sobrecarga en los recursos disponibles por parte de la máquina, habiendo fallas en el funcionamiento del SPS.

Algoritmo 4.4: Administración de réplicas de un operador ϕ dado su comportamiento en el sistema de distribución de carga.

Entrada: Operador ϕ a analizar y ι período en que se encuentra el sistema de distribución de carga.
Salida: Cantidad de réplicas a modificar del operador.

```
1: if ι mód 20 ≠ 0 then
2:    δ_ι ← AlgoritmoReactivo(φ)
3:    if δ_ι and δ_{ι−1} son estado inestable then
4:       if No excede la cantidad máxima de réplicas en el sistema then
5:          δ_ι ← estado estable
6:          return Crear una réplica del operador φ
7:       end if
8:    else if δ_ι and δ_{ι−1} son estado ocioso then
9:       return Remover una réplica del operador φ
10:   end if
11: else
12:   δ_ι ← AlgoritmoPredictivo(φ)
13:   if δ_ι es estado inestable then
14:      if No excede la cantidad máxima de réplicas en el sistema then
15:         return Crear cinco réplicas del operador φ
16:      end if
17:   else if δ_ι es estado ocioso then
18:      return Remover cinco réplicas del operador φ
19:   end if
20: end if
21: return No hacer nada al operador φ
```

CAPÍTULO 5. EXPERIMENTOS Y EVALUACIÓN

5.1 IMPLEMENTACIÓN DEL SISTEMA

Para la implementanción del sistema propuesto, se utilizó como base el motor de procesamiento de *stream* S4 (S4, 2014), cuyo modelo fue explicado en la sección 2.3. El funcionamiento del sistema de distribución de carga propuesto se desarrolló en Java, donde se modificó en el código fuente de S4, para incluir las funciones de este producto, de tal manera que fuera automático y transparente para el usuario del SPS.

El sistema diseñado fue añadido al proyecto S4, siendo una paquete denominado *monitor* que contenía las clases *S4Monitor*, *MarkovChain*, *MonitorMetrics*, *StatusPE* y *TopologyApp*. La primer clase correspondía a las funcionalidades del sistema de monitoreo, ya sea la recepción de los datos del sistema, ejecución del algoritmo reactivo o predictivo y administración de las réplicas de cada operador. La segunda clase hacía alusión al modelo de cadena de Markov implementando por el algoritmo predictivo, donde posee tanto la conformación de la matriz de transición como el cálculo de la distribución estacionaria. La tercera clase se utilizó para recolectar las estadísticas de cada operador, y poder analizar los experimentos. Finalmente, la cuarta y quinta clase fueron utilizadas para la implementación del sistema de monitoreo, siendo la primera el estado de cada PE y el segundo la topología que poseía el grafo creado por el usuario, los cuales pueden verse con más detalle en el Anexo B.

Para la ejecución del SPS conjunto con el monitor, se debe en primer lugar registrar los PEs. Para esto, se implementó una función que tenga como parámetro dos PEs, donde el primero es el PE emisor y el segundo el receptor. Esta información es importante para obtener la topología del grafo, dado que con esto se puede realizar un análisis del rendimiento de los PEs.

Además de lo anterior, para el análisis de cada uno de los PEs, se añadió los atributos de cantidad de eventos entrantes y salientes cada un

segundo y cinco segundos en la clase del PE. Esto fue utilizado posteriormente para las estadísticas deseadas para el algoritmo reactivo y predictivo. La tasa de llegada es considerada desde el momento que se recibe el evento desde el PE emisor, y la tasa de servicio es considera desde el momento que se termina de realizar la ejecución de la tarea en el PE. Además, se agregó un atributo booleano, el cual indica si el PE puede aumentar o disminuir la cantidad de réplicas que posee al iniciar el sistema.

Posteriormente, para el funcionamiento del sistema de distribución de carga se procedió a ejecutar dos tareas en la aplicación de S4: una tarea que guarde las muestras para el historial del operador y otra que envíe las estadísticas del operador, las cuales poseían un intervalo de ejecución de 1 y 5 segundos respectivamente. La primera tarea analiza cada uno de los PEs, tomando en consideración las estadísticas de eventos entrantes y salientes, los cuales darán origen a la tasa de llegada (λ) y servicio (μ), para que posteriormente se calcule la tasa de rendimiento en la ventana de tiempo analizada, guardando así la muestra en el historial. La segunda tarea obtiene las estadísticas de cada uno de los PEs, ya sea la tasa de llegada (λ) y servicio (μ), dado los eventos entrantes y saliente, cantidad de eventos totales e historial del PE, los cuales son enviados al monitoreo de carga, para posteriormente ejecutar el algoritmo que corresponda según la ventana de tiempo, y finalmente realizar la administración de réplicas para aumentar o disminuir las réplicas necesarias en cada operador según lo necesario. El código de la implementanción se puede observar en el Anexo C.

Debido al funcionamiento de los SPS, existe una fuente de datos, por lo tanto, para la correcta sincronización de las estadísticas, se procedió a realizar una espera por parte del monitor hasta que la fuente de datos esté lista para enviar los datos. Esta implementanción se puede ver en el Anexo C, conjunto con las dos tareas que debe ejecutar después que la fuente de datos avisa de su inicialización.

En el caso del envío de las estadísticas, después de ser enviadas, se consulta el estado de cada uno de los operadores, y en caso de ser necesario, se realiza las modificaciones necesarias en el sistema, ya sea de crear o

eliminar réplicas de un operador. En el caso de crear réplicas, lo que se realiza es consultar la cantidad de réplicas que dijo que deben existir por parte del administrador de réplicas, y en caso de poseer un número de réplicas mayor que el existente por el PE, se procede a crear la cantidad de réplicas faltantes. Por otra parte, en caso de que el número sea menor, procede a terminar la ejecución del PE y posteriormente elimina la cantidad de réplicas sobrantes. En caso que el operador que se haya identificado con el atributo único, significa que no puede aumentar ni disminuir la cantidad de réplicas asignadas a éste. Los códigos de crear o eliminar replicas se encuentra adjuntos en el Anexo C.

Dado que en el diseño se propuso un mecanismo de balance de carga basado en la técnica de replicación, fue necesario modificar S4, de manera de poder manejar las réplicas para los operadores. Se implementó el SPS de tal manera de poder manejar un algoritmo que escogiera la réplica que tuviera menor tamaño en la cola de espera al momento de enviar un dato, y así siempre escoger al operador con menor carga, y en caso que posean el mismo tamaño, utilizar la primera réplica disponible.

En la Figura 5.1 se explica gráficamente la distribución de la carga. En la parte (a) el operador A envía un evento al operador B, el cual posee tres réplicas, cuyas menores colas están en la réplica 2 y 3, como son iguales las colas de estas réplicas, se escoge la primera réplica disponible, es decir, la réplica 2. Posteriormente, como la réplica 2 aumento su cola, la réplica 3 es la que posee menor cantidad de cola, como se demuestra en la parte (b), por lo tanto, es la réplica candidata a recibir el dato enviado. Y finalmente, en la parte (c) todas las réplicas posee el mismo tamaño de la cola, por lo tanto, se procede a enviar a la primera réplica.

En el Algoritmo 5.1 está descrito la distribución de carga planteada anteriormente, la cual fue implementada en S4 para realizar los experimentos según lo diseñado en el planteamiento de los algoritmos.

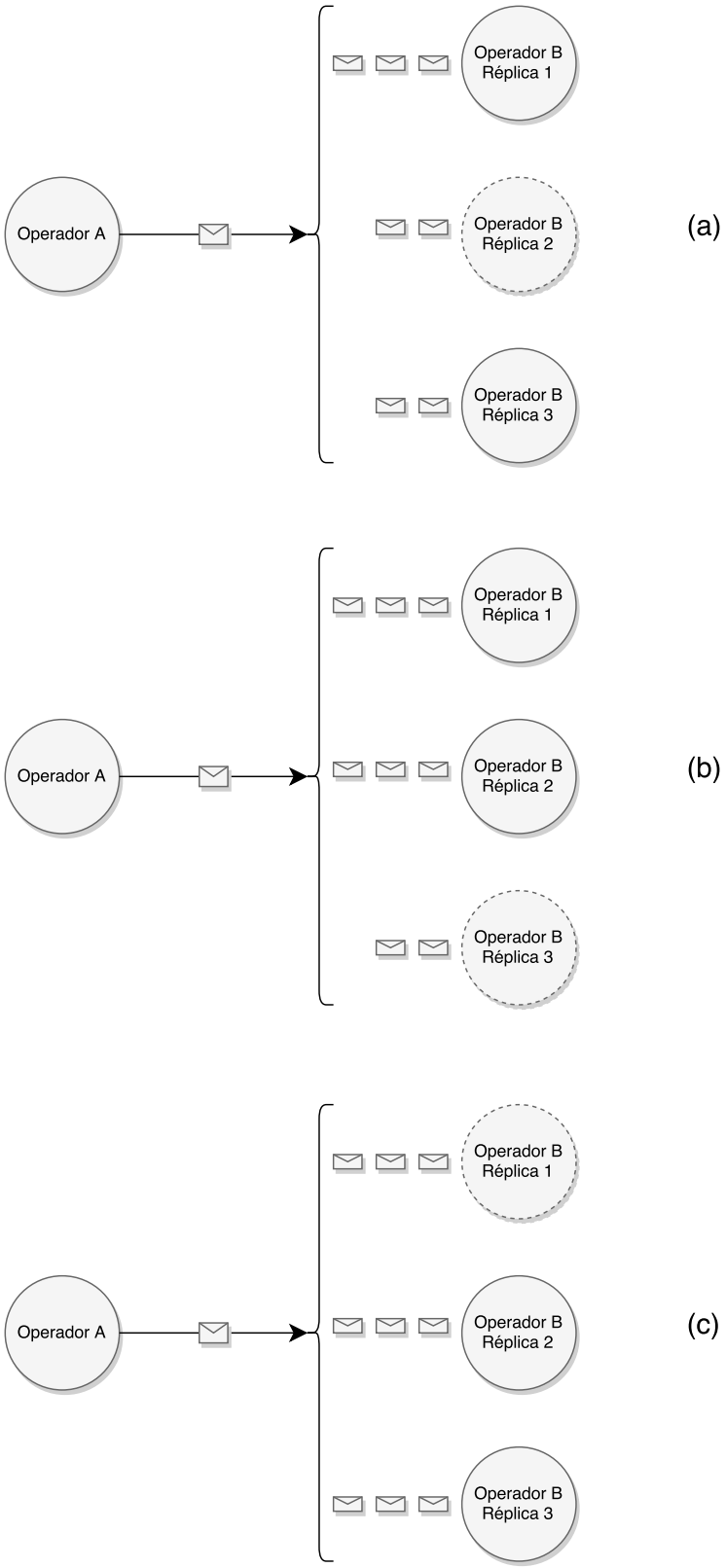

Figura 5.1: Distribución de la carga entre las réplicas.

Algoritmo 5.1: Distribución de carga entre las réplicas de un operador.

Entrada: Evento ϵ y operador ϕ.
Salida: Envío del evento a la réplica disponible del operador ϕ.
1: $\theta \leftarrow minTamanoCola(\phi)$ //Se escoge la réplica que posea menor cola
2: $envioEvento(\epsilon, \theta)$

5.2 DISEÑO DE LOS EXPERIMENTOS

Para los experimentos se diseñaron tres aplicaciones, una que realiza operaciones con estado, otra que no, y otra que es sintética. En el caso que la aplicación posea estado, significa que el operador guarda variables con el transcurso del tiempo, las cuales van a ser entregadas cada cierto tiempo o al finalizar la ejecución del sistema. Un ejemplo de esto, es un sistema que cuenta las palabras de un texto y que envié la cantidad de palabras contadas cada cierta ventana de tiempo. Por otra parte, la aplicación sintética se refiere a un sistema cuyos operadores sólo generan tiempo de demora artificial, en el caso de este experimento, se dejo durmiendo la hebra asignada al operador un período determinado de tiempo.

Para la generación del *stream* de la fuente de datos para la primera y segunda aplicación, se utilizaron datos de muestra utilizados fueron *tweets* recolectados entre los días 27 y 28 de Febrero y 1 y 2 de Marzo de 2010, tanto en inglés, portugués y español, cuya información correspondía a la interacción entre los usuarios debido al terremoto ocurrido el 27 de Febrero en el territorio Chile.

Por otra parte, el envío de los datos anteriores se realizó de dos maneras: constante y variable. La primera forma consiste en enviar 100 eventos por segundo constantemente. En cambio, la segunda forma consiste en enviar 50 eventos por segundos el primer tercio del experimento, para luego aumentar a 75 eventos por segundo, los primeros 120 segundos del segundo tercio, y seguir aumentando a 100 eventos por segundo, para posteriormente volver a 75 eventos por segundos los últimos 120 segundos del segundo tercio, y finalmente, disminuir a 50 eventos por segundos el último tercio de la ejecución.

5.2.1 Análisis de *tweets* en escenarios de desastres naturales

La primera aplicación fue orientada a un caso de desastres naturales, donde se generó un grafo que pudiera realizar un filtrado de palabras y análisis de los datos. Ninguno de los operadores posee estado, por lo tanto son independientes. Para la duración de esta prueba se consideró un tiempo de 70 minutos.

La aplicación consta del flujo de datos, cuyos datos serán la muestra de *tweets* de prueba, y cuatro operadores, los cual son denominados *Stopword*, *Language*, *Counter* y *MongoDB*.

Stopword : está encargado de analizar el *tweet* y remover las palabras no relevantes para el análisis de éste, basado en una bolsa de palabras, cuyas palabras se llaman *stopwords*. De esta manera, se puede analizar el texto usando las palabras más representativas de éste y así entregar información más precisa.

Language : está encargado de analizar el lenguaje existente en el *tweet*, para esto se utilizó una librería *Apache Tika* (Mattmann & Zitting, 2011). Con ésta se puede realizar un filtro del idioma de los tweets, de tal manera que en caso de ser requerido, sólo continúen los de un idioma en específico, en nuestro caso el español.

Counter : está encargado de contar e indicar la cantidad de palabras que existen en el *tweet* según una bolsa de palabras proporcionada por el programador. Para esta prueba se utilizó una bolsa de 26.000 palabras en español. Con esto se podría realizar un análisis de los *tweets* que poseían mayor cantidad de palabras claves asociados a una temática o evento en particular.

MongoDB : está encargado de guardar en la base de datos el evento según los atributos que éste posea, ya sea por el *tweet* original, sin *stopword*, idioma y

cantidad de palabras claves existentes en él. Para esto, se utilizó el motor de base de datos no relacional *MongoDB* (Chodorow, 2013).

En la Figura 5.2 se muestra un ejemplo de la aplicación con sus distintos operadores y relaciones. Las flechas muestran la dirección del flujo de datos emitido por la fuente y los operadores.

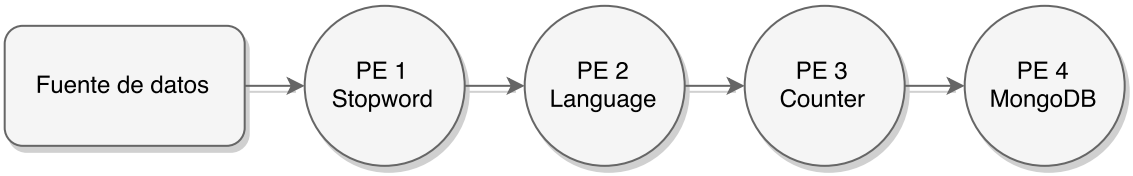

Figura 5.2: Primera aplicación de prueba.

5.2.2 Contador de palabras en muestras de textos

La segunda aplicación consiste en un contador de palabras, el cual cuenta la cantidad de veces que se repite una palabra en un conjunto de datos según una bolsa de palabras establecida por el usuario. Esta aplicación se considera con estado, debido que debe existir un contador en el operador, de tal manera de contar la cantidad de veces que se repite cierta palabra de la bolsa de palabras en los datos entrantes. Con esta aplicación es posible analizar posteriormente las palabras más frecuentes emitidas por los usuarios de la red social según un listado de palabras claves. Para la duración de esta prueba se consideró un tiempo de 70 minutos.

La aplicación consta del flujo de datos, cuyos datos serán la muestra de *tweets* de prueba, y tres operadores, denominados *Split*, *Counter* y *Merge*.

Split : está encargado de dividir el *tweet*, y enviar un arreglo con las palabras que poseía este texto al operador *Counter*.

Counter : está encargado de guardar las estadísticas de los contadores de cada palabra, es decir, en el momento que reciba un evento, este analiza las palabras

que posee y aumenta el contar de la palabra en el operador. Las estadísticas son enviadas cada 10 segundos al operador Merge, de tal manera de no enviar flujo constante al siguiente operador y generar mayor cantidad de carga.

Merge : está encargado de unir las distintas estadísticas enviadas por las distintas réplicas del operador Counter.

En la Figura 5.3 se muestra un ejemplo de la segunda aplicación con sus distintos operadores y sus relaciones, de igual manera que en el caso anterior, las flechas reflejan el flujo de datos. Cabe destacar que el único operador que puede replicarse es el *Counter*, debido que el operador *Split* y *Merge* son operadores que soportan la replicación del operador *Counter*, como se explicó en las técnicas de balance de carga en la subsección 3.2.4.

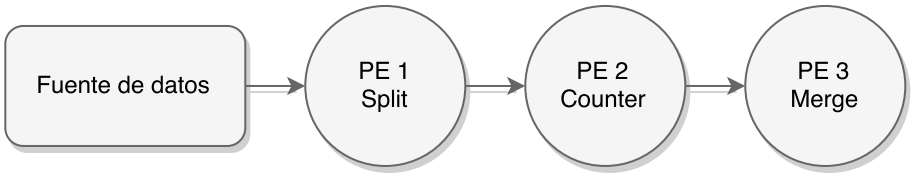

Figura 5.3: Segunda aplicación de prueba.

5.2.3 Aplicación sintética

La tercera aplicación consta de tres operadores, los cuales duermen a la hebra asignada al operador un determinado período de tiempo. De esta manera, se genera un tiempo de espera artificial, realizando un operador sintético, de tal manera que pueda simular el comportamiento de un operador real.

La tabla 5.1 muestra el período de tiempo que duerme cada PEs a su hebra asignada. Los tiempos que se consideraron para esta prueba es para generar una sobrecarga en el primer y segundo operador, de tal manera que después afectarán al tercer operador. Para la duración de esta prueba se consideró un tiempo de 15 minutos.

PE	Tiempo (ms)
1	20
2	30
3	15

Tabla 5.1: Período de tiempo que duerme la hebra asignada al PE.

En la Figura 5.4 se muestra un ejemplo de la tercera aplicación con sus distintos operadores y sus relaciones, de igual manera que en los casos anteriores, las flechas reflejan el flujo de datos.

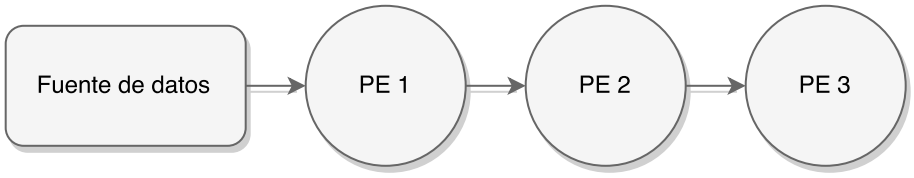

Figura 5.4: Tercera aplicación de prueba.

5.3 EVALUACIÓN

Para la ejecución de los experimentos, se utilizó un servidor con sistema operativo Ubuntu 14.04.2 LTS, cuyo procesador es un Intel®Xeon®CPU E5-2650 v2 de 2.60GHz y con 32 GB de RAM. Cabe recalcar, que el lenguaje de programación fue Java, debido a la integración del sistema propuesto en el SPS utilizado, el cual fue S4. La configuración de la configuración de S4 está detallada en el Anexo D.

La recolección de estadísticas fue realizada cada 5 segundos, por lo que desde ahora se hablará período un intervalo de tiempo de 5 segundos entre cada recolección de las estadísticas.

5.3.1 Primera aplicación

En la primera aplicación se procedió a realizar cuatro experimentos distintos, donde cada par era una comparación del experimento con y sin uso del monitor. Las dos primeras se consideró un envío constante de la fuente de datos, y las dos segundas se consideró un envío dinámico de la fuente de datos.

Para el análisis de los experimento se consideró la cantidad promedio de eventos procesados en un período, la cantidad total de eventos procesados y las estadísticas de cada PE en el transcurso de la ejecución de la aplicación.

Para analizar el comportamiento del sistema en el primer experimento, se procedió a estudiar las estadísticas de cada uno de los PEs. En la Figura 5.25 y 5.26 las estadísticas del primer operador del grafo, con y sin monitoreo en la carga de los operadores respectivamente. La tasa de llegada (λ) se puede observar como en el primer gráfico es constante, pero no así en el segundo gráfico, debido que en el segundo 2600 surge una disminución del flujo de datos. Esto se debe a la acumulación de eventos en el *buffer*, por lo que se genera cola en el PE, lo cual impide almacenar mayor cantidad de eventos, dado que la cantidad de eventos que se procesan es menor que la cantidad de eventos que llegan. Por lo tanto, al llenarse el *buffer*, no puede seguir almacenando eventos en la cola, siendo éste bloqueado.

Por otra parte, la tasa de rendimiento de la Figura 5.25 se estabiliza dentro de los primeros 100 segundos, debido que el sistema de distribución de carga detectó una sobrecarga en el operador y replicó el operador, a diferencia de la Figura 5.26, en el cual el operador posee una tasa de rendimiento estable que bordea entre 1 y 2, hasta el segundo 2600, donde disminuye debido que la tasa de llegada es menor. Cabe destacar que en este operador se encontró una sobrecarga, debido que posee un alto costo computacional, debido que existe una gran cantidad de palabras que debe analizar con cada una de las palabras del texto, por lo que se hace necesario otra réplica en el sistema.

En la Figura 5.27 y 5.28 se puede ver el siguiente operador del gráfico, el cual no tiene mayor inconveniente, a excepción del segundo 2400, donde el

sin monitoreo disminuye considerablemente la tasa de procesamiento del PE. Si analizamos la Figura 5.26 y 5.28, empiezan aproximadamente desde el rango de tiempo (2400s,2600s) los problemas de procesamiento de los operadores, por lo tanto, se puede deducir que si el problema surge en un operador no es un problema aislado, sino que también influye a los siguientes operadores.

Se puede observar como con el transcurso de los primeros 100 segundos en la Figura 5.29, fue aumentando la cantidad de réplicas hasta llegar a 5, el cual fue su óptimo, para ir procesando la cantidad de eventos y disminuir la cola existente en el sistema. Esto en contra posición a la Figura 5.30, donde la inexistencia de replicación, genera una cola la cual se mantiene constante en el segundo 2400.

Finalmente, se encuentra el último operador, el cual no presenta grandes inconvenientes tanto en la Figura 5.31 y 5.32, esto debido que el tiempo de procesamiento es bajo, y nunca llega una cantidad de eventos considerable para existir una sobrecarga en éste. Además es importante destacar que en la Figura 5.32 llega una menor tasa de llegada, exactamente un $83,827\,\%$ menos de eventos que un SPS ejecutado con el sistema de distribución de carga.

Por otra parte, también se analizó la cantidad promedio de eventos procesados en cada período, la cual está graficada en la Figura 5.33 y 5.34. Como se puede apreciar, en los primeros 50 segundos se puede ver una mejora considerable en la cantidad de eventos procesados, donde posteriormente se procesan aproximadamente 480 eventos por período con monitoreo, a diferencia del SPS sin monitoreo, que procesa 90 eventos por período aproximadamente, habiendo una mejora del $615,38\,\%$. Esta mejora se debe al factor de la replicación de los operadores que posee mayor sobrecarga, por lo que al aumentar la cantidad de réplicas, aumenta la tasa de procesamiento, lo significa mayor cantidad de flujo para el próximo operador.

Así también, se puede observar en la Figura 5.35 y 5.36 la cantidad total de eventos procesados con el transcurso de la ejecución en cada uno de los operadores, con y sin uso del monitoreo respectivamente. En el primer gráfico se denota como los cuatro operadores del SPS van aumentando casi linealmente

de la misma manera, tan sólo existe una menor cantidad de eventos procesados en el tercer PE, lo cual se traslada al cuatro PE, debido que al procesar menor cantidad de eventos el tercer PE, llega menor cantidad de datos al cuarto PE. En este gráfico se llegó a un total de 401.618 eventos procesados. En cambio, en el segundo gráfico existe una curva muy distinta por los distintos operadores, lo cual se ve reflejado desde la cantidad de eventos procesados en el primer operador hasta la cantidad total de eventos procesados por el sistema, el cual es de 67.141, existiendo una mejora del $598,171\,\%$ con el uso del sistema de distribución de carga.

El segundo experimento consistió en enviar un flujo dinámico de datos, y realizando una comparación de los sistemas con y sin monitoreo de la carga de los operadores.

En construcción...

5.3.2 Segundo experimento

En la segunda aplicación se procedió a realizar cuatro experimentos distintos, donde cada par era una comparación del experimento con y sin uso del monitor. Las dos primeras se consideró un envío constante de la fuente de datos, y las dos segundas se consideró un envío dinámico de la fuente de datos.

Para el análisis de los experimento se consideró la cantidad total de eventos y las estadísticas de cada PE en el transcurso de la ejecución de la aplicación.

Para analizar el comportamiento del sistema, se puede observar en la Figura 5.17 y 5.18 las estadísticas del primer operador del grafo, con y sin monitoreo en la carga de los operadores respectivamente. En el primer gráfico la tasa de llegada (λ) en los primeros 200 segundos se encuentran ciertos *peak*, los cuales se deben al *delay* en la toma de estadísticas, producto de la replicación del siguiente PE en el sistema, como se puede demostrar en la Figura 5.19. Independiente del uso del monitoreo, se puede denotar que el comportamiento

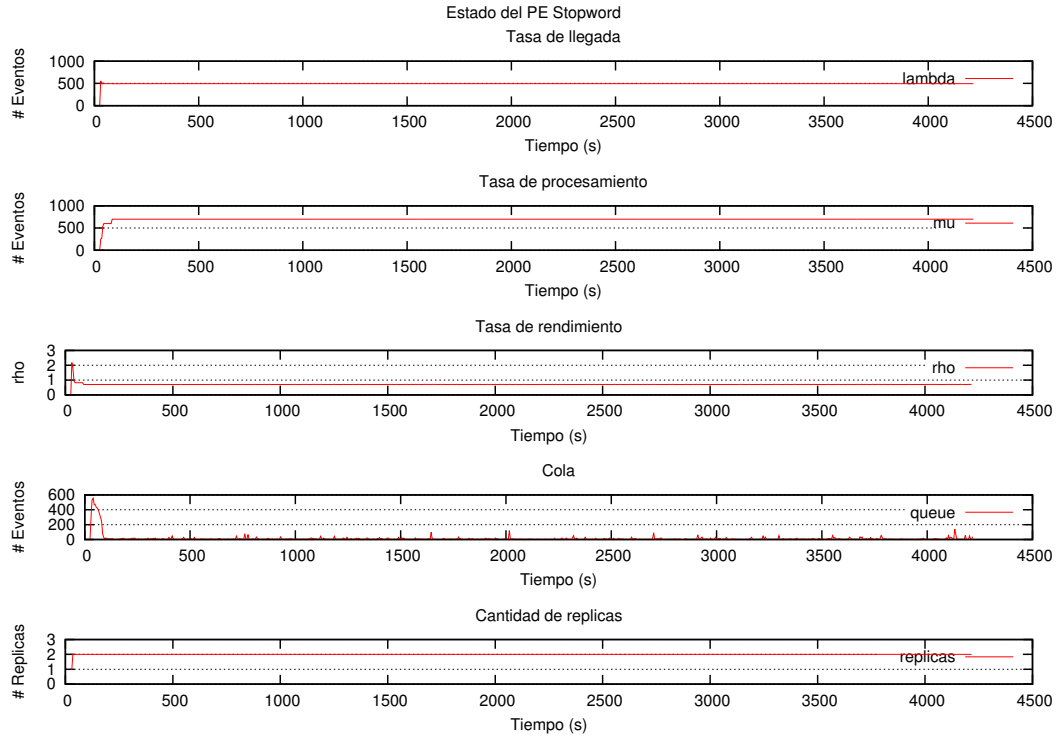

Figura 5.5: Estadísticas del PE Stopword en la primera aplicación con un envío constante de la fuente de datos con uso del monitor.

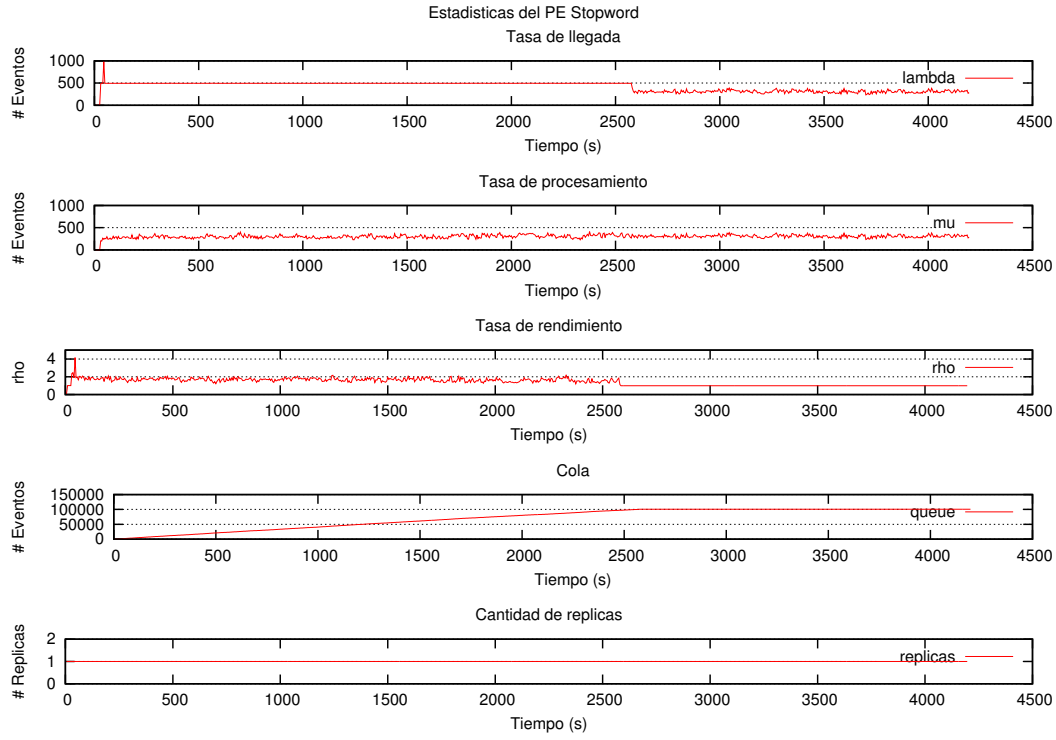

Figura 5.6: Estadísticas del PE Stopword en la primera aplicación con un envío constante de la fuente de datos sin uso del monitor.

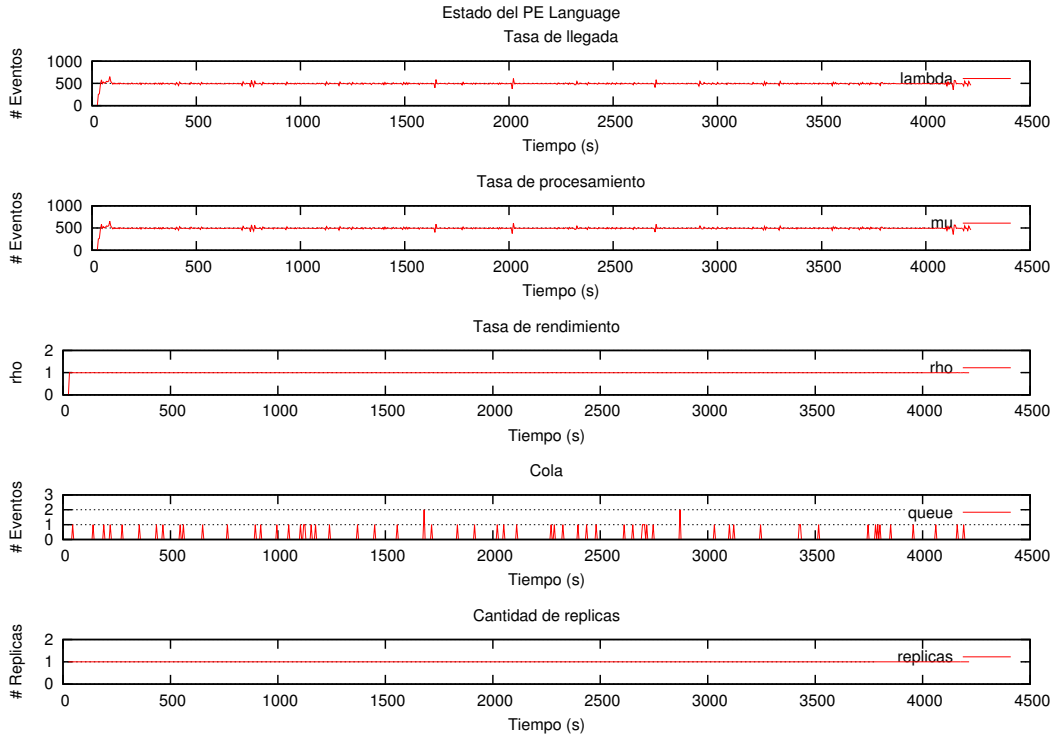

Figura 5.7: Estadísticas del PE Language en la primera aplicación con un envío constante de la fuente de datos con uso del monitor.

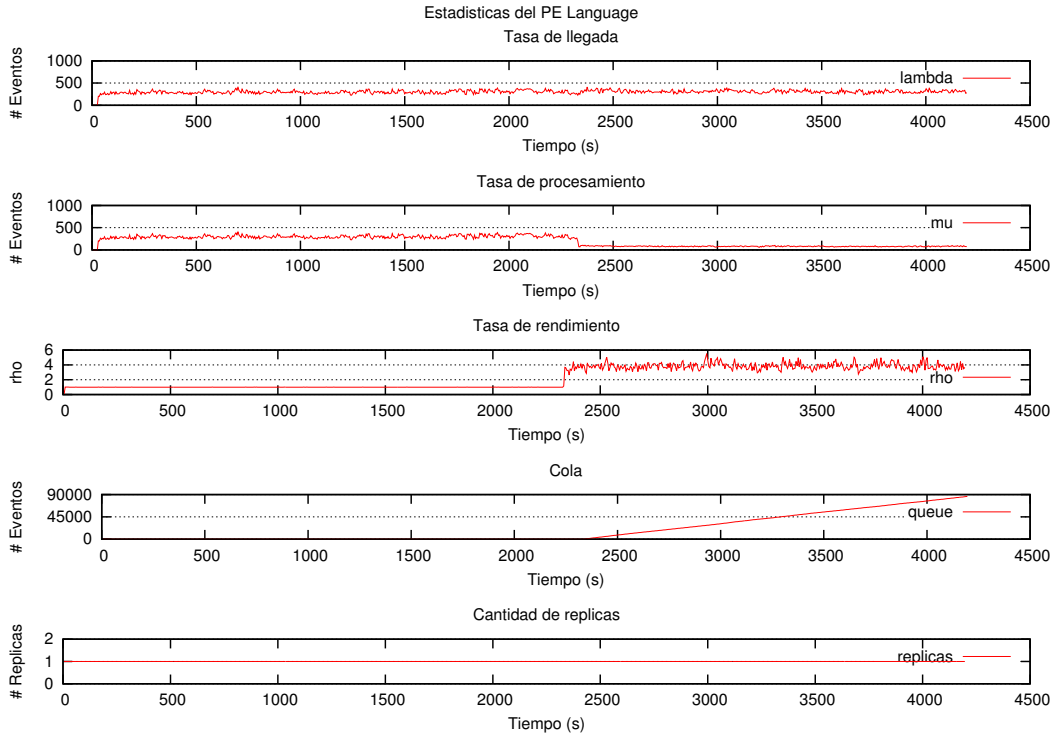

Figura 5.8: Estadísticas del PE Language en la primera aplicación con un envío constante de la fuente de datos sin uso del monitor.

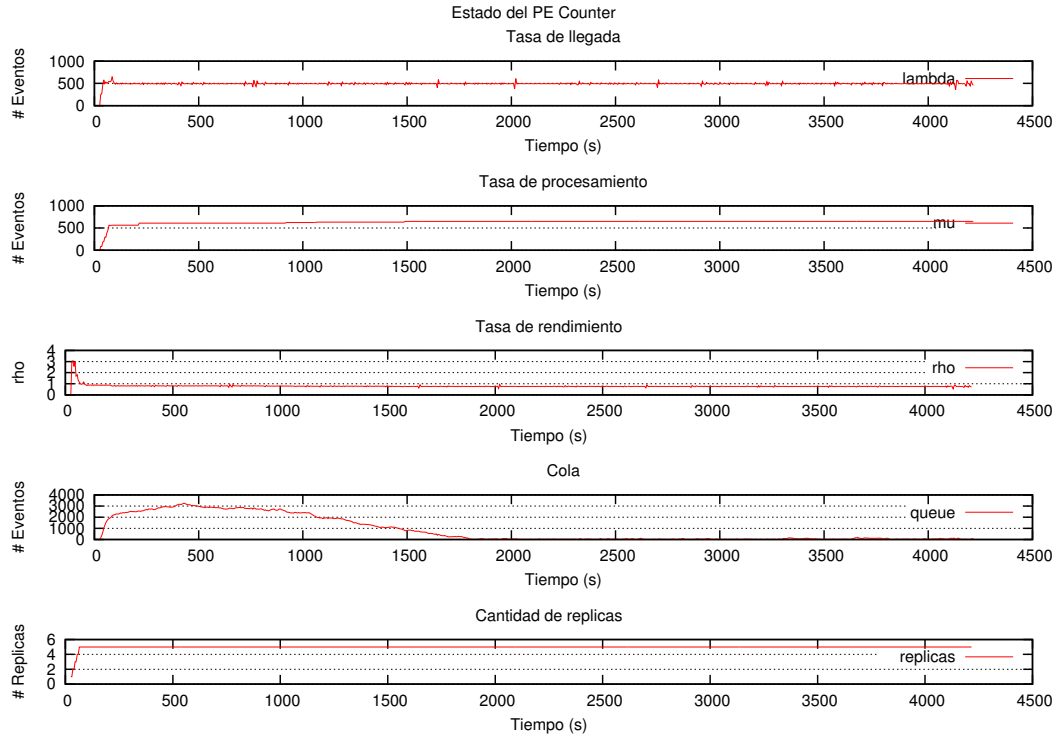

Figura 5.9: Estadísticas del PE Counter en la primera aplicación con un envío constante de la fuente de datos con uso del monitor.

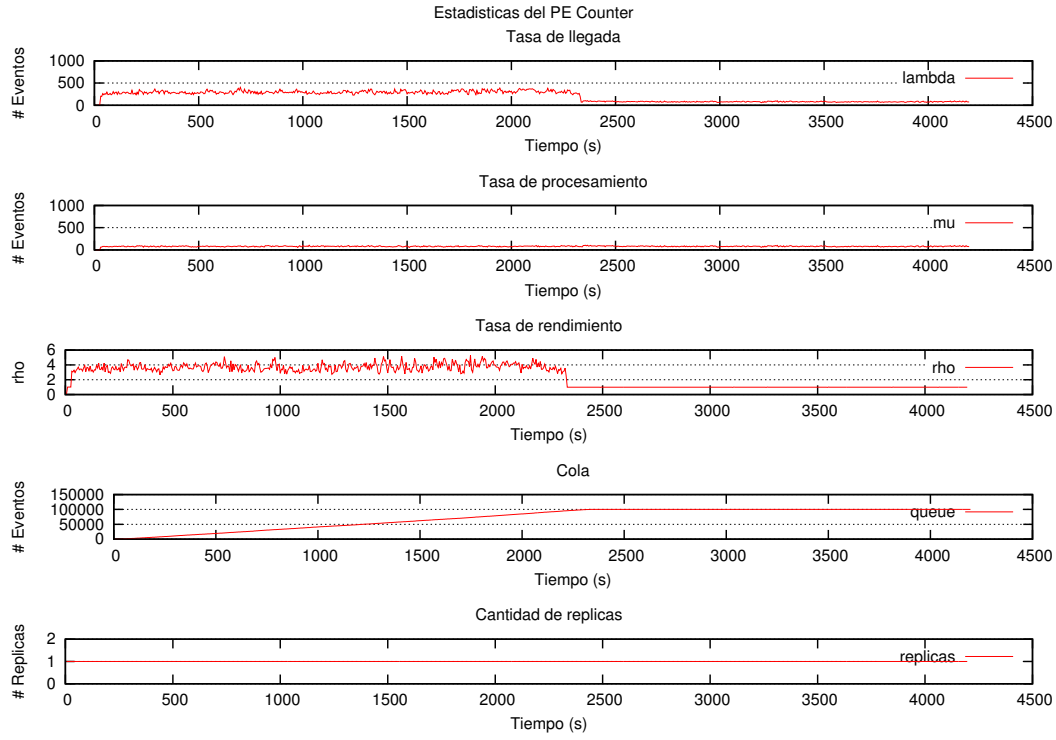

Figura 5.10: Estadísticas del PE Counter en la primera aplicación con un envío constante de la fuente de datos sin uso del monitor.

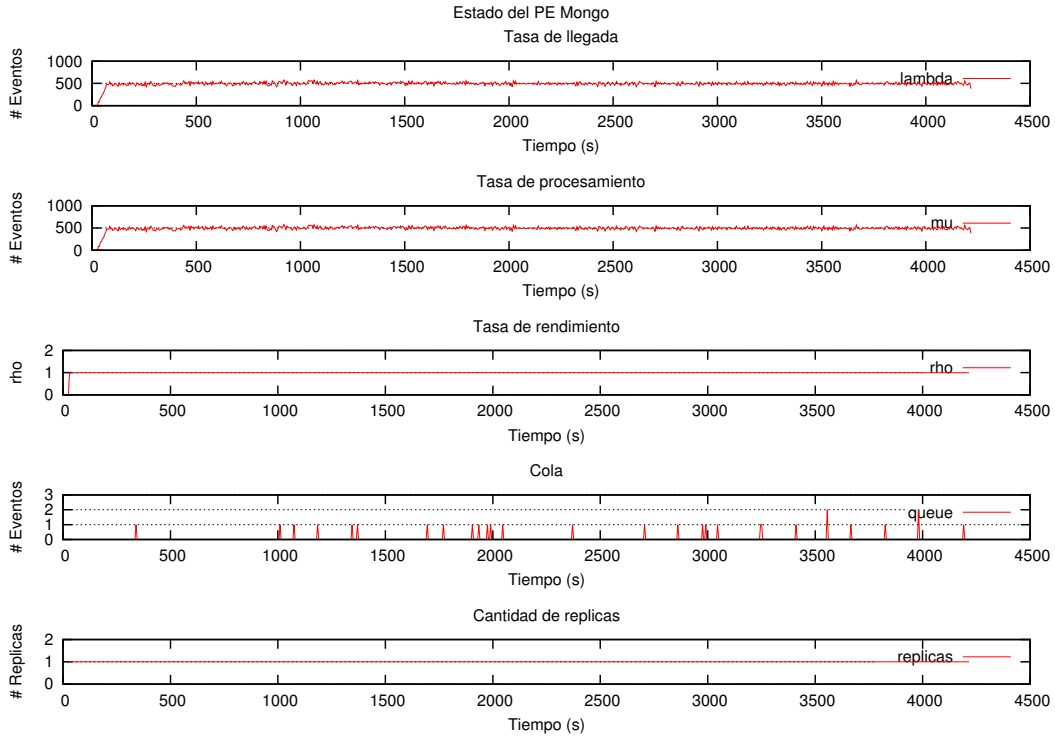

Figura 5.11: Estadísticas del PE Mongo en la primera aplicación con un envío constante de la fuente de datos con uso del monitor.

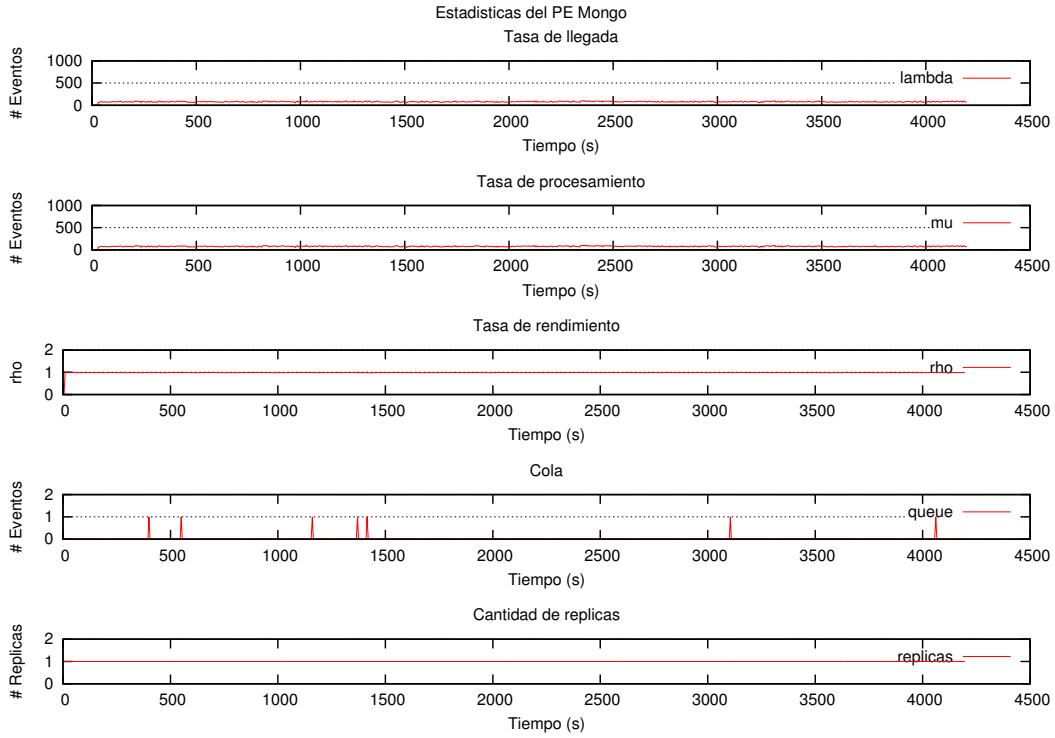

Figura 5.12: Estadísticas del PE Mongo en la primera aplicación con un envío constante de la fuente de datos sin uso del monitor.

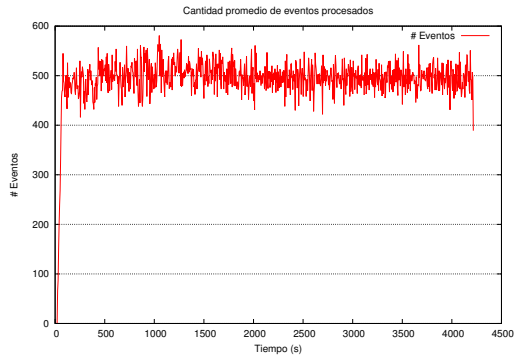

Figura 5.13: Tiempo promedio de procesamiento de un evento en la primera aplicación con una fuente de datos de distribución uniforme usando monitor.

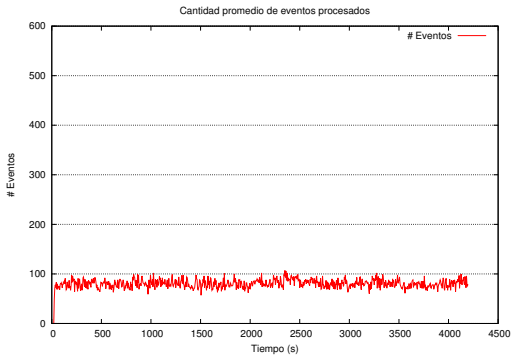

Figura 5.14: Tiempo promedio de procesamiento de un evento en la primera aplicación con una fuente de datos de distribución uniforme no usando monitor.

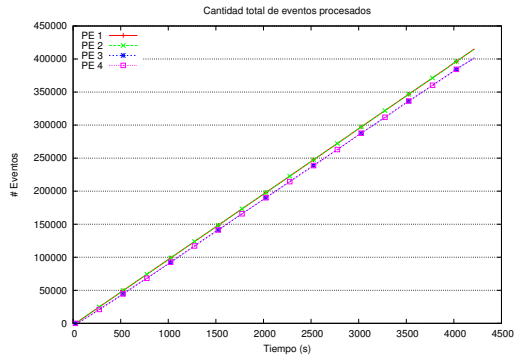

Figura 5.15: Cantidad total de eventos procesados en la primera aplicación con una fuente de datos de distribución uniforme usando monitor.

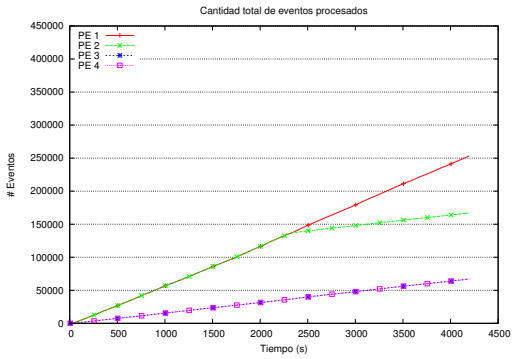

Figura 5.16: Cantidad total de eventos procesados en la primera aplicación con una fuente de datos de distribución uniforme no usando monitor.

entre los dos PEs es prácticamente igual, y esto se debe a que la carga que posee este PE es casi nula, debido que es un PE auxiliar para el PE Counter, dado que sólo separa la cantidad de palabras para que el PE Counter cuente la frecuencia de cada palabra en la información entrante.

En cambio, en la Figura 5.19 y 5.20 se puede observar una diferencia en los rendimientos del PE. Esto se debe, a que este operador posee una carga una alta carga, dado que posee una gran bolsa de palabras para comparar con las palabras del *tweets*. Debido a esto, al generar las réplicas, se produce una mejora considerable del operador en los primeros 100 segundo.

En este caso, el predictor se activó, dado que se realiza un aumento de 9 a 14 réplicas. Si bien, la cantidad de réplicas fue mayor a lo necesario, dado que

el promedio de tasa de procesamiento posterior al segundo 100 es de 0.63, no se considera el operador en estado ocioso, por lo que no se disminuye la cantidad de réplicas.

Un detalle importante a destacar, es que independiente que se generen más réplicas, el sistema no procesa mayor cantidad de eventos, como se puede observar en la cola del operador. Este problema fue detectado por la implementanción realizada en el SPS de S4, dado que posterior a un período de tiempo, la tasa de procesamiento disminuye considerablemente. La hipótesis que se posee de este problema es que el sistema no elimina los eventos procesados, por lo que independiente de si es procesado o no, continúa en el *buffer*, de esta manera, la cola sigue aumentando, sin ser removido por el *garbage collector* de Java.

En el último PE, se puede ver que existe una baja cantidad de eventos entrantes, como se muestra en la Figura 5.21 y 5.22. Esto se debe, a que los eventos entrantes sólo son enviados cada 10 segundos por el PE Counter, por lo que son una pequeña cantidad de eventos los que son enviados. En el primer gráfico, se puede ver que la tasa de llegada aumenta posterior a los 100 segundos, esto se debe a que se realizaron réplicas, por lo que cada una de las réplicas envía un evento cada 10 segundos, habiendo mayor flujo. En cambio, en el segundo gráfico, el flujo de entrada sólo se condiciona por una réplica, por lo que no se procesa la misma cantidad de eventos. Cabe destacar, que como la tasa de llegada es pequeña, no existe un problema en la tasa de rendimiento del operador, aunque posea un alto cómputo de ejecución su tarea.

Finalmente, en la Figura 5.23 y 5.24 se muestra la cantidad total de eventos procesados. En el primer gráfico se puede analizar que la diferencia de la pendiente entre las rectas del primer y segundo PE, es menor que en el segundo gráfico. Esto se debe al aumento de la cantidad de réplicas, de esta manera puede procesar mayor cantidad de eventos, siendo un procesamiento de 181114 con uso el monitor contra 30049 sin uso del monitor, habiendo una mejora del $602,7288\,\%$. El tercer PE procesa pocos eventos, debido que sólo le llega un evento por réplica cada 10 segundos.

Dentro de los análisis importantes a realizar por parte de este experimento, es que en el gráfico 5.35 no existe una mejora por parte del segundo PE de manera paralela al flujo de datos emanado por el primer PE. Esto se debió al funcionamiento de la herramienta S4, debido que después de un período de tiempo no procesa la misma cantidad de eventos, por lo que independiente que se siga replicando, no existe una mejora en el sistema. Este problema se debe a la forma en que esta implementando el *buffer* de S4, debido que éste se va llenando, pero no se libera, por lo que al llenarse bloquea el envió de los datos por parte del sistema, generando una demora en el procesamiento.

5.3.3 Tercer experimento

En la tercera aplicación se procedió a realizar dos experimentos, ambos con envío constante de la fuente de datos, donde el SPS funciona con y sin uso del monitor.

Para el análisis de los experimento se consideró el consumo de RAM, el uso de la CPU, la cantidad total de eventos, la cantidad promedio de eventos procesados por período y las estadísticas de cada PE en el transcurso de la ejecución de la aplicación.

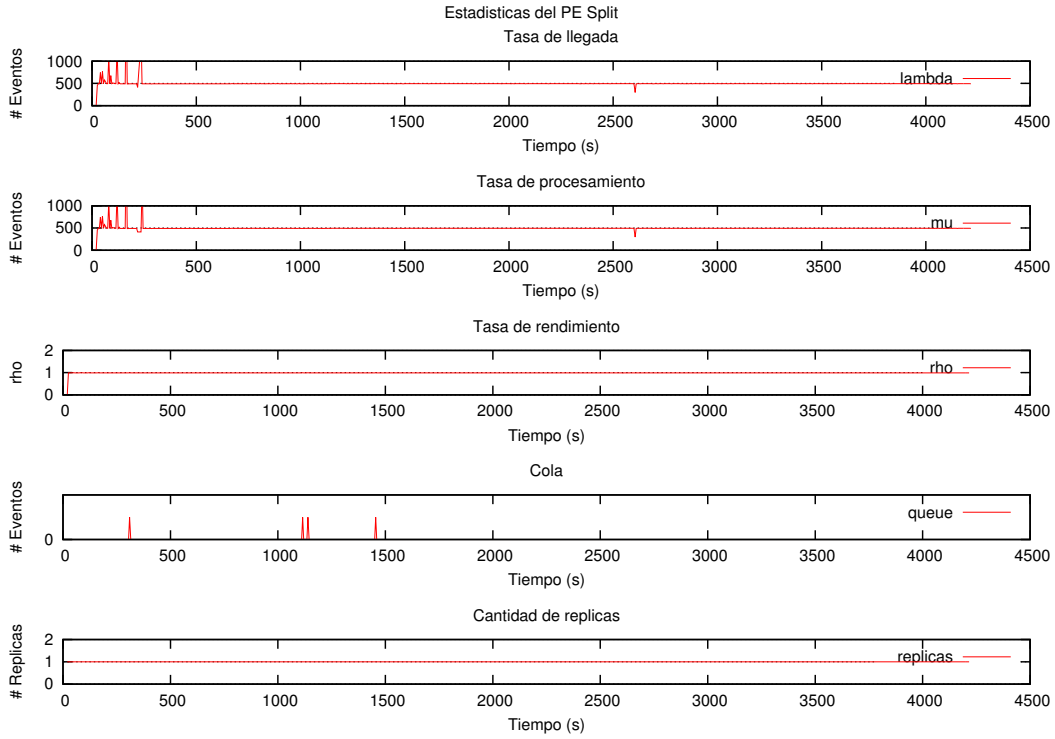

Figura 5.17: Estadísticas del PE Mongo en la primera aplicación con un envío constante de la fuente de datos con uso del monitor.

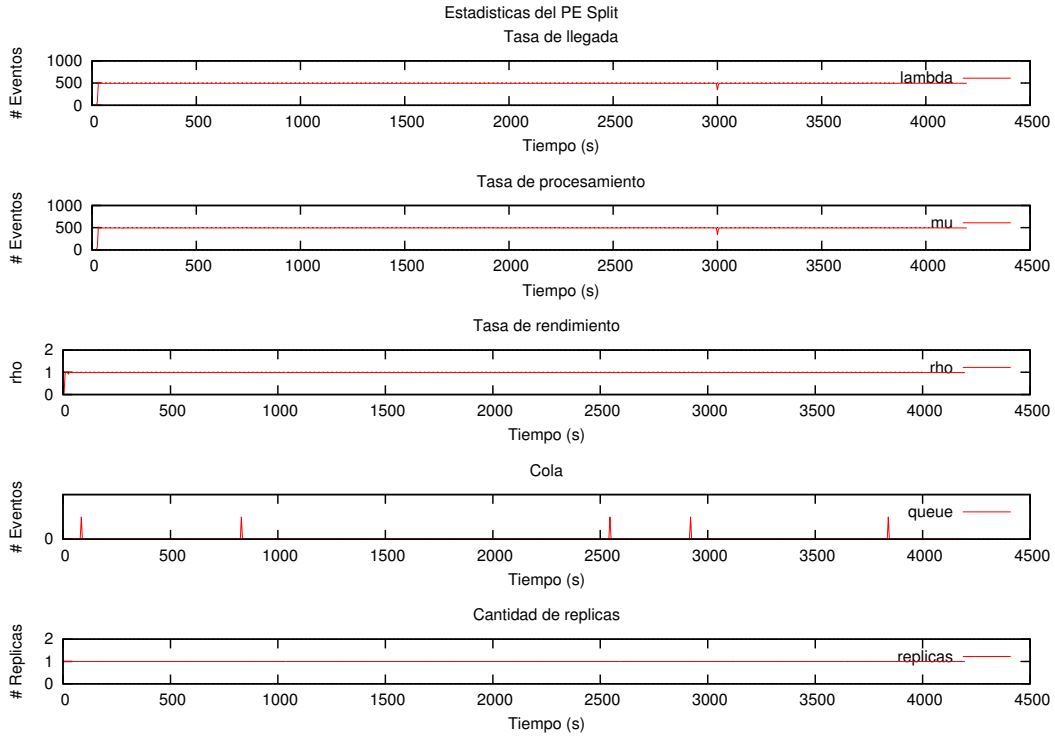

Figura 5.18: Estadísticas del PE Mongo en la primera aplicación con un envío constante de la fuente de datos sin uso del monitor.

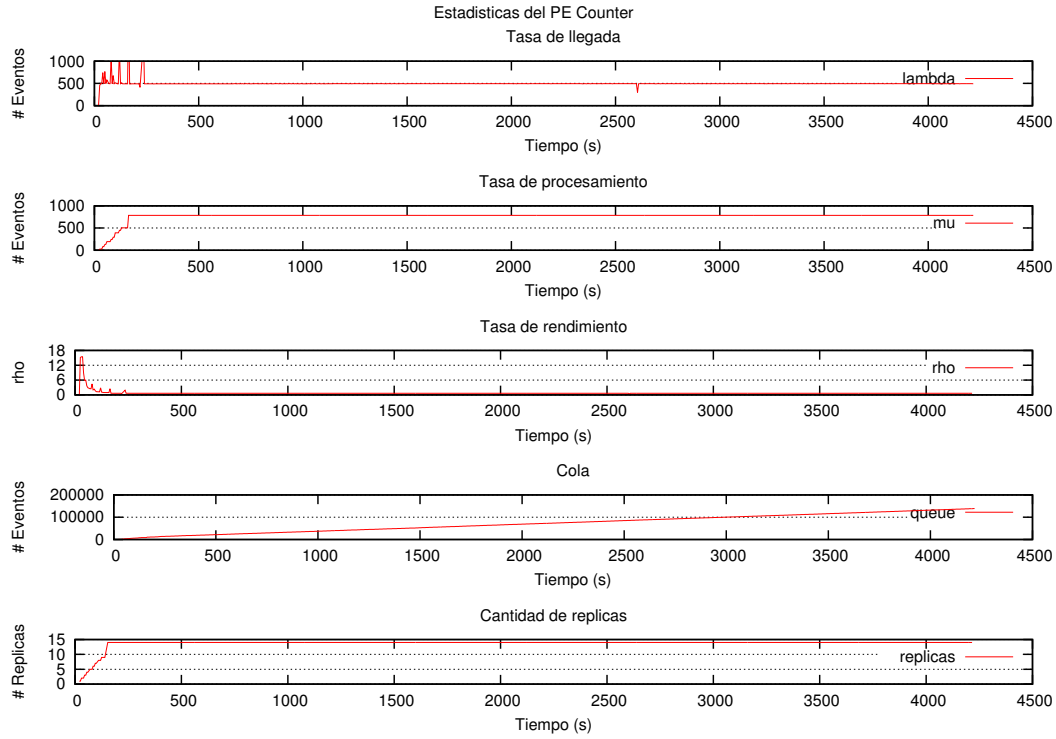

Figura 5.19: Estadísticas del PE Mongo en la primera aplicación con un envío constante de la fuente de datos con uso del monitor.

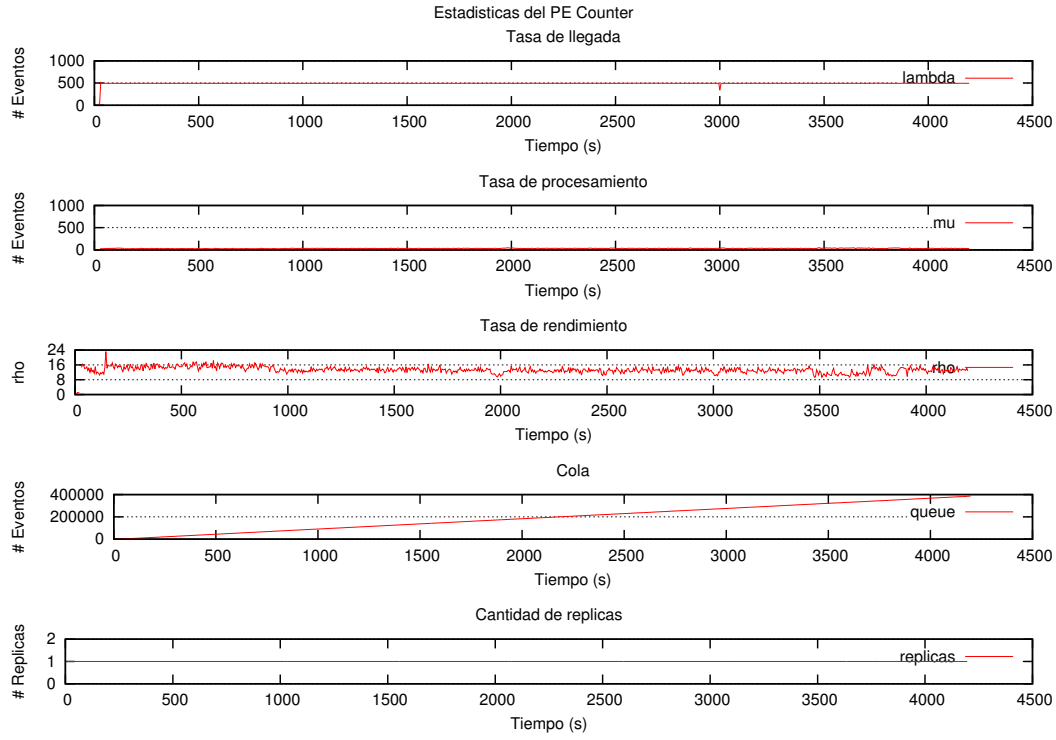

Figura 5.20: Estadísticas del PE Mongo en la primera aplicación con un envío constante de la fuente de datos sin uso del monitor.

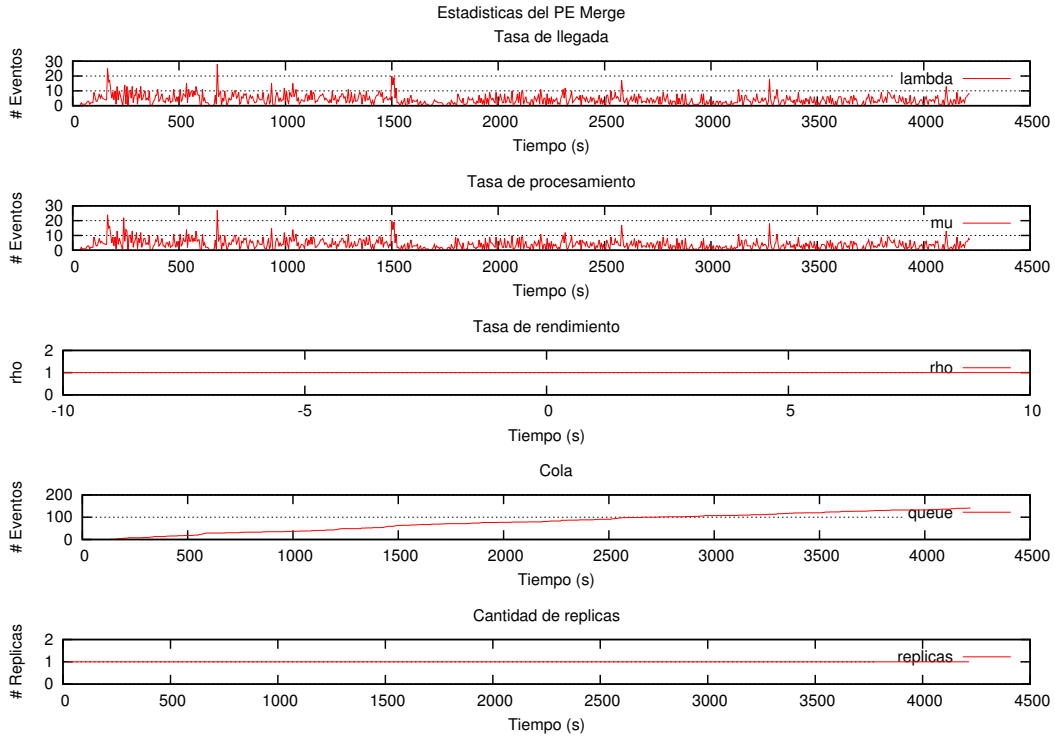

Figura 5.21: Estadísticas del PE Mongo en la primera aplicación con un envío constante de la fuente de datos con uso del monitor.

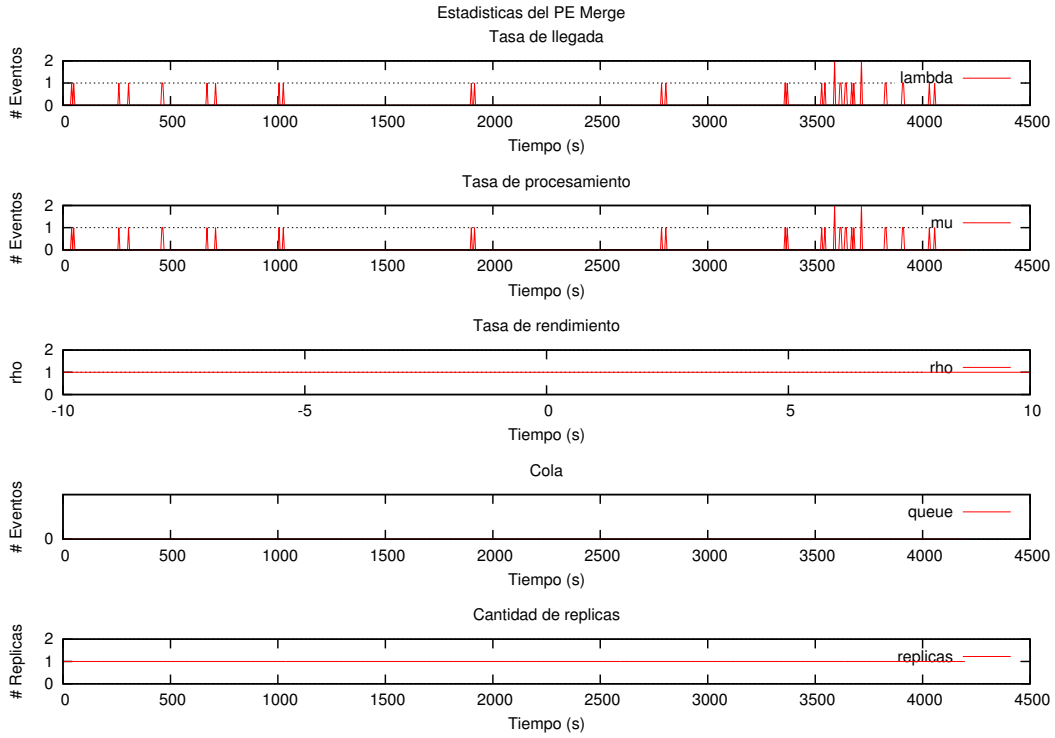

Figura 5.22: Estadísticas del PE Mongo en la primera aplicación con un envío constante de la fuente de datos sin uso del monitor.

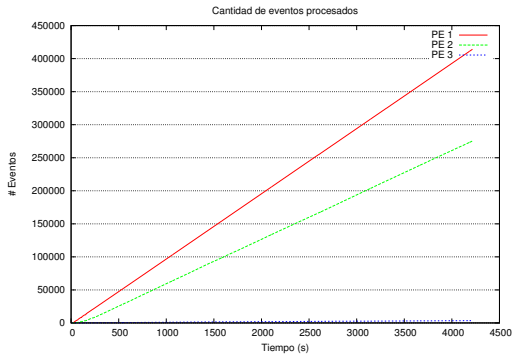

Figura 5.23: Cantidad total de eventos procesados en la primera aplicación con una fuente de datos de distribución uniforme usando monitor.

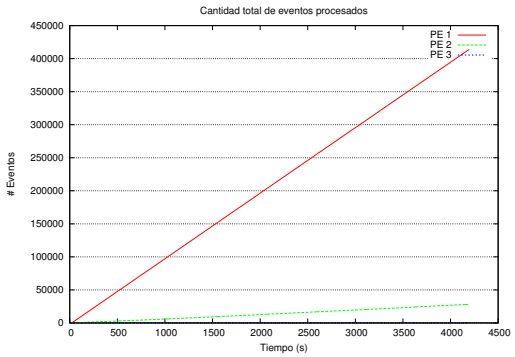

Figura 5.24: Cantidad total de eventos procesados en la primera aplicación con una fuente de datos de distribución uniforme no usando monitor.

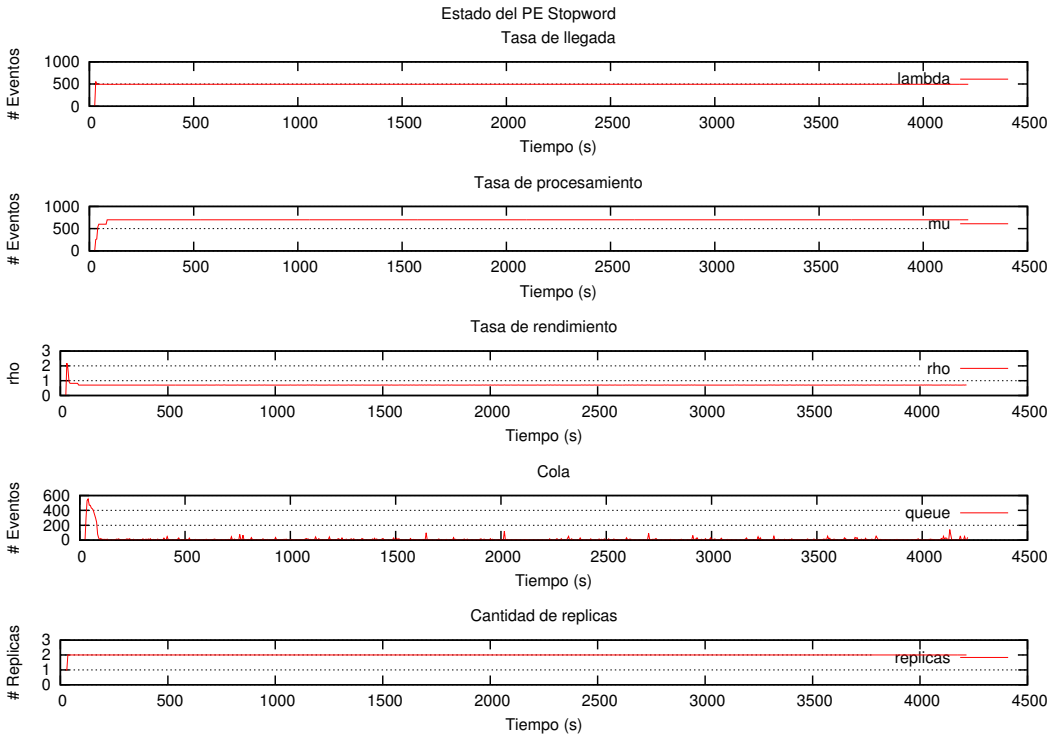

Figura 5.25: Estadísticas del PE Stopword en la primera aplicación con un envío constante de la fuente de datos con uso del monitor.

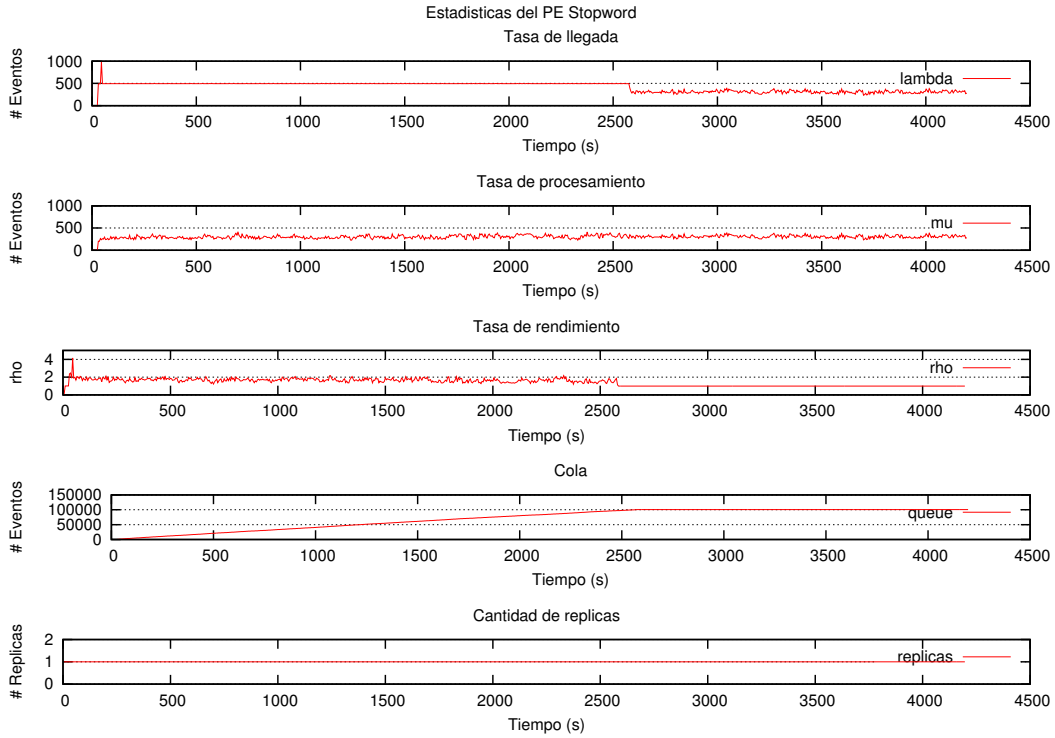

Figura 5.26: Estadísticas del PE Stopword en la primera aplicación con un envío constante de la fuente de datos sin uso del monitor.

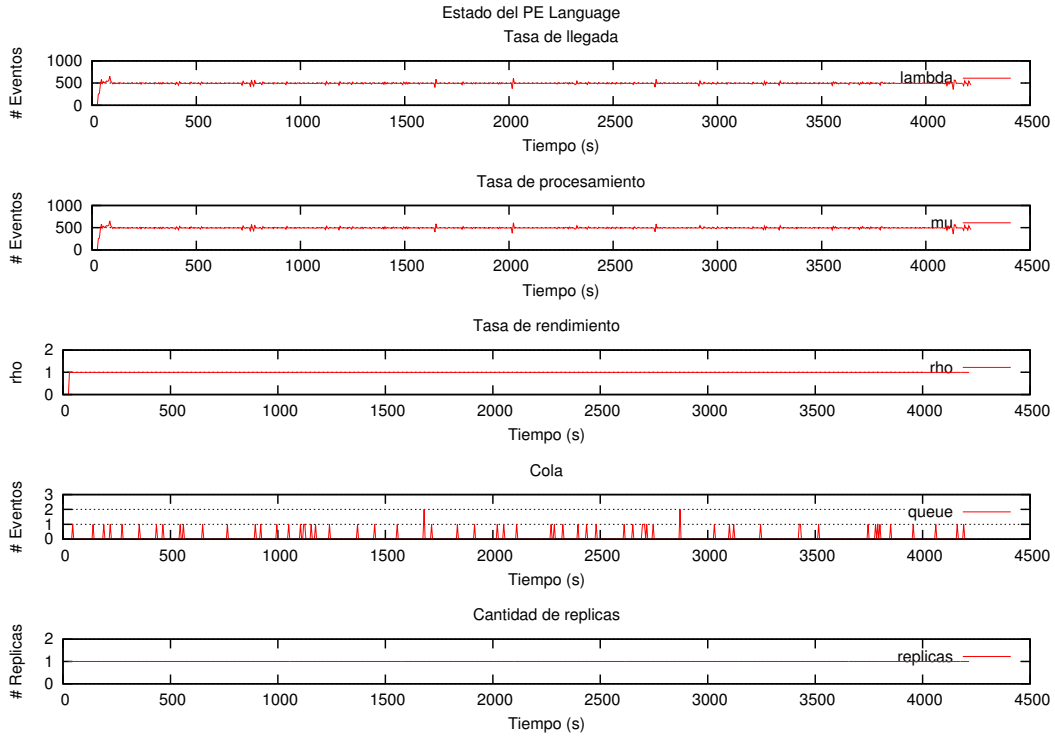

Figura 5.27: Estadísticas del PE Language en la primera aplicación con un envío constante de la fuente de datos con uso del monitor.

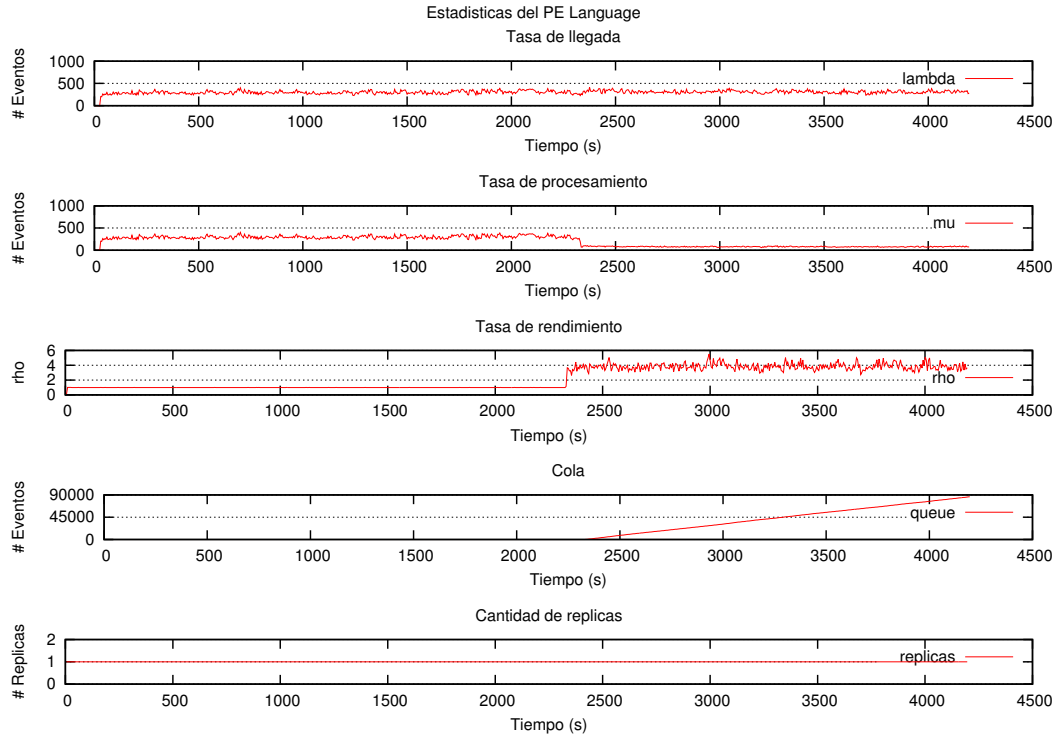

Figura 5.28: Estadísticas del PE Language en la primera aplicación con un envío constante de la fuente de datos sin uso del monitor.

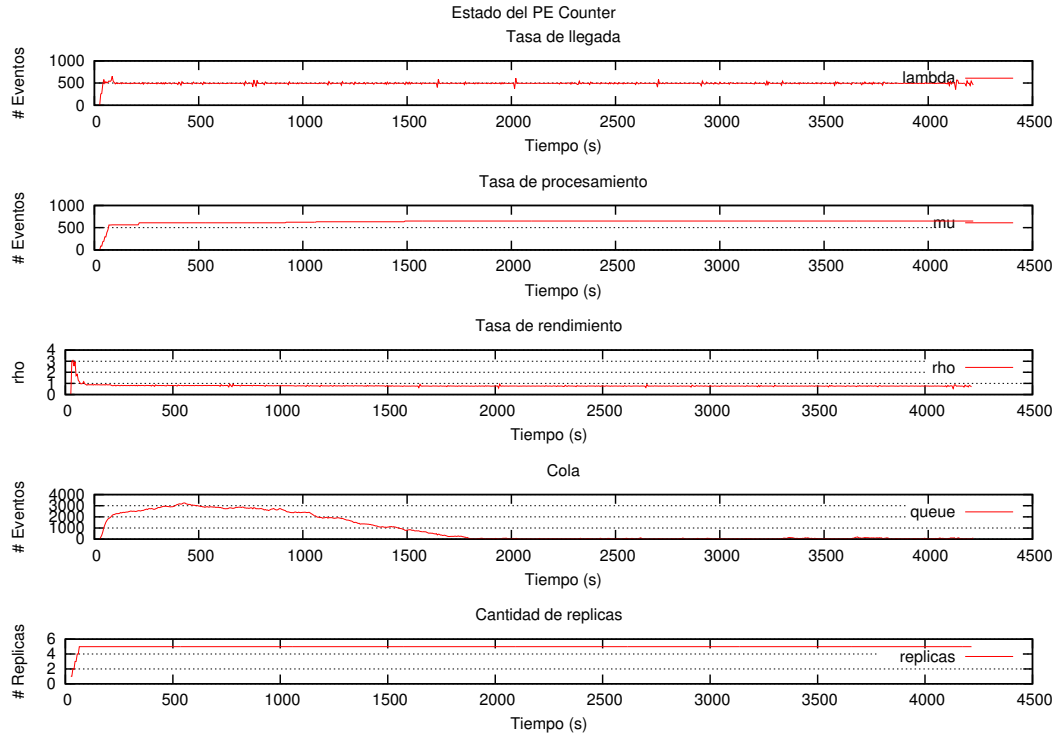

Figura 5.29: Estadísticas del PE Counter en la primera aplicación con un envío constante de la fuente de datos con uso del monitor.

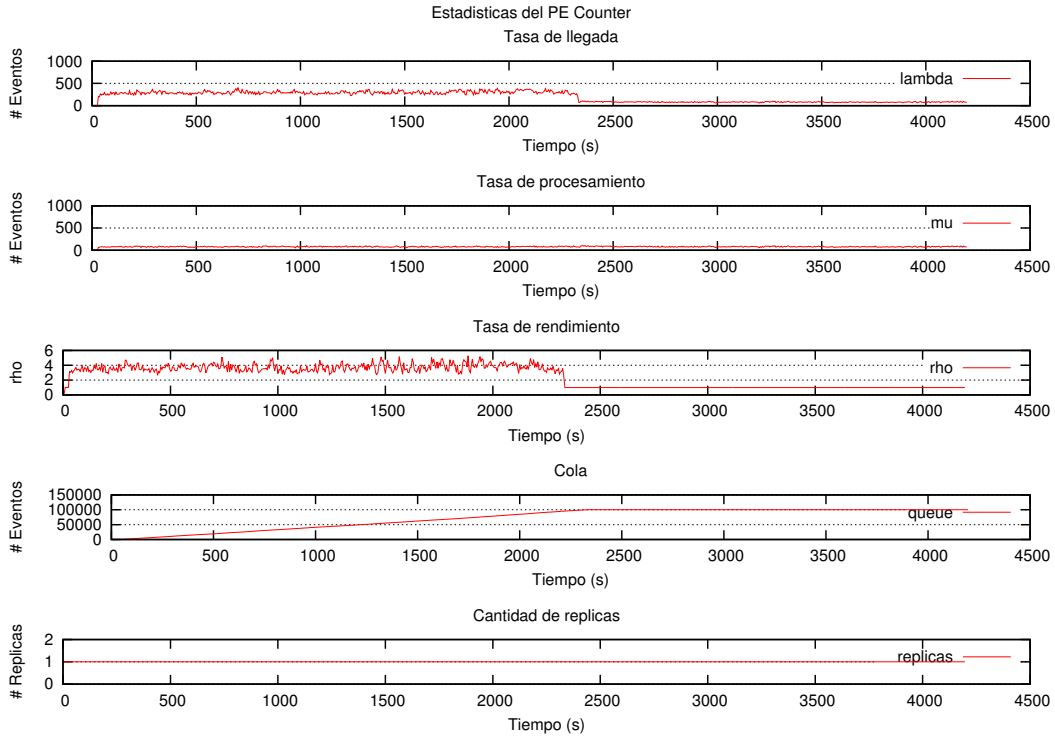

Figura 5.30: Estadísticas del PE Counter en la primera aplicación con un envío constante de la fuente de datos sin uso del monitor.

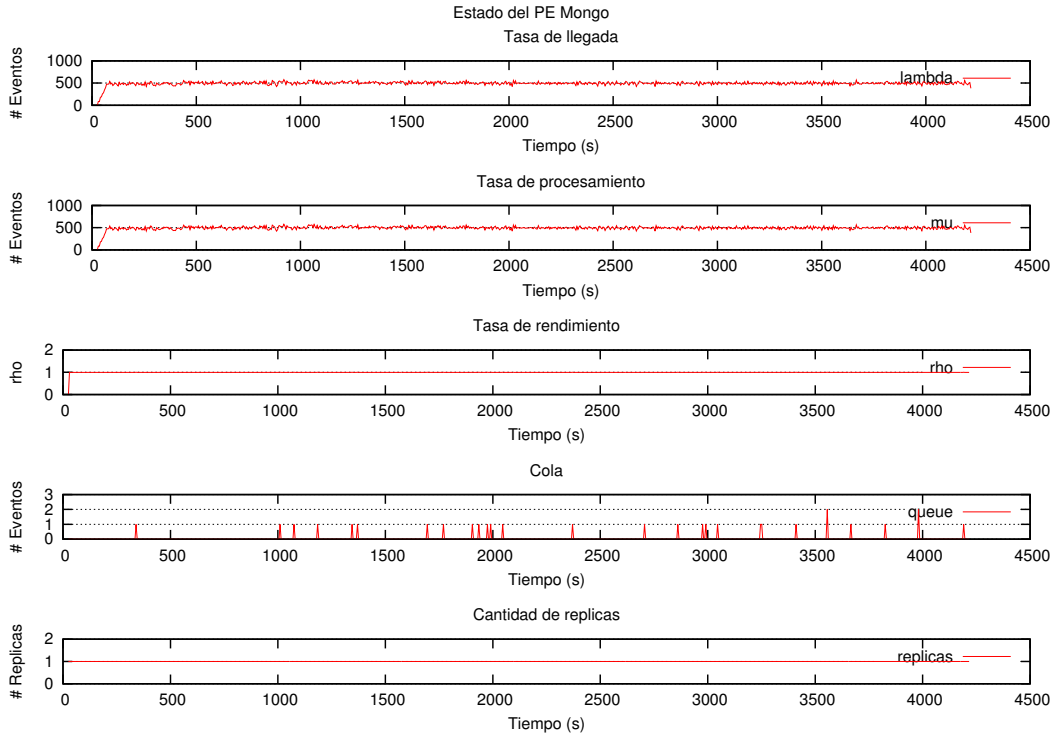

Figura 5.31: Estadísticas del PE Mongo en la primera aplicación con un envío constante de la fuente de datos con uso del monitor.

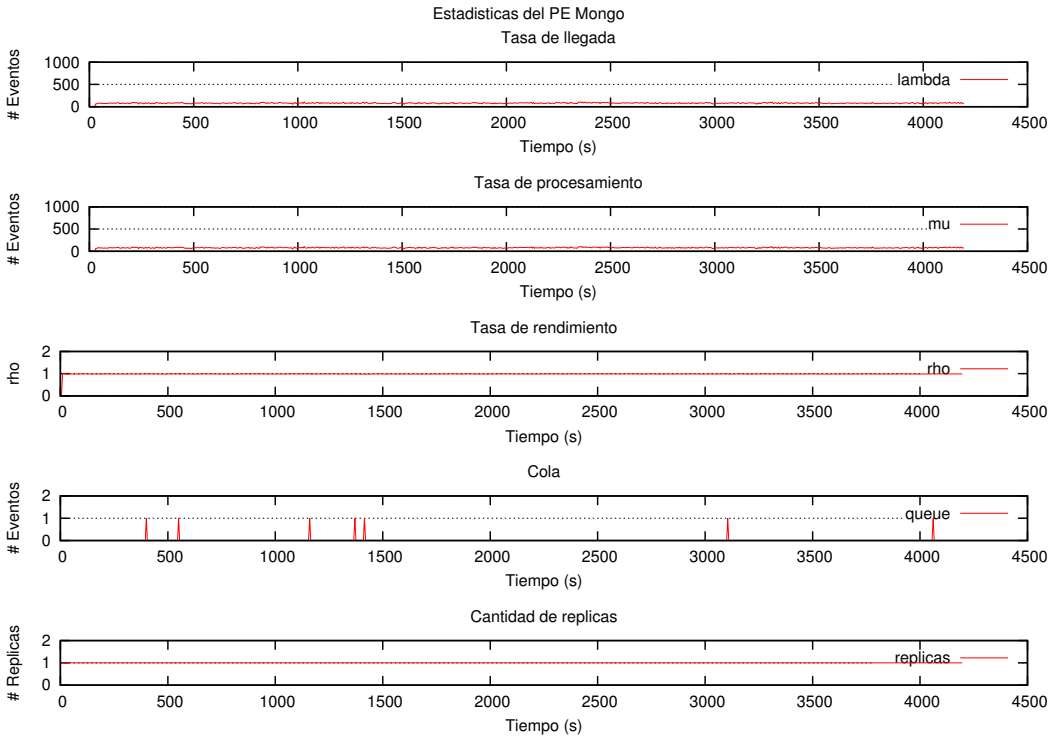

Figura 5.32: Estadísticas del PE Mongo en la primera aplicación con un envío constante de la fuente de datos sin uso del monitor.

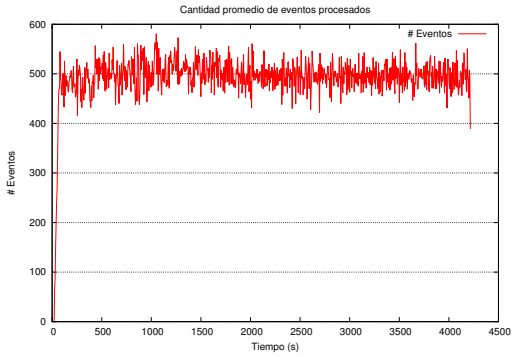

Figura 5.33: Tiempo promedio de procesamiento de un evento en la primera aplicación con una fuente de datos de distribución uniforme usando monitor.

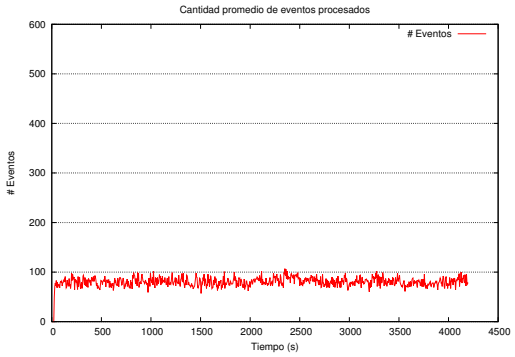

Figura 5.34: Tiempo promedio de procesamiento de un evento en la primera aplicación con una fuente de datos de distribución uniforme no usando monitor.

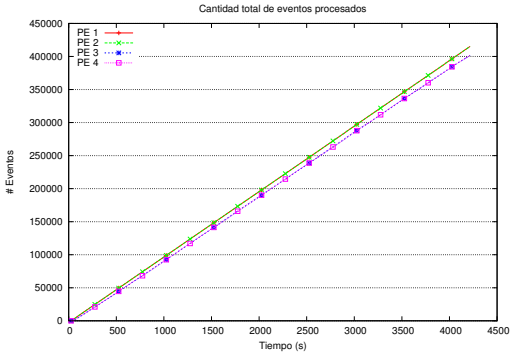

Figura 5.35: Cantidad total de eventos procesados en la primera aplicación con una fuente de datos de distribución uniforme usando monitor.

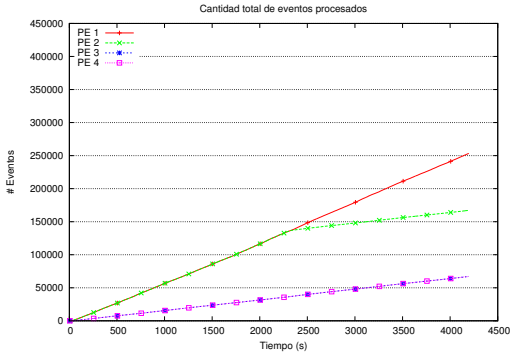

Figura 5.36: Cantidad total de eventos procesados en la primera aplicación con una fuente de datos de distribución uniforme no usando monitor.

CAPÍTULO 6. CONCLUSIONES

Dentro del trabajo realizado se encontraron distintas dificultades para poder realizar los experimentos como se deseaban, entre ellas el motor de procesamiento de *stream* S4. Si bien se escogió este SPS debido al apoyo en el ambiente de trabajo que se poseía, independiente que se encontrara descontinuado y obsoleto, el trabajo a bajo nivel que se realizó conllevó a encontrar errores en la programación del sistema en las cuales no existía apoyo técnico. Una de ellas, era la sincronización de los procesamientos de cada uno de los operadores al llevar sus mensajes, los cuales eran almacenados en una cola finita. Esta cola se volvió un problema, debido que S4 no tenía un sistema de purgamiento, lo cual significa que la cola seguía con eventos ya procesados, por lo que después de cierto período la cantidad de datos procesados disminuida considerablemente. Esto impidió que se pueda realizar experimentos con mayor cantidad de tiempo, pero no significó que el sistema realizado estuviera erróneo, sino que el sistema donde se implementó no fue el indicado.

Una de las situaciones importantes a destacar, fue la cantidad de código que se tuvo que modificar en S4, debido a problemas de implementación del sistema propuesto, dado el funcionamiento de este SPS. Uno de ellos fue la forma en que interactúa la fuente de datos, denominada *adapter* en S4, y los operadores del sistema. Por lo que se tuvo que realizar una sincronización por parte de estos dos entes, los cuales según el funcionamiento de S4 deben estar en dos máquinas distintas, aunque se puede ejecutar en la misma máquina, éste considera que son dos sistemas diferentes. Esto fue necesario para notificar al sistema que el *adapter* estaba disponible, y el sistema de distribución de carga puede empezar a funcionar, y para informar de los cambios ocurridos por parte de la fuente de datos respecto a los operadores que enviará información.

Otro de las situaciones importantes a analizar el valor que tomaba la tasa de procesamiento (μ), debido que esta podía ir variando con el transcurso del tiempo. Por ejemplo, si se posee un flujo de datos heterogéneos, puede suceder el caso que en cierto período de tiempo los datos posean mayor

tiempo de procesamiento, por lo tanto la tasa de procesamiento disminuye. Por lo tanto, surge un análisis engañoso de los datos, debido que si en períodos de tiempo, se analiza su tasa de procesamiento, no indica que en el próximo período siga con el mismo comportamiento, por lo tanto, eso puede significar que existen tomas de decisiones erróneas según el comportamiento de los datos. Es decir, si consideramos que según un período de tiempo calculamos su tasa de procesamiento, y en el próximo período utilizamos esa tasa como referencia, puede ser que los datos no sean del mismo tiempo de procesamiento que los anteriores, lo cual genera un mal análisis del comportamiento del sistema. Debido a esto, es que se consideran datos homogéneos, de tal manera que pueda encontrase una tasa de procesamiento estable para el sistema, y no encontrar ambigüedades en los cálculos realizados.

Dentro de las conclusiones que se puede realizar respecto al trabajo realizado, es que es un sistema que posee un buen análisis del sistema lógico, debido a las estadísticas obtenidas por parte del sistema, pero no analiza la cantidad de recursos disponibles por parte de la máquina, lo cual es una desventaja considerable, a excepción que se posea un supercomputador. Esto se presenta, debido que al momento de replicar, se utiliza a distribuir la carga en el computador, pero esto no significa que la máquina pueda tener una sobrecarga a futuro debido al alto nivel de procesamiento que posee, es por esto que en el diseño se específico un límite en la cantidad de réplicas que pueden realizarse.

Pero independiente de su desventaja, posee varias ventajas debido al bajo cómputo que se posee por parte de los cálculos realizados, tanto por el algoritmo reactivo como predictivo, siendo bajo el *overhead* existente y una rápida respuesta del estado de cada operador. También es importante destacar el simple y rápido análisis de los datos, ya sea la distribución de la carga en cada uno de los operadores o en la forma en que se determina si un operador posee sobrecarga o no, como también la disminución de sus réplicas en caso que este ocioso. El comportamiento elástico que posee el sistema es una gran ventaja, debido a la optimización de los recursos y al dinamismo que se posee en este.

6.1 TRABAJO FUTURO

Dentro de las mejores que se puede realizar por parte del sistema son fundamental dos: sistema de distribución que soporte más de una máquina y el predictor que dinamicamente se adapte al historial realizado.

En el primer caso, se podría realizar un sistema de monitoreo, en el cual se posea una máquina se analiza los datos de cada una de las máquinas disponibles, y ésta posea además las réplicas primarias de cada operador. En caso que exista una sobrecarga por parte de un operador, es necesario realizar una réplica por parte del monitor centralizado, y que este determine a cual de las máquinas disponibles debe enviar los datos según la cantidad de recursos disponibles, tasa de procesamiento por parte del operador, entre otras variables. De esta manera, se posee un sistema escalable, debido que se puede implementar un SPS en un servicio de *Cloud Computing*, de tal manera que en caso que sea necesario mayor cantidad de máquinas, se añadan y el monitor pueda distribuir mayor carga a estas nuevas máquinas, en caso de ser requerido.

En el segundo caso, se podría realizar un análisis más detallado de la historia, de tal manera que según el comportamiento que éste posea indica cuanta son las réplicas que se desean aumentar o disminuir según su historia. Por ejemplo, en el caso que la tasa de rendimiento sea muy alta en la historia, significa que posee una gran cantidad de réplicas, pero si la tasa de rendimiento es alta, pero no excesivamente, la cantidad de réplicas debe ser menor. En el caso propuesto, se realizó con valores constantes, por lo que podría generarse una mejora a futuro con este tipo de análisis.

Así mismo, se podría realizar un estudio respecto a *peak* que se encuentren de forma estacionaria según cierto flujo de datos, y que el sistema se adapte dinámicamente a esos *peak*. Por ejemplo, en el caso de *Twitter* existen períodos del día que los usuarios comentan más, por lo tanto, en esos períodos aumentar la cantidad de recursos, y en los períodos que no comentan tanto, se podría disminuir la cantidad de recursos.

REFERENCIAS

(2011). Consistent hashing. In *Encyclopedia of Parallel Computing*, (p. 406).

Abadi, D. J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., & Zdonik, S. B. (2003). Aurora: a new model and architecture for data stream management. *VLDB J.*, *12*(2), 120–139.

Alves, D., Bizarro, P., & Marques, P. (2010). Flood: Elastic streaming mapreduce. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, (pp. 113–114).

Andrade, H., Gedik, B., & Turaga, D. (2014). *Fundamentals of Stream Processing: Application Design, Systems, and Analytics*. Cambridge University Press.

Appel, S., Frischbier, S., Freudenreich, T., & Buchmann, A. P. (2012). Eventlets: Components for the integration of event streams with SOA. In *2012 Fifth IEEE International Conference on Service-Oriented Computing and Applications (SOCA), Taipei, Taiwan, Diciembre 17-19, 2012*, (pp. 1–9).

Bhuvanagiri, L., Ganguly, S., Kesh, D., & Saha, C. (2006). Simpler algorithm for estimating frequency moments of data streams. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2006, Miami, Florida, USA, January 22-26, 2006*, (pp. 708–713).

Birman, K. P. (2012). *Guide to Reliable Distributed Systems - Building High-Assurance Applications and Cloud-Hosted Services*. Texts in Computer Science. Springer.

Bose, S. K. (2013). *An introduction to queueing systems*. Springer Science & Business Media.

Breuer, L., & Baum, D. (2005). *An introduction to queueing theory and matrix-analytic methods*. Springer.

Brucker, P. (2004). *Scheduling algorithms*. Springer.

Casavant, T. L., & Kuhl, J. G. (1988). A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Trans. Software Eng.*, *14*(2), 141–154.

Chakravarthy, S., & Jiang, Q. (2009). *Stream Data Processing: A Quality of Service Perspective - Modeling, Scheduling, Load Shedding, and Complex Event Processing*, vol. 36 of *Advances in Database Systems*. Kluwer.

Chen, C. L. P., & Zhang, C. (2014). Data-intensive applications, challenges, techniques and technologies: A survey on Big Data. *Inf. Sci.*, *275*, 314–347.

Ching, W. K., & Ng, M. K. (2006). *Markov chains*. Springer.

Chodorow, K. (2013). *MongoDB: the definitive guide*. .º'Reilly Media, Inc.".

De Sapio, R. (1978). Calculus for the life sciences.

Dong, F., & Akl, S. G. (2006). Scheduling algorithms for grid computing: State of the art and open problems.

Dong, M., Tong, L., & Sadler, B. M. (2007). Information retrieval and processing in sensor networks: Deterministic scheduling versus random access. *IEEE Transactions on Signal Processing*, *55*(12), 5806–5820.

Falk, M., Marohn, F., Michel, R., Hofmann, D., Macke, M., Tewes, B., & Dinges, P. (2012). A first course on time series analysis: examples with sas.

Fernandez, R. C., Migliavacca, M., Kalyvianaki, E., & Pietzuch, P. (2013). Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, (pp. 725–736).

Ganguly, S. (2009). Deterministically estimating data stream frequencies. In *Combinatorial Optimization and Applications, Third International Conference, COCOA 2009, Huangshan, China, June 10-12, 2009. Proceedings*, (pp. 301–312).

Gedik, B., Schneider, S., Hirzel, M., & Wu, K. (2014). Elastic scaling for data stream processing. *IEEE Trans. Parallel Distrib. Syst.*, *25*(6), 1447–1463.

Gong, Z., Gu, X., & Wilkes, J. (2010). PRESS: predictive elastic resource scaling for cloud systems. In *Proceedings of the 6th International Conference on Network and Service Management, CNSM 2010, Niagara Falls, Canada, October 25-29, 2010*, (pp. 9–16).

Gulisano, V., Jiménez-Peris, R., Patiño-Martínez, M., Soriente, C., & Valduriez, P. (2012). Streamcloud: An elastic and scalable data streaming system. *IEEE Trans. Parallel Distrib. Syst.*, *23*(12), 2351–2365.

Gupta, D., & Bepari, P. (1999). Load sharing in distributed systems. In *In Proceedings of the National Workshop on Distributed Computing*.

Hawwash, B., & Nasraoui, O. (2014). From tweets to stories: Using stream-dashboard to weave the twitter data stream into dynamic cluster models. In *Proceedings of the 3rd International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications, BigMine 2014, New York City, USA, Agosto 24, 2014*, (pp. 182–197).

Hernández Sampieri, R., Fernández Collado, C., & Baptista Lucio, P. (2010). Metodología de la investigación. *México: Editorial Mc Graw Hill*.

Hirzel, M., Soulé, R., Schneider, S., Gedik, B., & Grimm, R. (2013). A catalog of stream processing optimizations. *ACM Comput. Surv.*, *46*(4), 46:1–46:34.

Hunt, P., Konar, M., Junqueira, F. P., & Reed, B. (2010). Zookeeper: Wait-free coordination for internet-scale systems. In *2010 USENIX Annual Technical Conference, Boston, MA, USA, June 23-25, 2010*.

Ishii, A., & Suzumura, T. (2011). Elastic stream computing with clouds. In *IEEE International Conference on Cloud Computing, CLOUD 2011, Washington, DC, USA, 4-9 July, 2011*, (pp. 195–202).

Karp, R. M., Shenker, S., & Papadimitriou, C. H. (2003). A simple algorithm for finding frequent elements in streams and bags. *ACM Trans. Database Syst.*, *28*, 51–55.

Lehrig, S., Eikerling, H., & Becker, S. (2015). Scalability, elasticity, and efficiency in cloud computing: a systematic literature review of definitions and metrics. In *Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures, QoSA'15 (part of CompArch 2015), Montreal, QC, Canada, May 4-8, 2015*, (pp. 83–92).

Leibiusky, J., Eisbruch, G., & Simonassi, D. (2012). *Getting Started with Storm - Continuous Streaming Computation with Twitter's Cluster Technology*. O'Reilly.

Lin, J., & Dyer, C. (2010). *Data-Intensive Text Processing with MapReduce*. Synthesis Lectures on Human Language Technologies. Morgan & Claypool Publishers.
URL `http://dx.doi.org/10.2200/S00274ED1V01Y201006HLT007`

Madsen, K. G. S., Thyssen, P., & Zhou, Y. (2014). Integrating fault-tolerance and elasticity in a distributed data stream processing system. In *Conference on Scientific and Statistical Database Management, SSDBM '14, Aalborg, Denmark, June 30 - July 02, 2014*, (p. 48).

Mattmann, C., & Zitting, J. (2011). *Tika in action*. Manning Publications Co.

Miao, R., Yu, M., & Jain, N. (2014). NIMBUS: cloud-scale attack detection and mitigation. In *ACM SIGCOMM 2014 Conference, SIGCOMM'14, Chicago, IL, USA, August 17-22, 2014*, (pp. 121–122).

Neumeyer, L., Robbins, B., Nair, A., & Kesari, A. (2010). S4: distributed stream computing platform. In *ICDMW 2010, The 10th IEEE International Conference on Data Mining Workshops, Sydney, Australia, 14 December 2010*, (pp. 170–177).

Nguyen, H., Shen, Z., Gu, X., Subbiah, S., & Wilkes, J. (2013). AGILE: elastic distributed resource scaling for infrastructure-as-a-service. In *10th International Conference on Autonomic Computing, ICAC'13, San Jose, CA, USA, June 26-28, 2013*, (pp. 69–82).

Oberhelman, D. (2007). Coming to terms with Web 2.0. *Reference Reviews*, *21*, 5–6.

Papoulis, A. (1984). *Probability, Random Variables, and Stochastic Processes*. Mc-Graw Hill.

Pittau, M., Alimonda, A., Carta, S., & Acquaviva, A. (2007). Impact of task migration on streaming multimedia for embedded multiprocessors: A quantitative evaluation. In *Proceedings of the 2007 5th Workshop on Embedded Systems for Real-Time Multimedia, ESTImedia 2007, October 4-5, Salzburg, Austria, conjunction with CODES+ISSS 2007*, (pp. 59–64).

Rushton, A. (2010). *The handbook of logistics and distribution management*. Kogan Page Publishers.

S4 (2014). Distributed stream computing platform. [Online] `http://incubator.apache.org/s4/`.

Samza, A. (2014). Samza. [Online] `http://samza.incubator.apache.org/`.

Schneider, S., Andrade, H., Gedik, B., Biem, A., & Wu, K. (2009). Elastic scaling of data parallel operators in stream processing. In *23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009*, (pp. 1–12).

Shahrivari, S. (2014). Beyond batch processing: Towards real-time and streaming big data. *Computing Research Repository*, *abs/1403.3375*.

Sheu, T., & Chi, Y. (2009). Intelligent stale-frame discards for real-time video streaming over wireless ad hoc networks. *EURASIP J. Wireless Comm. and Networking*, *2009*.

Soong, T. T. (2004). *Fundamentals of probability and statistics for engineers*. John Wiley & Sons.

Stonebraker, M., Çetintemel, U., & Zdonik, S. B. (2005). The 8 requirements of real-time stream processing. *SIGMOD Record*, *34*(4), 42–47.

Storm (2014). Distributed and fault-tolerant realtime computation. [Online] `http://storm.incubator.apache.org/`.

Sturm, R., Morris, W., & Jander, M. (2000). Foundations of Service Level Management.

Tanenbaum, A. S., & van Steen, M. (2007). *Distributed Systems - Principles and paradigms*. Pearson Education.

Taylor, H. M., & Karlin, S. (2014). *An introduction to stochastic modeling*. Academic press.

Wenzel, S. (2014). App'ification of enterprise software: A multiple-case study of big data business applications. In *Business Information Systems - 17th International Conference, BIS 2014, Larnaca, Cyprus, Mayo 22-23, 2014. Proceedings*, (pp. 61–72).

Xing, Y., Zdonik, S. B., & Hwang, J. (2005). Dynamic load distribution in the borealis stream processor. In *Proceedings of the 21st International Conference on Data Engineering, ICDE 2005, 5-8 April 2005, Tokyo, Japan*, (pp. 791–802).

Xu, J., Chen, Z., Tang, J., & Su, S. (2014). T-storm: Traffic-aware online scheduling in storm. In *IEEE 34th International Conference on Distributed Computing Systems, ICDCS 2014, Madrid, Spain, June 30 - July 3, 2014*, (pp. 535–544).

ANEXO A. CONFORMACIÓN DE MATRIZ DE TRANSICIÓN

En el Algoritmo A.1 se puede apreciar la conformación de la matriz de transición dado la historia de un operador determinado.

Algoritmo A.1: Algoritmo para la conformación de la matriz de transición.

Entrada: rho Historial de procesamiento de tamaño n del operador ϕ.
Salida: Γ Matriz de transición del operador ϕ.

1: $\Gamma \leftarrow Matriz[3x3]$ //Matriz de transición
2: $\tau \leftarrow Arreglo[3]$ //Contador para la normalización de los datos
3: **for** $i = 1$ a n **do**
4: **if** $\rho_i < 0.5$ **and** $\rho_{i+1} < 0.5$ **then**
5: $\Gamma_{1,1} + +$
6: $\tau 1 + +$
7: **else if** $\rho_i < 0.5$ **and** $0.5 \leqslant \rho_{i+1} \leqslant 1$ **then**
8: $\Gamma_{1,2} + +$
9: $\tau 1 + +$
10: **else if** $\rho_i < 0.5$ **and** $\rho_{i+1} > 1$ **then**
11: $\Gamma_{1,3} + +$
12: $\tau 1 + +$
13: **else if** $0.5 \leqslant \rho_i \leqslant 1.5$ **and** $\rho_{i+1} < 0.5$ **then**
14: $\Gamma_{2,1} + +$
15: $\tau 2 + +$
16: **else if** $0.5 \leqslant \rho_i \leqslant 1.5$ **and** $0.5 \leqslant \rho_{i+1} \leqslant 1.5$ **then**
17: $\Gamma_{2,2} + +$
18: $\tau 2 + +$
19: **else if** $0.5 \leqslant \rho_i \leqslant 1.5$ **and** $\rho_{i+1} > 1.5$ **then**
20: $\Gamma_{2,3} + +$
21: $\tau 2 + +$
22: **else if** $\rho_i > 1$ **and** $\rho_{i+1} < 0.5$ **then**
23: $\Gamma_{3,1} + +$
24: $\tau 3 + +$
25: **else if** $\rho_i > 1$ **and** $0.5 \leqslant \rho_{i+1} \leqslant 1.5$ **then**
26: $\Gamma_{3,2} + +$
27: $\tau 3 + +$
28: **else**
29: $\Gamma_{3,3} + +$
30: $\tau 3 + +$
31: **end if**
32: **end for**
33: **for** $i = 1$ a 3 **do**
34: **if** $\tau_i \neq 0$ **then**
35: **for** $j = 1$ a 3 **do**
36: $\Gamma_{i,j} \leftarrow {}^{\Gamma_{i,j}}/_{\tau_i}$
37: **end for**
38: **end if**
39: **end for**
40: **return** Γ //Retorno de la Matriz de transición normalizada, la cual define la cadena de Markov

ANEXO B. CLASES PARA LA IMPLEMENTACIÓN DEL SISTEMA DE MONITOREO

En el Código B.1 se muestra las estadísticas de un PE en específico, donde se guarda el nombre del *stream* asociado al PE, la tasa de llegada (λ), tasa de servicio ($\mu * s$), tasa de servicio unitaria (μ), tasa de rendimiento (ρ), cola, historial del PE para el cálculo predictivo, clase del PE, cantidad de réplicas, historial de alertas para la replicación según el algoritmo reactivo y cantidad total de eventos procesados.

Estas estadísticas son las que se utilizan como entrada para el algoritmo reactivo o predictivo, de tal manera que puedan realizar los cálculos correspondientes.

```
1  public class StatusPE {
2
3      private String stream;
4
5      private long recibeEvent;
6      private long sendEvent;
7      private double sendEventUnit;
8      private double processEvent;
9      private long queueEvent;
10     private Queue<Double> history;
11     private Class<? extends ProcessingElement> pe;
12     private int replication;
13     private Queue<Integer> markMap;
14     private long eventCount;
15
16     public StatusPE() {
17         stream = null;
18         recibeEvent = 0;
19         sendEvent = 0;
20         sendEventUnit = 0;
21         processEvent = 0;
22         queueEvent = 0;
23         history = new CircularFifoQueue<Double>(100);
24         pe = null;
```

```
25          replication = 0;
26          markMap = new CircularFifoQueue<Integer>(2);
27          eventCount = 0;
28      }
29
30      public String getStream() {
31          return stream;
32      }
33
34      public void setStream(String stream) {
35          this.stream = stream;
36      }
37
38      public long getRecibeEvent() {
39          return recibeEvent;
40      }
41
42      public void setRecibeEvent(long recibeEvent) {
43          this.recibeEvent = recibeEvent;
44      }
45
46      public long getSendEvent() {
47          return sendEvent;
48      }
49
50      public void setSendEvent(long sendEvent) {
51          this.sendEvent = sendEvent;
52      }
53
54      public double getSendEventUnit() {
55          return sendEventUnit;
56      }
57
58      public void setSendEventUnit(double sendEventUnit) {
59          this.sendEventUnit = sendEventUnit;
60      }
61
62      public double getProcessEvent() {
63          return processEvent;
```

```
64      }
65
66      public void setProcessEvent(double processEvent) {
67          this.processEvent = processEvent;
68      }
69
70      public long getQueueEvent() {
71          return queueEvent;
72      }
73
74      public void setQueueEvent(long queueEvent) {
75          this.queueEvent = queueEvent;
76      }
77
78      public Queue<Double> getHistory() {
79          return history;
80      }
81
82      public void setHistory(Queue<Double> history) {
83          this.history = history;
84      }
85
86      public Class<? extends ProcessingElement> getPE() {
87          return pe;
88      }
89
90      public void setPE(Class<? extends ProcessingElement> pe) {
91          this.pe = pe;
92      }
93
94      public int getReplication() {
95          return replication;
96      }
97
98      public void setReplication(int replication) {
99          this.replication = replication;
100     }
101
102     public Queue<Integer> getMarkMap() {
```

```
103         return markMap;
104     }
105 
106     public void setMarkMap(Queue<Integer> markMap) {
107         this.markMap = markMap;
108     }
109 
110     public long getEventCount() {
111         return eventCount;
112     }
113 
114     public void setEventCount(long eventCount) {
115         this.eventCount = eventCount;
116     }
117 
118     @Override
119     public String toString() {
120         return "[PE : " + pe.toString() + " | Recibe: " + recibeEvent
121                 + " | Send: " + sendEvent + " | Replication: " + replication
122                 + "]";
123     }
124 
125 }
```

Código B.1: Clase StatusPE, el cual contiene las estadísticas de un PE específico.

En el Código B.2 se muestra la clase que almacena una arista con sus respectivos vértices del grafo, de esta manera, se posee un mapa del grafo, siendo utilizado para saber la topología que utilizó el usuario en el grafo. De esta manera, se puede tratar la replicación por parte de un operador a otro, viendo los distintos cambios que surgen en la topología del grafo.

```
1 public class TopologyApp {
2     private Class<? extends AdapterApp> adapter;
3     private Class<? extends ProcessingElement> peSend;
4     private Class<? extends ProcessingElement> peRecibe;
5     private long eventSend;
6 
7     public TopologyApp() {
8         adapter = null;
```

```
9           peSend = null;
10          peRecibe = null;
11          eventSend = 0;
12      }
13
14      public Class<? extends AdapterApp> getAdapter() {
15          return adapter;
16      }
17
18      public void setAdapter(Class<? extends AdapterApp> adapter) {
19          this.adapter = adapter;
20      }
21
22      public Class<? extends ProcessingElement> getPeSend() {
23          return peSend;
24      }
25
26      public void setPeSend(Class<? extends ProcessingElement> peSend) {
27          this.peSend = peSend;
28      }
29
30      public Class<? extends ProcessingElement> getPeRecibe() {
31          return peRecibe;
32      }
33
34      public void setPeRecibe(Class<? extends ProcessingElement> peRecibe) {
35          this.peRecibe = peRecibe;
36      }
37
38      public long getEventSend() {
39          return eventSend;
40      }
41
42      public void setEventSend(long eventSend) {
43          this.eventSend = eventSend;
44      }
45
46      @Override
47      public String toString() {
```

```
48          return this.adapter == null ? "[PE Send: " + peSend.toString() + " | PE
                Recibe: "
49                  + peRecibe.toString() + " | Event: " + eventSend + "]" : "[
                        Adapter: " + adapter.toString() + " | PE Recibe: "
50                          + peRecibe.toString() + " | Event: " + eventSend + "]";
51      }
52  }
```

Código B.2: Clase TopologyApp, el cual contiene las topología del grafo diseñado por el usuario.

ANEXO C. MODIFICACIONES AL CÓDIGO FUENTE DE S4

En el Código C.1 se presenta la implementanción de las tareas que están a cargo del envío de estadísticas al sistema de distribución de carga, donde una de ellas está a cargo de obtener las muestras para el historia, y la otra está a cargo de enviar las estadísticas de los PE existentes en el sistema. Además de esto, para la ejecución del sistema, se debe esperar que el *Adapter* esté ejecutándose, por lo que espera la notificación por parte de éste para la ejecución de las tareas.

```
1  private void startMonitor() {
2      if (runMonitor) {
3          synchronized (getBlockAdapter()) {
4              try {
5                  getBlockAdapter().wait();
6              } catch (InterruptedException e) {
7                  getLogger().error(e.getMessage());
8              }
9
10             ScheduledExecutorService getEventCount = Executors
11                     .newSingleThreadScheduledExecutor();
12             getEventCount.scheduleAtFixedRate(new OnTimeGetEventCount(),
13                     1000, 1000, TimeUnit.MILLISECONDS);
14
15             ScheduledExecutorService sendStatus = Executors
16                     .newSingleThreadScheduledExecutor();
17             sendStatus.scheduleAtFixedRate(new OnTimeSendStatus(), 6000,
18                     5000, TimeUnit.MILLISECONDS);
19         }
20     } else {
21         ScheduledExecutorService sendStatus = Executors
22                 .newSingleThreadScheduledExecutor();
23         sendStatus.scheduleAtFixedRate(new OnTimeSendStatus(), 6000, 5000,
24                 TimeUnit.MILLISECONDS);
25     }
26
27 }
```

Código C.1: Tareas que ejecutan el sistema de distribución de carga.

En el Código C.2 se muestra la implementanción que realizada para añadir una réplica a un PE en específico. El tipo *StatusPE* hace referencia un objeto creado en la implementanción, para almacenar los datos y estadísticas correspondientes al PE en el análisis de carga según el sistema de distribución de carga, como la cantidad de réplicas deseadas.

```
1  public void addReplication(StatusPE statusPE) {
2      for (Streamable<Event> stream : getStreams()) {
3          for (ProcessingElement PEPrototype : stream.getTargetPEs()) {
4              if (PEPrototype.getClass().equals(statusPE.getPE())) {
5                  for (long i = PEPrototype.getNumPEInstances(); i < statusPE
6                          .getReplication(); i++) {
7                      PEPrototype.getInstanceForKey(Long.toString(i));
8                  }
9              }
10         }
11     }
12 }
```

Código C.2: Añadir réplicas a un PE en S4.

En el Código C.3 se muestra la implementanción que realizada para eliminar una réplica a un PE en específico.

```
1  public void removeReplication(StatusPE statusPE) {
2      for (Streamable<Event> stream : getStreams()) {
3          for (ProcessingElement PEPrototype : stream.getTargetPEs()) {
4              if (statusPE.getPE().equals(PEPrototype.getClass())) {
5                  for (int i = statusPE.getReplication(); i < PEPrototype
6                          .getInstances().size(); i++) {
7                      ProcessingElement peCurrent = PEPrototype
8                              .getInstanceForKey(Integer.toString(i));
9                      peCurrent.close();
10                 }
11             }
12         }
13     }
14 }
```

Código C.3: Eliminar réplicas a un PE en S4.

ANEXO D. CONFIGURACIÓN PARA LA COMUNICACIÓN DE S4

En la tabla D.1 se muestra los parámetros utilizados para la configuración para la comunicación de S4. La descripción de cada uno de los parámetros está en el proyecto de S4 en la carpeta de comunicación.

Parámetro	Valor
s4.comm.emitter.class	org.apache.s4.comm.tcp.TCPEmitter
s4.comm.emitter.remote.class	org.apache.s4.comm.tcp.TCPRemoteEmitter
s4.comm.listener.class	org.apache.s4.comm.tcp.TCPListener
s4.comm.timeout	1000
s4.sender.parallelism	5
s4.sender.workQueueSize	10000
s4.sender.maxRate	10000
s4.remoteSender.parallelism	5
s4.remoteSender.workQueueSize	100000
s4.remoteSender.maxRate	10000
s4.emitter.maxPendingWrites	1000
s4.stream.workQueueSize	1000000

Tabla D.1: Parámetros de la configuración para la comunicación de S4.