UNIVERSIDAD DE SANTIAGO DE CHILE
FACULTAD DE INGENIERÍA
Departamento de Ingeniería Informática

Modelo elástico de replicación de operadores para un sistema de procesamiento de *stream* en tiempo real

Daniel Pedro Pablo Wladdimiro Cottet

Profesor guía: Nicolás Andrés Hidalgo Castillo
Profesor co-guía: Erika Susana Rosa 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

Me cuesta encontrar las palabras para expresar lo que siento, se cierra un ciclo en mi vida, y con ello una larga travesía de mucho esfuerzo y trabajo, donde espero que de inicio a muchos más. Recibí apoyo, cariño y paciencia de diferentes personas, las cuales subían y bajaban de este excéntrico tren en cada parada que hacíamos, y aunque a veces entre paradas se hacía un viaje muy corto, habían otras veces que se hacían unos viajes eternos, y no terminaba de distinguir realmente bien dónde iba, pero eso daba igual, porque siempre en cada trayecto aprendía y recibía algo de alguna persona, y eso me hizo ir creciendo con el tiempo, lo cual estoy eternamente agradecido. De antemano pido disculpas si me he olvidado de alguien, no fue mi intención, y espero que pueda agradecerlo en un tiempo futuro.

Aunque uno a veces siente que todo cambia; la gente, el barrio, la universidad, la sociedad, el trabajo, hay sentimientos y pasiones que nunca cambian, y son esas que te hacen vibrar y llenarte el corazón por estar ahí. Sin duda alguna, Izquierda Libertaria es una mis grandes pasiones, donde a través del FeL me brindó una escuela de lucha para poder construir un pueblo digno y soberano. No saben como agradezco estar ahí y conocer a mis amigas y amigos que además son compañeras y compañeros de lucha: Thomi, Neto, Cachorro, Sussan, Andrés, Zarri, Cata, Pato, Cristián, Tuto, Nati, Joaco, Fofi, de corazón gracias por todo su apoyo incondicional. Y más todavía agradezco al FeL porque aquí conocí a mi amada polola, quien no sólo le agradezco su apoyo, sino también por su cariño, alegría y paciencia, y sin duda todo sería muy distinto sin ti, no sabes lo agradecido y feliz que soy de estar con vos. Te amo Cami.

Cuando uno va viajando, vas creciendo, vas madurando, te vas dando cuenta que puedes ir resolviendo problemas que antes se hacían imposibles, van dándose nuevas herramientas, nuevas habilidades. Por lo mismo es que estoy completamente agradecido de la oportunidad que me brindaron mis queridos profesores guías Erika y Nicolás, porque confiaron en mí y se la jugaron por sacar este trabajo, y no sólo eso, de abrirme puertas para poder sustentarme y tener un porvenir más tranquilo. De verdad no saben lo orgulloso que estoy de tenerlos como profesores, no sólo me han ayudado a formarme como profesional, sino también como persona y eso se los agradezco desde el fondo de mi corazón. También agradezco a Pamela, por su apoyo y entrega incondiconal, y la profesora Carolina y el profesor Mauricio, que gracias a CITIAPS me sentí en mi segundo hogar, y siento que mucha de las cosas que aprendí hoy en día es gracias a esa pequeña salita de gran corazón. Y por lo mismo, no puedo olvidar a mi compañero de trabajo, Pablo, siempre me acuerdo como nos conocimos en ese primer día de las clases de Magíster, nunca creí que gracias a ese saludo pudiéramos tener este viaje, donde salió un paper, un proyecto y una bonita amistad, de verdad gracias Pablo, sos un hermano para mí. Gracias a todos los que pasaron y están en CITIAPS, Álvaro, Jeff, Diego, Farisori, no me olvidaré de ustedes, porque sin duda me dejaron marcado y me ayudaron a formarme como profesional y persona. También a mis amigas y amigos de la Usach, Miguel, Gabo, Clau, Karla, Jorge, los quiero caleta y gracias por todo, porque fueron un pilar en mi vida universitaria, donde no los olvidaré, porque todo lo que viví con ustedes fue una de mis grandes alegrías.

Pero a veces uno se queda dormido, recuerdas el pasado, y te acuerda que eso fue lo que te formó y lo que explica el cómo eres hoy en día. Nunca los voy a olvidar, para mi son mi fuerza, mis ganas, mi apañe, quizá todos estamos en nuestras vidas, pero los tengo mas presente que nunca, José, Eduardo, Rubén y Felipe, los quiero demasiado. Y como te voy a olvidar, si sos mi hermano de leche, Simón, no sólo te tengo que agradecer, te debería hacer una oda por lo grande que has sido conmigo. Tantas cosas que pasamos, tanto que vivimos, tantos recuerdos, y como fuimos creciendo, donde sea que estés tengo claro que puedo contar contigo, y estoy seguro que tú sientes lo mismo de mí.

Y alguien tuvo que crear la primera estación del viaje, alguien tuvo que armar ese tren, y es algo que aunque suba y bajen mil personas, pasen mil años, nunca podrás borrarlo, y me alegro que sea así, porque es lo más valioso que uno tiene, la familia. Como no quererlas, si me mimaron, me criaron, me abrazaron, me apoyaron, y eso lo agradezco, a ti Amandi, a ti Sofi y a ti Mamá, las amo. Obvio, no me voy a olvidar de ti, gracias por todo tu apoyo Papá, siento que mucho de lo que aprendí fue gracias a ti, y me alegro haber tenido esa oportunidad.

Agradezco también al PMI-USA 1204 y al proyecto DICYT-USACH 061419HC por darme la tranquilidad de poder trabajar en este largo proyecto.

TABLA DE CONTENIDO

ÍNDICE DE TABLAS

ÍNDICE DE ILUSTRACIONES

ÍNDICE DE ALGORITMOS

RESUMEN

En el mundo actual de la información, grandes cantidades de datos son generados cada segundo desde las más diversas fuentes: redes sociales, redes de sensores, buscadores Web, entre otros. Extraer información de dichos datos muchas veces requiere que este procesamiento sea llevado a cabo en tiempo real, debido a que el análisis que se debe realizar depende de la temporalidad en que son generadas los eventos. Para lograr procesar grandes cantidades de datos con estas restricciones, existen sistemas especializados llamados sistemas de procesamiento de *stream* (SPS), los cuales pueden procesar en tiempo real los datos que van llegando por una o más fuentes de datos. Estos sistemas están basados en grafos, cuyos vértices realizan operaciones sobre un flujo de datos reflejado por las aristas del grafo. La topología del grafo le brinda flexibilidad al SPS para generar diversas aplicaciones de procesamiento, sin embargo, dicha topología es estática una vez el sistema se ejecuta. Dado este problema, este trabajo se plantea un modelo elástico que sea capaz de adaptar la topología del grafo a las condiciones del tráfico existente. Para esto se ha diseñado un algoritmo reactivo, usando la técnica de fisión, y otro predictivo, usando cadena de Márkov, ambas técnicas permiten estimar la carga de los operadores, y adaptar el grafo acorde a lo indicado por estos. Dicha modificación consiste en incrementar o disminuir la cantidad de réplicas de un operador según su nivel de carga. Los resultados obtenidos de los experimentos realizados en el SPS S4, muestran una mejora de hasta nueve veces más eventos procesados, con un costo asociado a un aumento de $0,01\%$ del uso de la CPU, pero una disminución de un $1,5\%$ en el consumo de memoria RAM.

Palabras Claves: SPS; Elasticidad; Distribución de carga; Balance de carga; Algoritmos reactivos; Fisión; Algoritmos predictivos; Cadena de Márkov

ABSTRACT

Nowadays, information generated by the Internet's interactions is growing exponentially, creating massive and continuous flows of events from the most diverse sources. These interactions contain valuable information for domains such as government, commerce, and banks, among others. Extracting information from such data requires powerful processing tools to cope with the high-velocity and high-volume stream of events with near real-time results. Specially-designed distributed processing engines build a graph-based topology of a static number of processing operators creating bottlenecks and load balance problems when processing dynamic flows of events. In this work we propose a self-adaptive processing graph that provides elasticity and scalability, increasing or decreasing automatically the number of processing operators to improve performance and resource utilization. Our solution uses a model that monitors, analyzes and changes the topology of the graph with a control algorithm that is both reactive and proactive to the flow of events. We have compared our solution with a baseline approach and results show that our system improves performance in terms of the number of processed events at a very low cost.

Keywords: SPS; Elastic; Load balancing; Reactive Algorithm; Fision; Predictive Algorithm; Markov chain

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, como en blogs, redes sociales u otras aplicaciones Web (Oberhelman, 2007).

Con el paso del tiempo, más y más información es generada por distintas interacciones generadas por los usuarios. Por lo que analizar o extraer esta información no es una tarea fácil, más aún cuando muchas de estas interacciones deben ser analizadas en tiempo real, dada su dependencia temporal. Por esta última característica es que sistemas tradicionales de procesamiento basados en MapReduce (Lin & Dyer, 2010) o *bash processing* (Hawwash & Nasraoui, 2014) no son ideales para el análisis de esta información.

Es así como con el tiempo se han ido creando distintos sistemas de procesamiento capaces de lidiar con las restricciones de temporalidad, 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). El requisito de procesar información en tiempo real surge por la necesidad de los usuarios en obtener 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. Este tipo de aplicaciones son necesarias para sus usuarios, debido que proveen de información actualizada que permite mejorar entre otros casos la toma de decisiones (Wenzel, 2014).

Un ejemplo de esto son las aplicaciones que analizan redes sociales en caso de un desastre natural, donde grandes cantidades de información son generadas, y se requiere procesar esta información lo más cercano al tiempo real para obtener información que permita un trabajo de recuperación más eficiente dado el suceso (Andrade et al., 2014). De esta manera, se puede construir un sistema que procese los datos realizando un análisis de la percepción de la gente, búsqueda de personas desaparecidas o focos de necesidad. Con esta información, se puede establecer sectores críticos, facilitar la búsqueda de personas, distribución de alertas, o detección de necesidades, lo cual es crucial para tomar decisiones en esos momentos.

Por otra parte, también son utilizados para llevar a cabo predicciones en la bolsa de comercio. De esta manera, se crean sistemas de procesamiento que apliquen modelos matemáticos, los cuales permiten predecir el comportamiento para el siguiente día en el mercado. 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.

También se aplica en casos de seguridad en redes, donde se permite realizar un monitoreo de las actividades ocurridas en la red. Como la información es procesada en tiempo real, ayuda a detectar a tiempo las posibles acciones de usuarios maliciosos. Dentro de las aplicaciones que existen sobre este tema, son el análisis de los logs de la red cerrada, donde con esta información se puede verificar si existe algún *bug*, error o anomalía, además de ver si existe algún intruso o violación al sistema.

Para dar soporte a estas aplicaciones existen los SPS (Sistemas de Procesamiento de *Stream*) tales como S4 (Neumeyer et al., 2010), Storm (Storm, 2014), Samza (Samza, 2014), entre otros. El paradigma de procesamiento de estos sistemas se basa en grafos, donde los vértices son operaciones realizadas al flujo de datos, representadas por las distintas aristas. Para la generación de una aplicación, el usuario debe diseñar una topología de procesamiento compuesto por las tareas u operaciones deseadas. Cada grafo o topología tiene un *input* desde una fuente de datos, y un *output* del flujo de salida proveniente del último operador. Aunque poseen bastante flexibilidad para la creación de diversas aplicaciones, por la facilidad de crear distintas topologías, no lo tiene para adaptarse con el tiempo a las condiciones del tráfico entrante cuando se encuentran en funcionamiento, esto debido a que las topologías de procesamiento generadas son estáticas en tiempo de ejecución. Dada la naturaleza dinámica de las interacciones, pueden surgir problemas de distribución de carga en la topología asociada a la aplicación.

El problema de sobrecarga conlleva a una baja en el rendimiento, produciendo una pérdida de recursos, tiempo e información. Abordar este problema es crítico, puesto que al realizar una optimización en el sistema, implica un aumento en la cantidad de datos procesados, y una mayor precisión en los resultados por parte de la aplicación.

Lo anterior lo podemos entender de mejor manera con el siguiente ejemplo: se posee un tiempo t para procesar n datos. Si se aumenta la cantidad de datos procesados, se tiene que en el mismo tiempo t se procesan una cantidad $n + m$ de datos, donde m son los datos adicionales a analizar debido a la mejora del rendimiento. Como existe un aumento en la cantidad de datos procesados, la información obtenida puede ser más precisa, dado que se posee una mayor cantidad de datos. Por ejemplo, al procesar una mayor cantidad de transacciones en la bolsa de comercio, se puede poseer una predicción más precisa de cómo se comporta la bolsa a futuro. Desde otro punto de vista, se efectúa una mejora en los recursos utilizados, habiendo una disminución de los recursos ociosos.

1.2 DESCRIPCIÓN DEL PROBLEMA

Dado el carácter estático del grafo de procesamiento en tiempo de ejecución y el carácter altamente dinámico del tráfico, puede surgir problemas de balance de carga entre los operadores de la topología, sobrecargando alguno de estos y comprometiendo el rendimiento del sistema.

1.3 SOLUCIÓN PROPUESTA

La solución propuesta consiste en un modelo elástico de replicación de operadores para los sistemas de procesamiento de *stream*, el cual permite adaptar el grafo de procesamiento a las variaciones del tráfico. Para esto se ha implementado cuatro módulos que componen la estructura del modelo elástico: 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 analizador y predictor de carga, los cuales están encargados de medir el nivel de carga del operador, y según eso modificar la cantidad de réplicas necesarias. Cada uno de estos módulos trabaja de forma independiente y poseen distintos enfoques: reactivo y predictivo.

El analizador de carga aplica un enfoque reactivo, el cual analiza el tráfico de los operadores en el tiempo actual y cuantifica su carga. El estado de la carga de cada operador depende de un umbral, por lo que según éste se solicita al administrador de réplica incrementar o disminuir la cantidad de réplicas del operador.

El predictor de carga aplica un enfoque predictivo, el cual analiza la carga de los distintos operadores en una ventana de tiempo y predice la carga para la siguiente ventana. Con esta información el administrador de réplicas determina la mejor configuración de los operadores para dicho período.

El administrador de réplicas por su parte se alimenta de la información entregada por los dos módulos anteriores, y en base a esto, toma una decisión respecto a los recursos asignados al operador. En otras palabras, verifica cuántas réplicas son necesarias según el tamaño del tráfico.

1.4 OBJETIVOS Y ALCANCE DEL PROYECTO

1.4.1 Objetivo general

Diseño, construcción y evaluación de un modelo elástico de replicación de operadores para un sistema de procesamiento de *stream* en tiempo real.

1.4.2 Objetivos específicos

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

1.4.3 Alcances

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

1. La evaluación de la solución se ha implementado sobre un solo sistema de procesamiento de *stream*.
2. Los datos emitidos de la fuente de datos son homogéneos, teniendo una tasa de procesamiento constante para cada operador.
3. La distribución de flujo de datos es a nivel de operadores y no de máquinas, por lo que no se ha analizado la carga de estos últimos.

4. Los algoritmos propuestos no incluyen técnicas que garanticen el procesamiento de todo el flujo de datos.
5. En la evaluación de los algoritmos propuestos se ha considerado 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 del trabajo propuesto, se ha utilizado el método científico para la realización de éste. Dentro de las etapas propuesta por (Sampieri et al., 2010) están:

1. Formulación de la hipótesis: "La utilización de un modelo elástico en un sistema de procesamiento de *stream* permitirá aumentar la cantidad de eventos procesados, y con ello la precisión de los resultados".
2. Elaboración del marco teórico: Exponer los trabajos que existen sobre problemas de carga en los operadores de SPS. Así mismo, los conceptos fundamentales de estos sistemas.
3. Seleccionar el diseño apropiado de investigación: Diseñar el modelo elástico para el problema de balance de carga a nivel lógico en un SPS, vale decir, los distintos módulos, donde se posea un algoritmo reactivo y otro predictivo. Cada diseño y ejecución de los experimentos se basan en los principios de un SPS definiendo métricas acordes para dicho modelo de procesamiento.
4. Analizar los resultados: De debe analizar los resultados según las métricas establecidas 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 ha utilizado Apache S4 0.6.0, por lo que es necesario para su configuración Java SE Development Kit 7. Dentro esto, el lenguaje de programación de cada uno de los módulos del modelo desarrollado se ha utilizado el lenguaje de programación Java, y se ha trabajado sobre el IDE Eclipse Standard 4.4.2. De forma complementaria, se ha utilizado Texmaker 4.1 para la confección de los distintos informes requeridos y la documentación correspondiente al trabajo.

1.6 ORGANIZACIÓN DEL DOCUMENTO

En el presente documento se divide en seis capítulos. En este capítulo se presenta la problemática y la solución propuesta, en conjunto con los objetivos y la metodología utilizada. En el segundo capítulo se exponen los conceptos teóricos involucrados. Posteriormente, el tercer capítulo aborda los distintos enfoques y técnicas que se han brindado en la literatura para dar soluciones al problema planteado. Luego, el cuarto capítulo se describe el diseño de los algoritmos utilizados en el modelo propuesto, explicando las distintas decisiones que se han tomado para el diseño de éste. En el quinto capítulo se presentan los distintos experimentos realizados para evaluar el sistema diseñado, donde se explica su implementación y evaluación según los experimentos diseñados. Finalmente, en el sexto capítulo se exponen las respectivas conclusiones obtenidas a partir del presente trabajo.

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. En caso de estar ocupado el servicio, se dejan los datos en cola (Menin, 2002). 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 del 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 sistema, se guarda en un *buffer* los datos para posteriormente ser procesados.

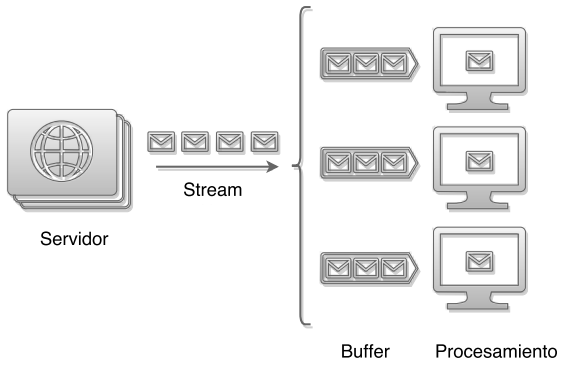

Figura 2.1: Flujo de datos entre el servidor y los clientes.

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 la *Streaming API* proporcionada por Twitter, donde esta información se puede utilizar para estudiar los *tweets*, 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, 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*. Para el funcionamiento correcto de estas aplicaciones, es necesario cumplir con ciertas características. (Andrade et al., 2014) establece los requerimientos para el procesamiento continuo de datos, estos se desglosan 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 sistemas de *bash processing*. Por lo tanto es necesario otro mecanismo que pueda procesarlos mientras va llegando la información entrante. *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 existente. Esto es importante, dado que no sirve un sistema de estimación de la bolsa del mercado si es que presenta alta latencia. 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. Si analizamos el funcionamiento de los sistemas, estos poseen un porcentaje de fallas, y los SPS no son la excepción, por ello es importante contar con un mecanismo de tolerancia a fallos que permita reducir la pérdida de información. De no existir, se puede perder información, comprometiendo la precisión de los resultados y requiriendo de un mayor tiempo para recolectar la información perdida o alcanzar un estado similar.

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, para posteriormente procesarlos de manera *offline* (Hawwash & Nasraoui, 2014). Este cambio de paradigma implica el análisis de datos sin requerir su almacenamiento, de esta manera los datos fluyen mientras son procesados.

El paradigma utilizado se basa en grafos de procesamiento como muestra la Figura 2.2, donde los operadores (o tareas de procesamiento) 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 redes sociales, estadísticas del monitoreo de un sistema, o transacciones en la bolsa de comercio, la cual entrega los datos iniciales a los primeros operadores del grafo (Appel et al., 2012).

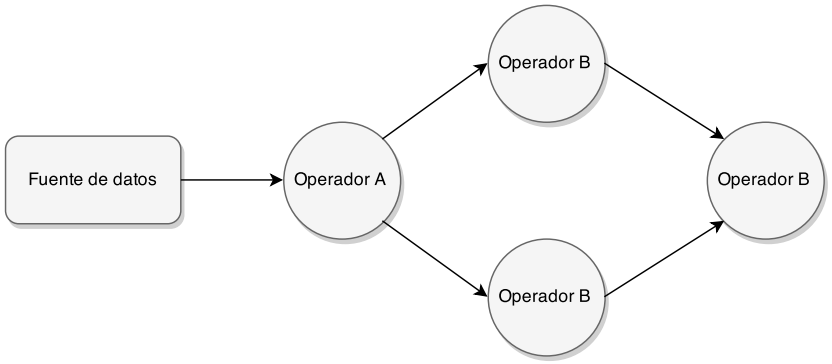

Figura 2.2: Ejemplo de modelo de SPS.

Cabe destacar que los SPS son distribuidos, es decir, 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*. Para lograr la comunicación entre los operadores, se utilizan sistemas anexos especialmente diseñados para este tipo de tareas, como Apache ZooKeeper (Hunt et al., 2010). Este último consiste en un servicio centralizado que mantiene información de configuración y sincronización de las aplicaciones distribuidas que se posean. Para esto, cada nodo de procesamiento debe registrarse en el sistema, y es el servidor central el que posteriormente se encarga de señalar los nodos disponibles para el procesamiento y distribuir

los eventos entre ellos.

Las principales aplicaciones que se le dan a estos SPS, están orientadas al manejo de grandes cantidades de datos, las cuales deben ser procesadas 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) establece los siguientes requerimientos:

- **Baja latencia**: este concepto está asociado con la comunicación fluida entre los distintos nodos del sistema, de tal manera que no existan altos *delay* o retrasos en el procesamiento.
- **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 que se pueda unir las respuestas entregadas de forma paralela. Dado lo anterior, se hace necesario diseñar un sistema que cumpla con operadores adicionales a los utilizados en las consultas tradicionales por sistemas centralizados.
- **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 determinístico 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 explicado anteriormente, en general se trabaja con datos heterogéneos, por lo que se requiere estandarizarlos para su procesamiento, de tal manera que no exista una discordancia en la información procesada.
- **Garantizar la seguridad y disponibilidad de los datos**: este requerimiento está orientado a disponer de mecanismos de *checkpoint*, técnica utilizada para respaldar el estado del operador cada cierto período de tiempos. Por lo que en caso de existir alguna falla, el sistema pueda volver a estar disponible 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 los distintos procesadores o máquinas, deseando idealmente una escalabilidad incremental. Esto significa que el flujo de datos sea entregado a los distintos recursos que se poseen, y en caso de necesitar más recursos incrementarlos (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 de bajo *overhead*, es decir con bajo costo de implementación o recursos necesarios para su funcionamiento, aumentando así la eficiencia y el rendimiento del sistema.

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

El primer modelo llamado *push* consiste en el envío de datos desde el operador al próximo operador según la topología del grafo. 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 el Operador A envía los datos al Operador B, donde en caso que el Operador B esté procesando un dato, éste lo guarda en cola. Dada la forma en como se realiza el envío, éste no asegura que llegue el dato, debido a que no existe una respuesta por parte del operador receptor. Por lo tanto, existe una abstracción en el envío de los eventos por parte del operador receptor, debido que no es necesario saber de quién es el operador receptor.

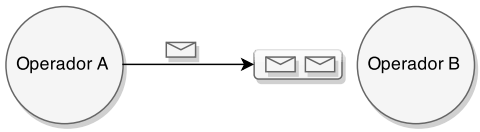

Figura 2.3: Modelo push de procesamiento.

Por otra parte, el segundo modelo llamado *pull*, se basa en la petición de datos a un operador, por lo que son enviados solo sí 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 qué 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 (a) se observa que existen dos

operadores, donde se solicita por parte del Operador B el envío de un dato para ser procesado, para que posteriormente en la Figura 2.4 (b) el Operador A envía el dato para que posteriormente sea procesado por el Operador B.

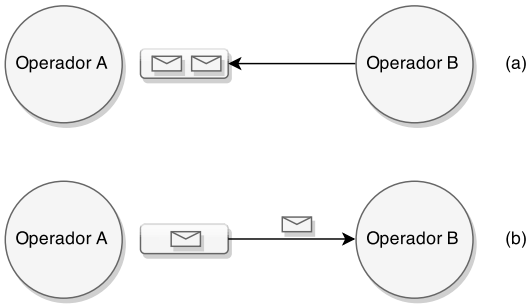

Figura 2.4: Modelo pull de procesamiento.

2.3.1 S4

S4 (Simple Scalable Streaming System) (Neumeyer et al., 2010) es un sistema de propósito general, distribuido y escalable que permite diseñar aplicaciones para procesar flujos de datos de forma continua y sin restricciones, el cual está inspirado en MapReduce (Lin & Dyer, 2010). Cada evento en S4 es descrito como un par (clave, atributo). La unidad básica son los elementos de procesamiento (PEs, por sus siglas en inglés). Los PEs pueden emitir o pueden publicar resultados y son alojados en servidores llamados nodos de procesamiento o 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. Para este fin, en la capa de comunicación S4 utiliza Apache ZooKeeper (Hunt et al., 2010), el cual provee manejo de *clusters* y reemplazo automático de nodos que fallan (*failover*). 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, orientada a la computación de flujos de datos en tiempo real 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, el cual es llamado operador. Transformaciones complejas requieren múltiples *Bolts*, los cuales crean topologías o grafos. La plataforma provee de 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 su almacenamiento. Sin embargo, esta base de datos es su mayor desventaja respecto de S4 puesto que no es completamente distribuida. Storm define diferentes técnicas para el particionamiento de los *streams* de datos y para la paralelización de *Bolts*, por lo tanto la asignación de máquinas para alguna actividad debe efectuarse de forma manual, lo que complica el desarrollo de aplicaciones. Al igual que S4, Storm usa Apache 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* es la capacidad que el sistema tiene de adaptarse dinámicamente a las condiciones variables del sistema, como por ejemplo el tráfico. Esto quiere decir que aumenten o disminuyan los recursos por parte del sistema según la necesidad que éste posea, de tal manera que funcione de manera eficiente (Goetsch, 2014)

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.

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

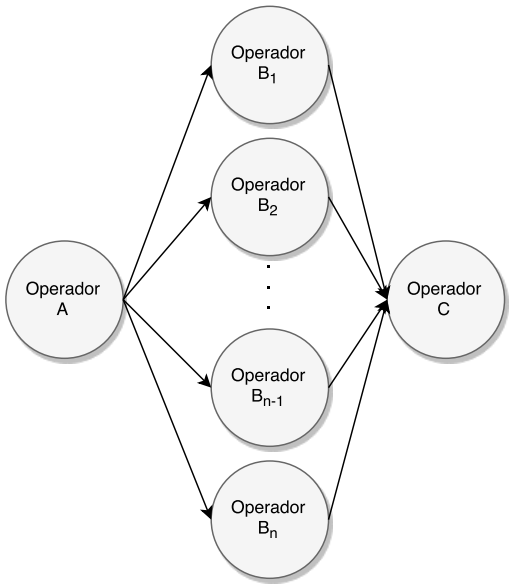

Figura 2.5: Elasticidad en un SPS.

En el trabajo realizado, se propone un sistema elástico basado en la carga de los operadores. De esta manera, según el tráfico recibido se aumentan o disminuyen el número de los operadores, de tal manera de mantener un rendimiento eficiente en el tiempo, disminuyendo el riesgo de sobrecarga y con ello la pérdida de datos.

2.5 MAPE

MAPE (Monitor-Analyze-Plan-Execute) es un modelo basado en módulos, cuyo objetivo es la adaptabilidad del sistema según condiciones externas (Kephart et al., 2005). Para el manejo automático de los recursos del sistema, se propone implementar un control inteligente bajo ciclos.

En la Figura 2.6 se muestra una arquitectura diseñada bajo cuatro módulos para la administración automática del sistema:

- Monitor: Este módulo provee los mecanismo de recolección, añadiendo filtros o algún operador necesario para la administración de los recursos.
- Análisis: Este módulo utiliza modelos matemáticos, ya sea series de tiempo o modelos de colas, para el análisis de la información entrante. Con esto mismo, se espera realizar un aprendizaje del sistema, para posteriormente ayudar en la predicciones futuras por parte del modelo matemático.

- Planificación: Este módulo designa las acciones necesarias para cumplir con los objetivos y las metas necesarias del sistema. En general, se diseñan políticas para la planificación de las tareas.
- Ejecución: Este módulo está encargado en ejecutar la planificación elaborada según las necesidades dinámicas del sistema.

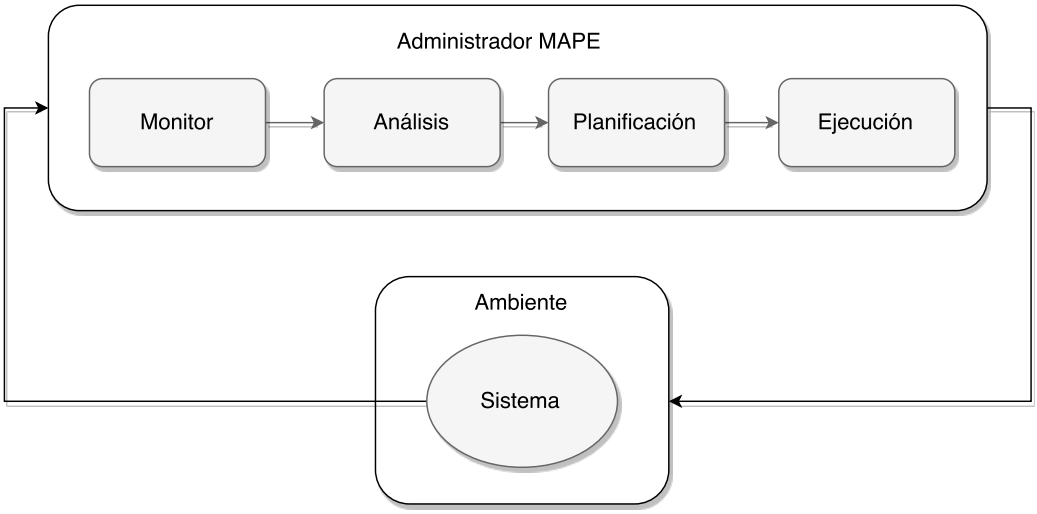

Figura 2.6: Ejemplo de un sistema con un administrador basado en MAPE.

2.6 PROCESOS ESTOCÁSTICOS

Se define un 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 se comporta de cierta manera en el tiempo, sin poseer un proceso determinístico entre sus variables, es decir, que las variables dependan de la historia (Taylor & Karlin, 2014).

Esto permite definir un estado como el posible comportamiento que puede tener una variable aleatoria en el sistema. Para ejemplificar, considera un modelo que contemple tres estados: estable, inestable y ocioso, y 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, donde cada uno representa un comportamiento del sistema (De Sapio, 1978).

Las cadenas de Markov son procesos estocásticos, las cuales han sido utilizadas para permitir la predicción de carga en modelos como el propuesto por (Gong et al., 2010). En

este trabajo se definen estados que son independientes en el transcurso del tiempo, con el fin de realizar análisis a futuro tomando en consideración los datos *a priori*.

2.6.1 Cadena de Markov

Sea X_t el valor de una variable aleatoria X en un tiempo t, donde el conjunto de todos los valores posibles para X se denominada espacio de estado (Ching & Ng, 2006). La variable aleatoria es un proceso de Markov, si sólo y si las probabilidades de transición entre dos estados 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 2.7. Cabe destacar que este tipo de proceso es un caso específico de los procesos estocásticos.

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

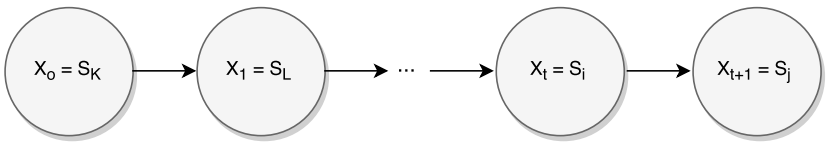

Figura 2.7: Proceso de Markov.

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

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

Donde en la Ecuación 2.3 se define la cadena bajo las probabilidades de transición existentes en éste. En la Figura 2.8 se muestra un ejemplo de la transición del estado i al estado j, dada la probabilidad P_{ij}.

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

En la Ecuación 2.4 se presenta una matriz de transición de estados finitos, donde P_ij la probabilidad de pasar de un estado a otro. Las probabilidades son tales que la suma de todas las transiciones de un estado a otro debe ser igual a 1.

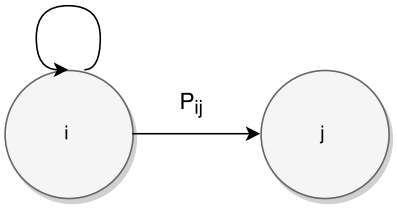

Figura 2.8: Cadena de Markov.

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

En la Figura 2.9 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 el del caso actual, tal como es definido en los procesos estocásticos. Las probabilidades de transición de un clima a otro, se pueden ver en la Ecuación 2.5.

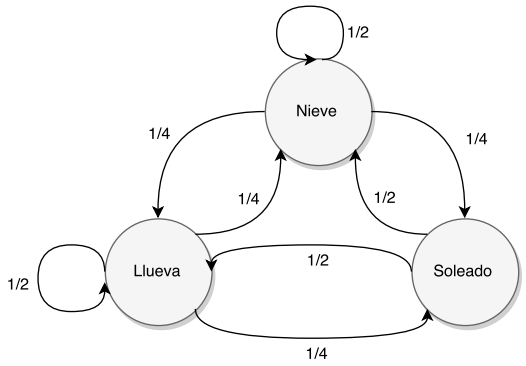

Figura 2.9: Ejemplo de cadena de Markov.

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

Dentro de las cadenas de Markov existen clasificaciones, las cuales son requisitos para realizar cálculos como la distribución estacionaria. Una de ellas son las cadenas irreductibles, esto significa que todos sus estados son accesibles entre ellos, por lo tanto existe una posible transición de un estado a otro, habiendo una única clase de estados, por lo tanto, todo estado debe cumplir con la Ecuación 2.6. En la Figura 2.10 se presenta un ejemplo de cadena

reductible, donde no existe trancaron posible luego de llegar al estado *Lluvia*. A diferencia de la cadena de la Figura 2.9, donde todos los estados pueden realizar una transición a otro estado.

$$P_{ij}^{(n)} = P(X_n = j | X_0 = i) > 0 \tag{2.6}$$

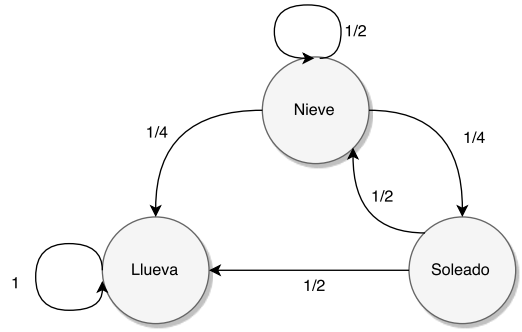

Figura 2.10: Ejemplo de cadena de Markov reductible.

Una de las definiciones atribuidas a un estado es la periodicidad, esto significa que es la cantidad de transiciones necesarias para volver al mismo estado desde donde se inicio la transición original, definida por el valor d. En la Ecuación 2.7 indica la condición para cumplir con la periodicidad, donde d es el mayor valor posible de n, siendo este último el índice de la transición realizada. En el caso que la cantidad de transiciones sea mayor a 1 significa que el estado es periódico, y en el caso que sea 1 se denomina aperiódico. En la Figura 2.9 se presenta una cadena con dos estados aperiódicos y uno periódico, debido que el estado *Soleado* posee $d = 2$, dado que para volver a si mismo debe realizar dos transiciones en la cadena.

$$P_{ii}^{(n)} = P(X_n = i | X_0 = i) > 0; n = d, 2d, 3d, ..., m \cdot d \tag{2.7}$$

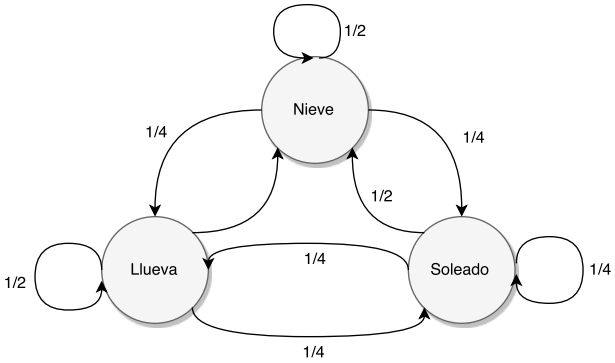

Figura 2.11: Ejemplo de cadena de Markov con estados aperiódicos.

Sea $F(i, i)$ la probabilidad que se posee que después de k transiciones se retorne al estado i, el cual se define según la Ecuación 2.8. En el caso que $F(i, i) = 1$ se define este estado

como recurrente, y en caso contrario, es transitorio. En el caso que se posea una cadena con un número finito de estados y éste sea recurrente, entonces éste es recurrente positivo. De haber un número infinito de estados, entonces el estado es recurrente nulo.

$$F(i,i) = \sum_{k=1}^{\infty} F_k(i,i) \tag{2.8}$$

Por otra parte, 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 & Pillai, 1984), la cual se muestra en la Ecuación 2.9.

$$\begin{aligned} \Pi_i(t+1) &= P_r(X_{t+i} = S_i) \\ &= \sum_k P_r(X_{t+i} = S_i / X_t = S_k) \cdot P_r(X_t = S_k) \\ &= \sum_k P_r(X_{t+i} = S_i / X_t = S_k) \cdot \Pi_k(t) \end{aligned} \tag{2.9}$$

Y en notación matricial en la Ecuación 2.10.

$$\begin{aligned} \Pi_{(t+1)} &= \Pi_{(t)} P \\ \begin{bmatrix} \Pi_1 & \Pi_2 & \Pi_3 \end{bmatrix}_{(t+1)} &= \begin{bmatrix} \Pi_1 & \Pi_2 & \Pi_3 \end{bmatrix}_{(t)} \begin{bmatrix} P_{1,1} & P_{1,2} & \cdots & P_{1,n} \\ P_{2,1} & P_{2,2} & \cdots & P_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ P_{n,1} & P_{n,2} & \cdots & P_{n,n} \end{bmatrix} \end{aligned} \tag{2.10}$$

En caso que la cadena sea irreducible y sus estados sean recurrentes positivos aperiódicos, entonces se puede calcular la distribución estacionaria como se muestra la Ecuación 2.11, la cual indica el comportamiento a futuro de la cadena de Markov, dado los estados y transiciones que éste posee.

$$\begin{aligned} \Pi(t) &= \Pi(t-1)P \\ &= \Pi(t-2)P^2 \\ &= \Pi(0)P^t \end{aligned} \tag{2.11}$$

Donde $\Pi(0)$ es la distribución inicial de la cadena.

2.6.2 Trabajo relacionado

Existen modelos predictivos que simulan el comportamiento del sistema, ya sea del flujo o de la carga de un operador, de tal manera que pueden predecir cual es su estado en un tiempo futuro. En general, para poder realizar una predicción se analizan las variables 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). Este sistema orientado a *Cloud Computing*, analiza la cantidad de recursos disponibles, ya sea la memoria disponible o el uso promedio de CPU en las máquinas virtuales que se disponen 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 ver si posee correlación con algún estado de la cadena de Markov, para ver la transición de ese estado a otro y construir la matriz de transición. Posteriormente, con la ecuación de Chapman-Kolmogorov, se calcula la distribución estacionaria de la matriz de transición, de tal manera de saber en qué estado está en la próxima ventana de tiempos 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 el número de máquinas virtuales de forma dinámica en un *Cloud*. Este trabajo aplica la transformada de Fourier (Falk et al., 2012) a series temporales obtenidas mediante el muestreo de la carga de CPU en una ventana de tiempo. Luego, la función resultante se analiza con distintas frecuencias, de tal manera de determinar la predicción de la próxima ventana de tiempo a cada una de las funciones creadas. De esta manera, se sintetizan todas las 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.

2.7 TEORÍA DE COLAS

La teoría de colas se centra en el estudio matemático de las colas existentes en un sistema, cuyo caso de estudio es el *performance* del servidor según el número de clientes que se posea (Cooper, 1972). En la Figura 2.12 se muestra un ejemplo de un sistema basado en teoría de colas, donde existen n productores que envían tareas a m servidores disponibles, y en caso

de no estar disponibles se genera una cola de espera en el sistema. Los elementos del sistema son:

- **Productor**: es quién provee la fuente de datos de entrada para el servidor.
- **Cola o línea de espera**: la cual está encargada de almacenar provisionalmente 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 de entregar una fuente de salida con la información procesada.

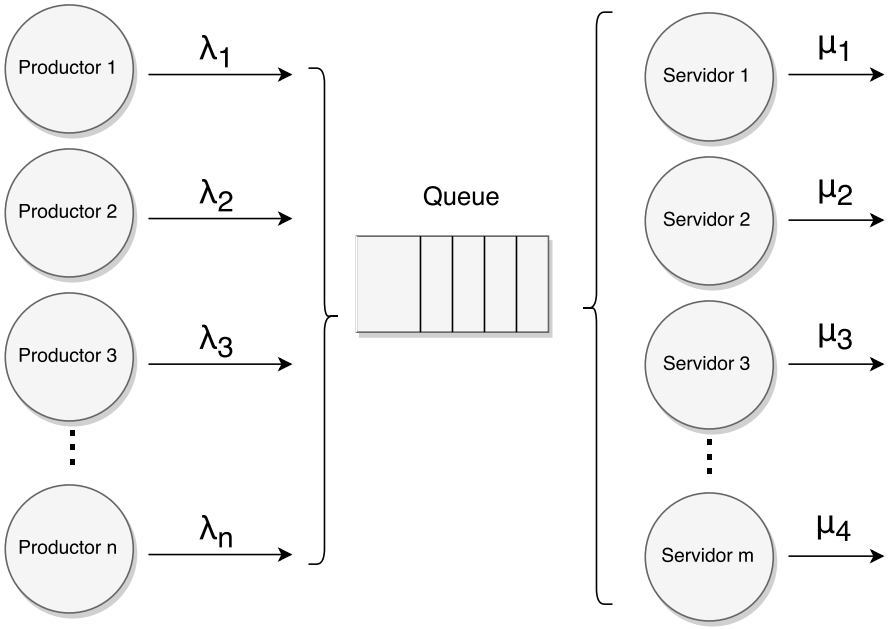

Figura 2.12: Ejemplo de un sistema basado en teoría de colas.

Otros componentes importantes en el sistema, son definidos a continuación:

- **Tasa de llegada**: denotado λ, es la cantidad de datos, eventos o información que van llegando en un determinado período de tiempo, la cual está determinada por los productores que existen en el sistema.
- **Tasa de procesamiento**: denotado μ, también llamada tasa de servicio, es la cantidad de datos, eventos o información que salen del sistema, producto del procesamiento provisto por cada servidor.
- **Tasa de rendimiento**: denotado ρ, es el porcentaje de utilización del sistema, definido como $\rho = \frac{\lambda}{s\mu}$, siendo s la cantidad de servicios disponibles, definiendo así un sistema estable si es que $\rho < 1$, dado que la capacidad de procesamiento es mayor que la tasa de llegada.

- **Disciplina de la cola**: Política utilizada para extraer los datos encolados en el sistema. Alguna de las políticas tradicionales son: *FIFO*, *LIFO*, *RSS*, entre otros.

Este tipo de modelo se puede aplicar a los SPS de manera directa, debido a que el operador emisor o la fuente de datos es el productor, y el operador receptor es el servidor del sistema. Sin embargo existe un problema interesante a analizar asociado a las a la carga de los operadores. Por ejemplo, si se tiene un operador con una tasa de llegada λ y una tasa de servicio μ, donde $\mu < \lambda$, se tiene un sistema inestable, debido que se procesa más lento de lo que llegan los datos. Esto genera colas por lo que es necesario un aumento del rendimiento del sistema, por lo que es necesario modificar la cantidad de operadores.

CAPÍTULO 3. BALANCE DE CARGA EN SPS

3.1 PERSPECTIVAS DE BALANCE DE CARGA

Dentro de la literatura se ha observado distintas perspectivas del problema de balance de carga en un SPS, las cuales consideran los recursos físicos como fuente del problema, o los recursos lógicos como el foco del mismo.

3.1.1 Recursos físicos

En esta perspectiva se toma en consideración la sobrecarga del sistema dado las limitantes físicas de éste, ya sea por condiciones de los recursos disponibles o por el ambiente de desarrollo. Las estrategias de balance de carga que se presentan están basados en el concepto de umbrales de recursos, es decir, cuando estos umbrales son alcanzados se aplica una estrategia para aliviar la carga. 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), entre otros.

Una de las soluciones con la perspectiva anterior es Borealis (Xing et al., 2005). Borealis considera la cantidad de carga de los nodos en ventanas de tiempo pre-determinadas, las cuales son monitoreadas por un coordinador centralizado. Este coordinador se encarga de analizar los recursos del sistema, y en caso que se sobrepase el umbral propuesto, se aplica la migración de los operadores del nodo hacia a otro nodo candidato cuya carga es menor.

Si bien esta estrategia realiza una balance de carga en cuanto a las máquinas físicas, no aumenta el rendimiento dado los recursos lógicos disponibles, dado que sólo mueve el operador sobrecargado de una máquina a otra. En el trabajo anterior, para elegir el nodo candidato se realiza una análisis de correlación existente entre el operador y el nodo candidato, de tal manera de no sólo enviarlo al nodo con menor carga, sino el que entre menor latencia en la interacción con el operador. Uno de los problemas existes en este sistema radican en la conexión entre los distintos nodos, por lo que para las pruebas se ha considerado un alto ancho de banda, de tal manera que aparente una red sin limitaciones de este tipo.

Por otra parte, Flood (Alves et al., 2010) propone un DPS (*Distributed data stream processing*) que considera factores físicos para agregar o eliminar máquinas virtuales en Amazon EC2 (Services, 2015). Para esto, se utiliza un administrador que considera las estadísticas en tiempo de ejecución como la cantidad de CPU utilizada, latencia o memoria disponible. Estas

estadísticas permiten ver en que rango de carga se encuentra la máquina según los umbrales establecidos, para posteriormente agregar o eliminar máquinas.

3.1.2 Recursos lógicos

A diferencia de la perspectiva basada en recursos físicos, en esta perspectiva 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 bajo esta mirada, analizan componentes como el flujo de datos o el tamaño de la cola de un operador, tomándolos como parámetros y definiendo umbrales en los algoritmos implementados para determinar cuando realizar las mejoras al sistema.

Existen dos tipos de enfoques: estático y dinámico (Gupta & Bepari, 1999). El primer enfoque está centrado en un modelo predefinido y que se mantiene fijo luego de la inicialización del sistema, sin considerar el estado del mismo. Éste se caracteriza por una implementación de bajo costo computacional, comparado con el enfoque dinámico, el cual es utilizado en general por los SPS. Por otra parte, el segundo enfoque está basado en un modelo el cual analiza el sistema según su estado en tiempo de ejecución, donde se utilizan políticas para optimizar el rendimiento de éste.

3.1.3 Enfoque estático

Este enfoque se ha implementando en distintos sistemas de procesamiento de *stream* y consiste en definir previo a la ejecución, la cantidad de operadores para la aplicación. De esta manera, no existe una interrupción en la ejecución o un cambio luego de la ejecución. (Casavant & Kuhl, 1988).

Storm utiliza distintas técnicas de distribución de las tuplas en los operadores según la política que se desee, todas tomando un 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 tuplas que requieren un mayor tiempo de procesamiento.

Fields grouping determina ciertas llaves a un operador determinado. Por ejemplo, se contiene un flujo de datos que se identifican por el nombre 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 puede repetir con mayor frecuencia que otras, lo cual es demostrado en la ley de potencia (Rushton, 2010).

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 crea un nuevo operador de manera automática. Debido a la infinidad de combinaciones de llaves que pueden generarse, S4 recomienda aplicar una función *consistent hashing* para evitar posibles colisiones (Wang & Loguinov, 2007). 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 puede poseer 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 palabras utilizadas en el diccionario (Rushton, 2010).

Una ventaja del enfoque estático es el bajo costo de la implementación de los métodos, comparado con modelos matemáticos o modelos planteados por el enfoque dinámico, lo cual es beneficioso para sistemas con escasos recursos y con restricciones de tiempo de respuesta. Por otra parte, una desventaja considerable es la existencia de puntos críticos en la topología, es decir, que un operador recibe más carga que sus pares, por lo que no se 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 en ejecución, siendo esto el parámetro base para optimizar su rendimiento (Casavant & Kuhl, 1988). Esto significa que si el sistema posee un operador sobrecargado, se realiza 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 el análisis de carga en a través de un monitor (Gulisano et al., 2012), el cual recibe periódicamente la información de carga de cada uno de los operadores, y en caso que sobrepase un umbral, se aplica una técnica para balancear la carga y con ello aumentar el rendimiento. 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, en el trabajo de Schneider (Schneider et al., 2009) se considera el rendimiento de cada operador. En caso de existir congestión en el operador, se procede a replicarlo de tal manera que exista un operador adicional que puede recibir un flujo de datos y realizar en paralelo la misma operación que el operador sobrecargado.

Si bien estas soluciones en su mayoría son eficientes, al otorgar un mejor rendimiento del sistema, 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 que puede surgir, corresponde a los falsos positivos, dado que puede existir un *peak* en el tráfico en una ventana de tiempo pequeña, pero eso no significa que ese comportamiento determine el flujo entrante.

Predictivo: este modelo está basado en modelos matemáticos que calculan o estiman el comportamiento a futuro del sistema, utilizando información histórica relativa a flujo entrante, CPU u otra variable. Si bien no existen modelos predictivos para SPS, si los existen en otras áreas, como se presentó en la Sección 2.6.2 con cadenas de Markov y análisis de series temporales en *Cloud Computing*.

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 enfocadas 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 más relevantes y que han sido trabajadas en la literatura relacionada al dominio del 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 cómo debe actuar el sistema.

En el área de *Stream processing* se ha realizado diferentes análisis de la estimación de frecuencia de *data stream* en el sistema. Para esto, se ha 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 llega al operador en la ejecución del sistema (Ganguly, 2009). También existen algoritmos que determinan la frecuencia del sistema dado el flujo de datos que se puede recibir a futuro (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 esta manera, se determina según la intensidad de la frecuencia, localización o clima, a donde debe enviar la señal para que se recolecte la información deseada (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 que puede analizarse respecto al promedio, sin embargo pueden surgir *peaks* o sucesos inesperados del tráfico en el transcurso de la ejecución que pueden generar una sobrecarga en el sistema. Por lo tanto, la estimación al realizarse *a priori*, sólo considera los datos del inicio del sistema o los de ventanas de tiempo anteriores, por lo que puede existir un porcentaje de error considerable. Por otra parte, se considera que esta técnica posee mejor rendimiento si es que la frecuencia o función analizada es estacionaria o comportamientos repetitivos, pero no sucesos inesperados, como es el caso de las redes sociales, debido que suceden eventos externos que influyen en el tráfico analizado (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 operadores sobrecargados, ya sea en términos del tamaño de la cola, tasa de rendimiento u otro factor. En la Figura 3.1 se observa que existe un operador A, el cual recibe datos en un período de tiempo, pero debido a la cola existente en el

sistema, se considera utilizar un operador denominado *Shedding*, que en caso de existir una cola mayor al umbral propuesto, éste descarta los eventos candidatos según la métrica establecida. 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 y existir cola, son descartados por éste. De esta manera, al existir menor cantidad de ruido, existe un mejor procesamiento del video, teniendo una mejor calidad en la visualización de los mismos (Sheu & Chi, 2009).

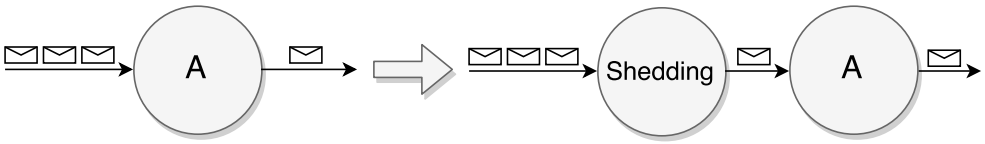

Figura 3.1: Load shedding en un SPS.

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 son 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 posee un bajo cómputo computacional, siendo pensada para la disminución rápida de carga, se introduce al sistema una baja en la precisión y fiabilidad en los resultados. 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 da una menor precisión de la información obtenida por el sistema, dado que puede perderse información que esté relacionada con los comportamientos 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 del un operador al nodo 2, ya que éste último se encuentra con menor carga. De esta manera, se reparten homogéneamente la carga.

Si bien no existe una 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 que puede existir una falla en la comunicación del operador, donde no existe un mecanismo para evitar la pérdida de los datos. Debido a lo anterior, es que se propone el uso de *buffer*, el cual respalde la información, aumentando los costos del sistema (Pittau et al., 2007).

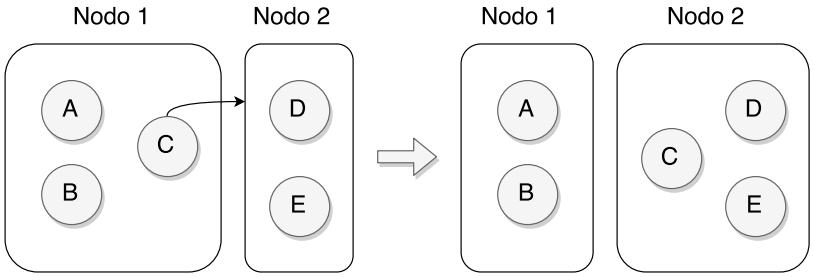

Figura 3.2: Técnica de migración en un SPS.

3.2.4 Fisión

Otra técnica utilizada en el balance de carga es la fisión, o también llamada replicación, que permite paralelizar el procesamiento, la cual consiste en crear una réplica paralela del operador, sin perder el funcionamiento y estado. En Figura 3.3 se presenta un operador A, el cual en primera instancia recibe un flujo de entrada q_1 y genera un flujo de salida q_2. Sin embargo, dado que este operador se sobrecarga, se procede a replicar, por lo que se hace necesario dos operadores auxiliares, en caso de ser un operador con estado (Wu et al., 2012), los cuales son el operador *Split* y *Merge*. Estas fases son necesarias para distribuir y juntar la información respectivamente, y en general son de bajo costo. En caso de ser necesario replicar el operador *Merge*, debe existir un *merge* de las distintas réplicas de éste, lo cual hace más complejo el diseño del sistema, por lo que se omite su replicación.

En algunos SPS 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.

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.

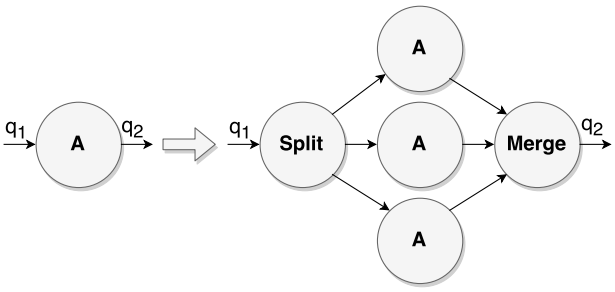

Figura 3.3: Técnica de fisión en un SPS.

Otro sistema que utiliza esta técnica, bajo un enfoque dinámico, es *StreamCloud* (Gulisano et al., 2012). Según la cantidad de consultas realizadas al sistema se aumenta o disminuye la cantidad de operadores que cumplen las tareas que se solicitan. Para esto, es necesario un operador que distribuya los datos, denominado *split*, y uno que junte la información entregadas por las réplicas del operador, denominado *merge*. Por lo que este sistema sólo soporta ciertas operaciones, de tal manera que se creen de forma automática los operadores 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 que aplica 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 éste propone utilizar *Cloud* para el uso del SPS. De esta manera, el enfoque se basa en la paralelización de las tareas entre las distintas máquinas.

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 cada cierto período de tiempo el estado de cada uno de los operadores. De esta manera, en caso que un operador sobrepase el umbral de carga establecido, el cual está determinado por la utilización de CPU se replique el operador. 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 que *o* se sobrecarga (cuello de botella), éste se replica tantas veces como sea necesario hasta reducir la carga hasta el umbral deseado.

Dentro de las desventajas existentes por parte de estos trabajos realizados, es que no utilizan la historia del operador para analizar el comportamiento a futuro de la carga. El uso de sistemas de predicción puede estabilizar el sistema cuando existen *peaks* en el tráfico, debido que estos son detectados en el pasado, y se verifica si pueden ocurrir en el futuro. Otra desventaja, es que en algunos trabajos es necesario detener la ejecución de la aplicación, cambiar el número

de réplicas y volver a iniciar, lo cual genera una pérdida de eventos.

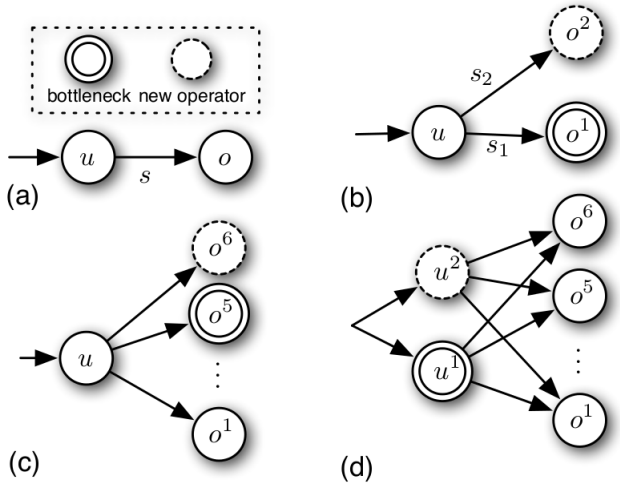

Figura 3.4: Ejemplo de replicación de los operadores (Fernandez et al., 2013).

CAPÍTULO 4. DISEÑO DEL MODELO ELÁSTICO

En este capítulo se aborda el análisis del modelo elástico de replicación de operadores de un SPS, tomando en consideración una técnica utilizada para el balance de carga, como también las bases teóricas para el modelo diseñado. Además de lo anterior, se hace una descripción detallada del diseño de cada uno de los módulos del modelo.

4.1 ANÁLISIS DEL MÓDELO ELÁSTICO

Dentro del análisis realizado en la arquitectura del sistema implementado, se utiliza una perspectiva basada en los recursos lógicos del sistema según el enfoque dinámico, la cual es explicada en la Sección 3.1.2, para el balance de carga del SPS. El presente trabajo no se enfoca en el análisis del comportamiento de cada uno de los nodos físicos del sistema, sino más bien en el estado de 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, es necesario utilizar una que minimice la pérdida de datos, que sea capaz de adaptarse a las condiciones dinámicas del tráfico, y que sea de bajo *overhead* para el sistema, de tal manera que sea escalable. Bajo estas restricciones, se considera que la mejor opción es utilizar la técnica de fisión, utilizando el modelo de replicación de (Fernandez et al., 2013), donde basándose en el nivel de carga del operador, se evalúa la generación de una réplica para éste. Dentro de los supuestos realizados, se plantea que el costo de un operador es menor a la formación de las colas de datos en el sistema, lo cual puede variar según la arquitectura del SPS implementando.

En la Figura 4.1 se muestra un ejemplo de la replicación propuesta. La Figura 4.1 (a) se presentan tres operadores, donde en el operador B existe una sobrecarga representada por una doble circunferencia, por lo que es necesario replicar el operador. En la Figura 4.1 (b) se presenta el mismo 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 Figura 4.1 (c).

Para la detección del nivel de carga de un operador es necesario contar con un modelo basado en umbrales que permita determinar cuando está o no sobrecargado un operador. Para modelar esta situación se utilizan conceptos de teoría de colas (Bose, 2013). Dado que los SPS utilizan un paradigma orientado a grafos, se puede obtener tanto la tasa de llegada (λ) como la tasa de procesamiento (μ) de cada uno de los operadores que lo componen, como

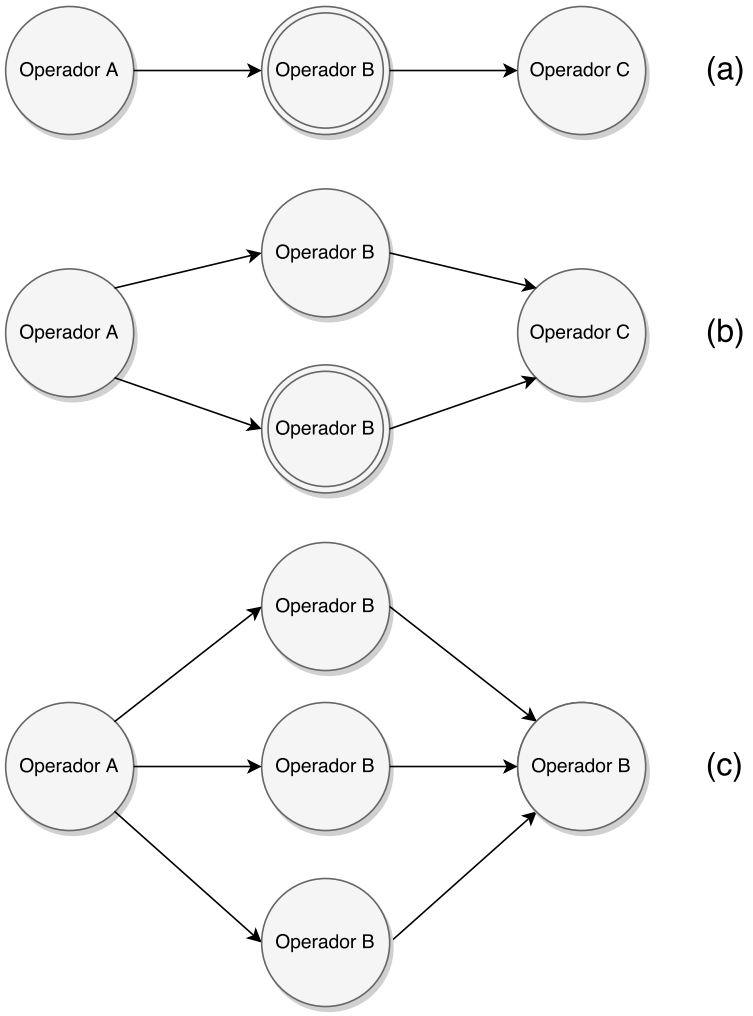

Figura 4.1: Ejemplo de replicación del modelo propuesto.

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 calcula la tasa de rendimiento (ρ), la cual está definida por la tasa de llegada, de procesamiento y la cantidad de réplicas del operador, como se muestra en la Ecuación 4.1, cuyo valor representa el factor de utilización del sistema, donde se define un sistema estable si y sólo si $\rho < 1$.

$$\rho = \frac{\lambda}{s\mu} \tag{4.1}$$

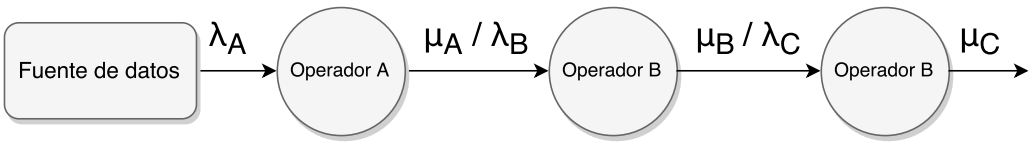

Figura 4.2: Enfoque de un SPS con conceptos de teoría de colas.

Tomando en consideración el enfoque dinámico en el algoritmo de balance de carga y la elasticidad que se pretende por parte del modelo, es que se han tratado tres posibles estados en el sistema: ocioso, estable e inestable. El primer estado corresponde a un exceso en la cantidad de recursos asignados. El segundo está definido por el rendimiento óptimo del sistema. Y por último, el tercero hace referencia a un sistema sobrecargado, donde es necesario una mayor cantidad de recursos por parte de éste. Los posibles estados de cada operador son la base para el análisis y predicción de la carga del modelo propuesto.

Para el modelo se consideraron dos tipos de algoritmos: predictivo, enfocado en el futuro basado en la historia del operador, y reactivo, analizando el comportamiento del momento. Este diseño tiene la finalidad de analizar dos factores: los *peaks* existentes en la historia del operador, dado el algoritmo predictivo, y analizar el comportamiento en el momento, haciendo uso del algoritmo reactivo, de tal manera de solucionar los comportamientos que no son detectados con la predicción.

Otra de las consideraciones realizadas es que cada uno de los algoritmos va a resolver un problema en específico según su política de resolución. Por ejemplo, si se desea detectar un comportamiento según el historial, el más indicado es el algoritmo predictivo. Y en caso de analizar *peaks* según el presente, el algoritmo reactivo puede dar soluciones al respecto.

Además de lo anterior, se considera que es necesario trabajar con los dos algoritmos en temporalidades distintas, es decir, en cierto período de tiempo se utiliza el reactivo y en otro el predictivo. Esto quiere decir que mientras se obtienen las n muestras para realizar el cálculo de predictivo, el algoritmo reactivo está realizando un análisis de los distintos operadores.

Como se diseñaron dos tipos de algoritmos que se complementan, es necesario considerar un mecanismo 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 ha diseñado un modelo elástico basado en el modelo MAPE(Jacob et al., 2004), el cual posea cuatro componentes: monitor de carga, analizador de carga, predictor de carga y administrador de réplicas, que se presenta en la Figura 4.3.

A continuación se describe de manera resumida cada uno de los siguientes módulos, los cuales posteriormente son explicados con mayor profundidad:

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 indica el estado del operador, el cual

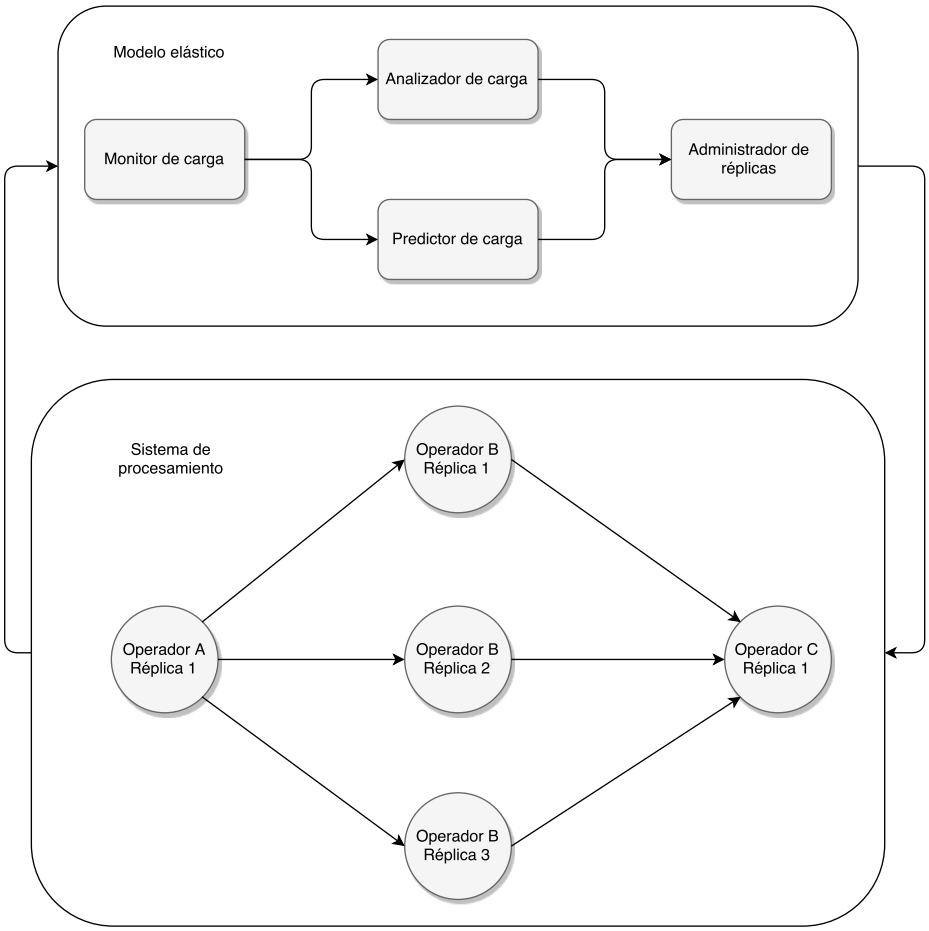

Figura 4.3: Estructura del modelo elástico.

puede ser ocioso, 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 las variaciones del sistema. Posteriormente, para la predicción de la carga del operador, se calcula la distribución estacionaria (Papoulis & Pillai, 1984), el cual entrega la probabilidad que el operador se encuentre en cada uno de los posibles estados.

Administrador de réplicas: se encarga de determinar cuál 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 modelo elástico, tanto el historial para el algoritmo predictivo, como la tasa de rendimiento para el algoritmo reactivo. La recolección de muestras se realiza en una ventana de tiempo T_m.

Cada muestra está compuesta por la tasa de llegada y la tasa de procesamiento del operador, la cual es utilizada tanto por el algoritmo reactivo como predictivo. Por parte del algoritmo reactivo, se considera el promedio de las tasas de llegada y servicio en la última ventana de tiempo T_r. En cambio, para el algoritmo predictivo, se considera las muestras obtenidas en la última ventana de tiempo T_p.

Dentro de las consideraciones realizadas para la recolección de datos, dentro de las consideraciones está que la tasa de procesamiento del operador es homogénea. Por lo tanto, se realiza un *benchmark* de la tasa de procesamiento del operador al inicio de la ejecución del sistema con las primeras muestras obtenidas, de tal manera de obtener el valor que se utiliza para el cálculo de la tasa de rendimiento.

4.3 ALGORITMO REACTIVO

Este análisis emplea las muestras obtenidas en la última ventana de tiempo T_r, las cuales permiten describir el estado del operador (ocioso, estable o inestable) en dicha ventana de tiempo.

En el Algoritmo 4.1 se presenta el mecanismo de análisis de estado para un operador dado. Dicho estado está determinado por su tasa de rendimiento, la cual en caso de ser mayor a 1, su estado es inestable, si es menor a 0.5, su estado es ocioso, y en otro caso, significa que está estable. Estos datos posteriormente son considerados por el administrador de réplicas, para una posible modificación de la cantidad de réplicas según el comportamiento del operador.

En la Figura 4.4 se observa el estado de un operador según la tasa de procesamiento. En los primeros 90 segundos la tasa del operador es mayor al límite superior, lo cual indica que el sistema es inestable, es decir, el operador posee sobrecarga. Posteriormente, en el segundo 50, 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 que se encuentra entre el límite inferior y superior, cuyo rango define al operador como un sistema estable hasta el segundo 170, donde se encuentra

Algoritmo 4.1: Algoritmo reactivo del modelo elástico.

Entrada: Tasa de rendimiento ρ del operador i.
Salida: Actual estado δ del operador i.

```
1: if ρ_i > 1 then
2:     return φ_i: "inestable"
3: else if ρ_i < 0.5 then
4:     return δ_i: "ocioso"
5: else
6:     return δ_i: "estable"
7: end if
```

ocioso, hasta llegar al segundo 230.

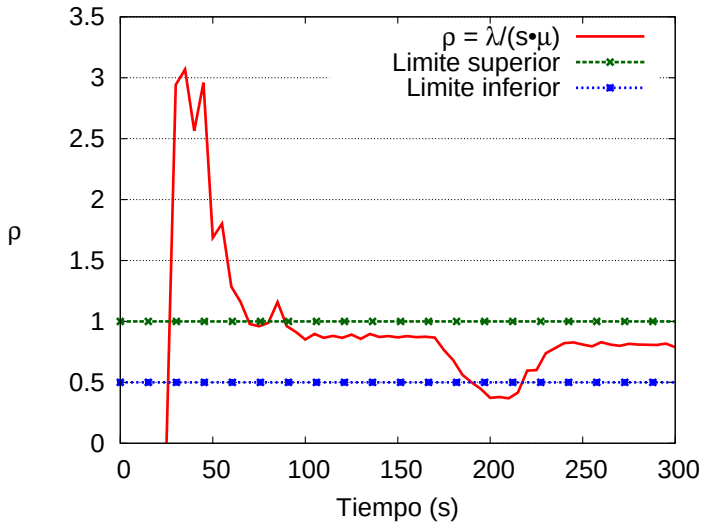

Figura 4.4: Comportamiento de la tasa de procesamiento de un operador.

4.4 ALGORITMO PREDICTIVO

Para la confección del algoritmo predictivo se ha realizado 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 cambian con el tiempo según un proceso estocástico. Las muestras se definieron como la tasa de procesamiento (ρ) del operador, la cual dependiendo de su valor, otorgan un estado al operador.
- Determinar los estados finitos que se utilizan para la conformación de la cadena, que son los estados que se puede encontrar el operador: ocioso, estable o inestable.

- Obtener una cantidad representativa de muestras para la construcción de la cadena de Markov en el período analizado. Estas muestras son independientes entre un período y otro, por lo que los valores de la cadena de Markov cambian en cada período de tiempo.

Tomando las bases anteriores, se ha diseñado una cadena de Markov en base a tres posibles estados: ocioso, estable e inestable, como se muestra 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.

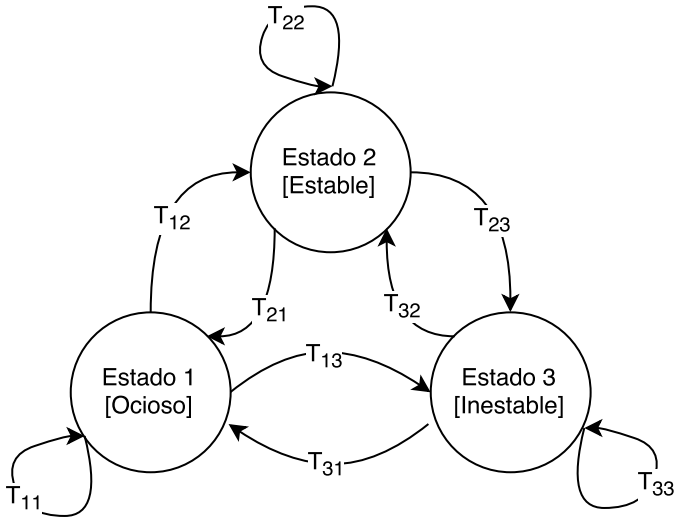

Figura 4.5: Cadena de Markov dado el modelo propuesto del sistema.

Por lo tanto, para cada operador se construye una cadena de Markov según el historial obtenido en la ventana de tiempo T_p. Para la conformación de la cadena de Markov se ha considerado las muestras de la historia de los operadores, por lo que el comportamiento del estado entre una muestra a otra representa una transición, las cuales dan origen a la matriz de transición. En el Anexo A se presenta el algoritmo que se ha empleado para construir esta matriz. En la ecuación 4.2 se muestra la matriz de transición que se obtiene de la cadena de Markov de la Figura 4.5.

$$P = \begin{bmatrix} T_{1,1} & T_{1,2} & T_{1,3} \\ T_{2,1} & T_{2,2} & T_{2,3} \\ T_{3,1} & T_{3,2} & T_{3,3} \end{bmatrix} \tag{4.2}$$

Obtenida la matriz de transición se puede calcular la distribución estacionaria de la cadena de Markov, la cual indica las probabilidades de que en el futuro el operador esté en cada uno de los posibles estados, ya sea ocioso, estable o inestable. Para este cálculo se utiliza la ecuación de Chapman-Kolmogórov (Papoulis & Pillai, 1984) descrita en la Sección 2.6.1.

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

Entrada: P Matriz de transición del operador i y υ cantidad de iteraciones deseadas.
Salida: Δ Distribución estacionaria de la cadena de Markov del operador ϕ.

```
1: i ← 1
2: for k = 0 a υ do
3:    u = randomUniform(0, 1)
4:    σ = 0
5:    for j = 1 a 3 do
6:       σ = σ + P_{i,j}
7:       if u ⩽ σ then
8:          τ_j + +
9:          i ← j
10:         break
11:      end if
12:   end for
13: end for
14: for k = 1 a 3 do
15:   Δ_k ← τ_k/υ
16: end for
17: return Δ
```

El Algoritmo 4.2 describe el cálculo de la distribución estacionaria, cuya entrada es la matriz de transición de un operador del SPS y $upsilon$ que corresponde a la cantidad de transiciones realizadas para obtener la distribución estacionaria.

Antes de realizar el cálculo de la distribución estacionaria, es importante determinar si la cadena es irreductible y sus estados son recurrentes positivos aperiódicos. En caso de ser reductible, se debe determinar la cadena irreductible, y utilizar ésta para el cálculo de la distribución estacionaria.

La cantidad de iteraciones υ que debe realizarse para el cálculo correspondiente, es un parámetro entrada del algoritmo. Es importante destacar que entre mayor es la cantidad de iteraciones, mayor es la precisión del valor predicho. Esto implica a su vez un mayor tiempo de cómputo, por lo que en este trabajo se ha considerado un valor medio determinado por pruebas que permitieron medir el costo en tiempo de cómputo versus la calidad de los resultados obtenidos.

Obtenida la distribución estacionaria, se procede a analizar las probabilidades obtenidas y como es el comportamiento del operador. Para esto, se ha considerado que las variables aleatorias obtenidas tengan una desviación estándar superior a α. El anterior parámetro tiene por objetivo reducir la incertidumbre en el momento de seleccionar un estado como posible comportamiento futuro. Por lo tanto, un valor de α adecuado permite establecer una diferencia significativa entre las variables aleatorias al considerar uno de estos estados, disminuyendo el error de la predicción realizada. En el caso que no supere la desviación estándar, puede

Algoritmo 4.3: Algoritmo predictivo del modelo elástico.

Entrada: Δ Distribución estacionaria de la cadena de Markov del operador ϕ.
Salida: Futuro estado δ^+ de la predicción del operador i

```
if σ(Δ_1, ..., Δ_m) > α; mϵ[1, 3] then
    return δ+: max(Δ_1, ..., Δ_m)
else
    return δ+: "estable"
end if
```

ser que dos probabilidades sean muy parecidas y la probabilidad no sea un comportamiento determinante (Soong, 2004). El Algoritmo 4.3 detalla el análisis que se realiza a la distribución estacionaria, siendo en primer lugar el análisis estadístico de las probabilidades, y en segundo lugar la obtención del estado con mayor valor de las probabilidades, retornando finalmente el estado del operador.

4.5 ADMINISTRACIÓN DEL SISTEMA

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

Para esto, se ha diseñado un administrador que ejecuta el algoritmo reactivo en un período de T_r y el algoritmo predictivo en un período de T_p. Cabe destacar, que la ventana de tiempo utilizada para el algoritmo reactivo es menor que la del predictivo, esto debido que uno analiza el comportamiento en el momento y otro a futuro según la historia.

En el Algoritmo 4.4 está el procedimiento de administración, donde primero se analiza qué tipo de algoritmo se debe ejecutar según el período del ciclo. En caso de ejecutarse el módulo predictivo, se analiza el resultado de la predicción, por lo que si el estado es ocioso el sistema disminuye la cantidad de réplicas y si es inestable las aumenta. Como el proceso de predicción se realiza con menor frecuencia y analiza una mayor ventana de tiempo, se ha considerado que se debe modificar una mayor cantidad de réplicas que en el módulo reactivo. Por otra parte, de realizarse el algoritmo reactivo se analiza si existen β 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 de cambiar el estado actual del operador a estable, de tal manera de no considerar ese período para el próximo análisis reactivo.

Una de las consideraciones que se han tenido para el diseño del administrador, es limitar la cantidad máxima de réplicas que pueden realizarse. Esto se debe a que una de las limitantes de este trabajo es que se utiliza solo una máquina, y por ende la cantidad de

recursos son limitados, por lo que al aumentar la cantidad de réplicas indefinidamente genera una sobrecarga en los recursos disponibles por parte de la máquina.

Algoritmo 4.4: Administración de réplicas de un operador i dado su comportamiento en el modelo elástico.

Entrada: Operador i a analizar y ι la ventana de tiempo del modelo elástico.
Salida: Cantidad de réplicas a modificar del operador.

1: **if** ι es T_p **then**
2: $\delta_\iota \leftarrow AlgoritmoPredictivo(\phi)$
3: **if** δ_ι: "inestable" **then**
4: **if** No excede la cantidad máxima de réplicas en el sistema **then**
5: **return** Crear θ réplicas del operador i
6: **end if**
7: **else if** δ_ι: "ocioso" **then**
8: **return** Remover θ réplicas del operador i
9: **end if**
10: **else** //ι es T_r
11: $\delta_\iota \leftarrow AlgoritmoReactivo(\phi)$
12: **if** $\delta_\iota, \ldots, \delta_{\iota-(\beta-1)}$: "inestable" **then**
13: **if** No excede la cantidad máxima de réplicas en el sistema **then**
14: $\delta_\iota \leftarrow$ estado estable
15: **return** Crear ω réplica del operador i
16: **end if**
17: **else if** $\delta_\iota, \ldots, \delta_{\iota-(\beta-1)}$: "ocioso" **then**
18: $\delta_\iota \leftarrow$ estado estable
19: **return** Remover ω réplica del operador i
20: **end if**
21: **end if**

CAPÍTULO 5. EXPERIMENTOS Y EVALUACIÓN

5.1 IMPLEMENTACIÓN DEL SISTEMA

Para la implementación del sistema propuesto, se ha utilizado como base el sistema de procesamiento de *stream* S4 (S4, 2014), cuyo modelo fue explicado en la Sección 2.3. El desarrollo del modelo elástico de replicación de operadores de un SPS se ha implementando en Java, modificándose el código fuente de S4 para incluir las funcionalidades de la solución propuesta, de tal manera que sea automático y transparente para el usuario del SPS.

El sistema diseñado se ha añadido al proyecto S4, generando un paquete denominado *modelo* que contiene las clases: *S4Monitor*, *MarkovChain*, *modeloMetrics*, *StatusPE* y *TopologyApp*, para mayor información véase el Anexo B.

Para la ejecución del SPS en conjunto con el modelo, se debe en primer lugar registrar los PEs. Además de esto, se ha añadido a cada PE como atributo la cantidad de eventos y salientes, y también un boolean en caso que se desee sólo una réplica del operador.

Para el funcionamiento del modelo se ejecutan dos tareas en la aplicación de S4: una tarea que recolecta los datos y otra que recupera los resultados obtenidos por el modelo elástico. En caso que el administrador de réplicas indique se requiere modificar la cantidad de réplica, la segunda tarea realiza los cambios correspondientes al PE expresado. Los códigos asociados a los de creación o eliminación de réplicas se presentan en el Anexo C.

Dado que en el diseño se propuso un mecanismo de balance de carga basado en la técnica de replicación, se hace necesario modificar S4, de manera de poder manejar las réplicas de los operadores. Para la distribución de eventos cuando existe más de una réplica, se aplica una política basada en el largo de la cola de los operadores, donde aquel que posea menor largo es el candidato para recibir el evento entrante. El componente que escoge esto, es el *pipeline* que existe entre los dos operadores, el cual analiza a qué operador debe enviar el dato, según la política explicada anteriormente. Este pipeline es una clase llamada *Stream*, el cual va a actuar como intermediario entre el operador emisor y receptor, de tal manera que en caso que se envíe el evento, va a pasar por esta clase y encola el evento para ser procesado posteriormente en alguna de réplicas disponibles.

En la Figura 5.1 se explica gráficamente la distribución de la carga. En la Figura 5.1 (a) el operador A envía un evento al operador B, el cual posee tres réplicas, siendo las colas de menor tamaño las que están en la réplica 2 y 3. Dado que el tamaño de las colas son iguales, se escoge la primera réplica disponible, es decir, la réplica 2. Posteriormente, como la réplica 2 aumenta su cola, la réplica 3 es la que posee menor cantidad de eventos en cola, como se

demuestra en la Figura 5.1 (b), por lo tanto, ésta es la réplica candidata a recibir el evento enviado. Finalmente, en la Figura 5.1 (c) todas las réplicas poseen el mismo tamaño de la cola, por lo tanto, se procede a enviar a la primera réplica.

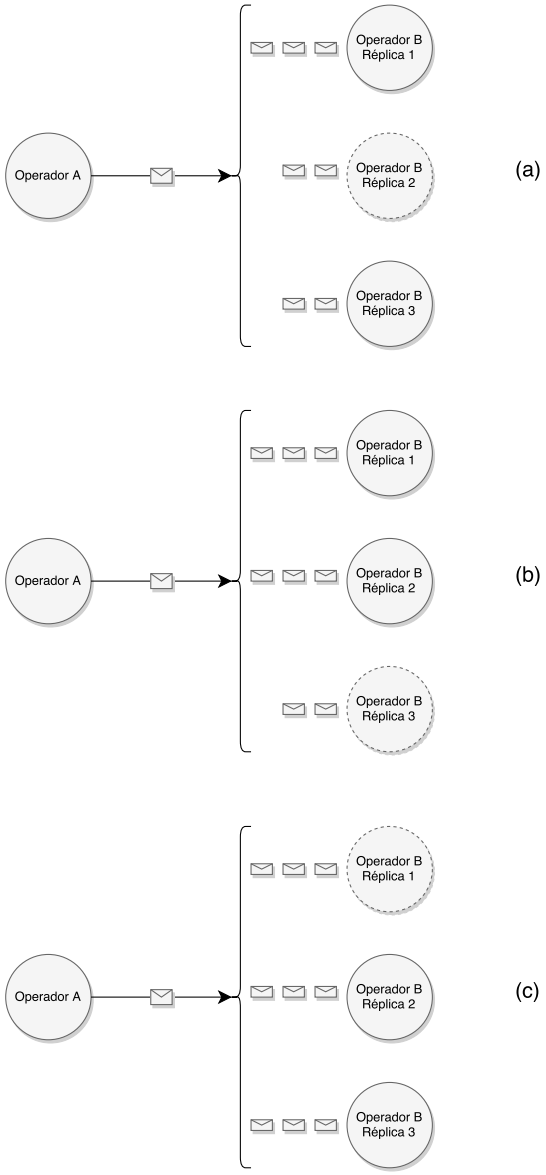

Figura 5.1: Distribución de la carga entre las réplicas.

5.2 DISEÑO DE LOS EXPERIMENTOS

Para los experimentos se han diseñado tres aplicaciones: una que realiza operaciones con estado, otra que realiza operaciones sin estado, y otra aplicación sintética. En el

caso que la aplicación posea estado, significa que el operador guarda variables al procesar con el transcurso del tiempo, las cuales son entregadas cada cierto tiempo o al finalizar la ejecución del sistema. Este tipo de aplicaciones son las más utilizadas, dado que utilizan contadores o variables globales, las cuales son necesarias para el análisis de los datos. Un ejemplo de esto, es un sistema que cuenta las palabras de un texto y que envía la cantidad de palabras contadas cada cierto tiempo. Si bien, las aplicaciones sin estado no son tan utilizadas, se consideraron importantes para la validación del modelo, dado que son las aplicaciones más simples y básicas de diseñar y ejecutar. Por otra parte, se ha generado una aplicación sintética, la cual simula el procesamiento de datos por medio de la introducción de retrasos de cada uno de los operadores. Cada operador es una hebra de procesamiento, para reflejar el tiempo de ejecución, por lo que se procede a dormir la hebra por un tiempo predefinido, el cual está determinado por la complejidad del operador. Los operadores más complejos son dormidos por una mayor cantidad de tiempo.

Para la generación del *stream* de la fuente de datos para la primera y segunda aplicación, se ha utilizado un conjunto de 4.5 millones de *tweets*[1], los cuales fueron recolectados entre los días 27-28 de Febrero y 1-2 de Marzo de 2010, tanto en inglés, portugués y español. Estos contienen información correspondiente a la interacción entre los usuarios durante el terremoto ocurrido el 27 de Febrero en Chile.

Por otra parte, el envío de los datos se ha realizado de dos maneras: constante y variable. El envío de datos constante consiste en enviar 100 eventos por segundo durante todo el experimento. El envío de datos variable consiste en enviar 50 eventos por segundos el primer tercio del experimento, para luego aumentar a 100 eventos por segundo en el segundo tercio, y finalmente, disminuir a 50 eventos por segundos el último tercio de la ejecución.

5.2.1 Aplicación 1: Análisis de *tweets* en escenarios de desastres naturales

La primera aplicación es orientada a un escenario de desastres naturales, donde se genera un grafo que realice un filtrado de palabras, identificación del idioma y conteo de palabras. Ninguno de los operadores posee estado, por lo tanto son independientes. Para la duración de esta prueba se ha considerado un tiempo de ejecución de 70 minutos, cuya duración está basada en el *benchmark* confeccionado por (Arasu et al., 2004). El objetivo de esta aplicación, es comprobar que el sistema diseñado puede funcionar con aplicaciones sin estado, las cuales son las más básicas y sencillas de diseñar en SPS, y que tienen utilidad cuando se hacen análisis directo del flujo de datos.

[1] Un *tweet* es una publicación o actualización de estado realizada en la red social *Twitter*. Como tal, un *tweet* es un mensaje cuyo límite de extensión son 140 caracteres. Puede contener letras, números, signos y enlaces.

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

Stopword: es el operador encargado de leer el *tweet* y remover las palabras que no son relevantes para el análisis de éste, usando una bolsa de palabras (*stopwords*). De esta manera, se puede analizar el texto usando las palabras más representativas de éste y así entregar información más precisa.

Language: es el operador encargado de identificar el lenguaje existente en el *tweet*, para esto se utilizado 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: es el operador encargado de contar la cantidad de palabras que existen en el *tweet* según una bolsa de palabras proporcionada por el programador. Para esta prueba se ha utilizado una bolsa de 26.000 palabras en español. Con esto se pueden detectar los *tweets* que poseen mayor cantidad de palabras claves asociados a una temática o evento en particular.

MongoDB: es el operador encargado de guardar en la base de datos el evento según los atributos que posee, ya sea por el *tweet* original, sin *stopword*, idioma y cantidad de palabras claves existentes en él. Para esto, se ha utilizado 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.

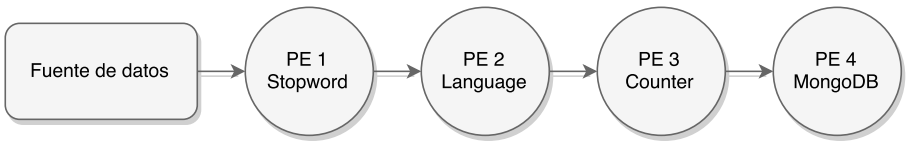

Figura 5.2: Aplicación 1: Análisis de *tweets* en escenarios de desastres naturales.

5.2.2 Aplicación 2: Contador de palabras en muestras de textos

La segunda aplicación consiste en un contador de palabras, la 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 es necesario un contador en el operador, de tal manera de contar la cantidad de veces que se repite cierta palabra según una bolsa de palabras definida. 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 ha considerado un tiempo de 70 minutos, al igual que en la anterior aplicación. El objetivo de esta aplicación es validar el modelo diseñado con aplicaciones con estado, las cuales son las más aplicadas, debido que se realizan análisis generales de los datos, como el caso de los *trending topics* o frecuencia de palabras.

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

Split: es el operador encargado de dividir el *tweet*, y enviar un arreglo con las palabras que posee al operador *Counter*.

Counter: es el operador encargado de llevar las estadísticas de los contadores de cada palabra. Cuando recibe un evento, el operador aumenta el contador de las palabras que corresponde al arreglo de palabras enviado. Las estadísticas son enviadas cada 10 segundos al operador Merge, de tal manera de no enviar flujo constante al siguiente operador y sobrecargar al operador.

Merge: es el operador encargado de unir las distintas estadísticas enviadas por las distintas réplicas del operador Counter, sumando los valores que corresponde a las palabras enviadas.

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 Sección 3.2.4.

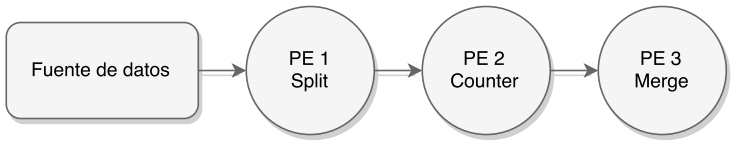

Figura 5.3: Aplicación 2: Contador de palabras en muestras de textos.

5.2.3 Aplicación 3: Aplicación sintética

La tercera aplicación conste de tres operadores sintéticos, los cuales son hebras de procesamiento que llevan a cabo una tarea específica, dependiendo del tipo de operador que representa. Para simular el procesamiento de eventos, se procede a dormir la hebra en el momento que recibe el evento por un tiempo determinado experimental de acuerdo a la complejidad del operador. Esto quiere decir, los operadores más complejos toman mayor tiempo en procesar cada evento, por ende la hebra debe ser dormida un mayor tiempo para simular dicho comportamiento. El objetivo de este experimento es evaluar los costos en términos de memoria y uso de la CPU que posee el modelo elástico propuesto.

Para definir los tiempos de procesamientos de las hebras, se crearon operadores reales con complejidades diferentes y midiendo el tiempo que tarda en procesar un evento. Con esto tiempos promedios obtenidos, se ha definido el período de tiempo que duerme la hebra en cada uno de los PEs, los cuales se presentan en la Tabla 5.1.

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

PE	Tiempo (ms)
1	20
2	30
3	15

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.

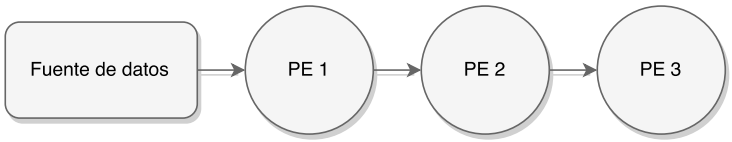

Figura 5.4: Aplicación 3: Aplicación sintética.

5.3 EVALUACIÓN

Para la ejecución de los experimentos, se ha utilizado un servidor con sistema operativo Ubuntu 14.04.2 LTS, cuyo procesador es un Intel Xeon CPU E5-2650 v2 de 2.60 GHz y con 32 GB de RAM. Cabe recalcar, que el lenguaje de programación es Java, debido a la

integración del sistema propuesto en el SPS utilizado. La configuración de S4 es detallada en el Anexo D.

Para la ventana de tiempo del recolector de datos se utiliza un valor de $T_m = 1seg$. Para el administrador de replicas los parámetros del numero de replicas a crear por cada algoritmo se definieron con los valores $\omega = 1$ (reactivo), y $\theta = 5$ (predictivo). Las ventanas de tiempo entre las ejecuciones de los algoritmos son $T_r = 5seg$ y $T_p = 100seg$ respectivamente. Los parámetros del algoritmo predictivo para estos experimentos son una cantidad de muestras de $n = 100$, una cantidad de iteraciones para el cálculo de la distribución estacionaria de $\upsilon = 100000$ y una desviación estándar de $\alpha = 0.25$ entre las variables aleatorias obtenidas por la distribución estacionaria. La cantidad de alertas consecutivas para activar el análisis del algoritmo reactivo es de $\beta = 2$.

5.3.1 Aplicación 1: Análisis de *tweets* en escenarios de desastres naturales

Con la primera aplicación se efectuaron dos experimentos, donde el primero consta de un envío constante de eventos, y el segundo de un envío variable. En ambos experimentos, se han considerado dos pruebas, la primera consiste en la ejecución de la aplicación en un sistema SPS con el uso del modelo elástico, y la segunda sin el uso de éste.

Para el análisis de los experimentos, se ha considerado el rendimiento y la cantidad de réplicas del grafo y la cantidad total de eventos procesados. En el Anexo E se presentan las estadísticas de cada uno de los operadores.

Las Figuras 5.5 y 5.6 corresponden al primer experimento y muestra el rendimiento que posee el sistema, y como varía la cantidad de réplicas totales del grafo según el flujo constante de datos. En la Figura 5.5 se observa que en los primero segundos existe una sobrecarga en el grafo, específicamente en los PE Stopword y Counter, por lo que el modelo detecta esta inestabilidad en el sistema y aumenta la cantidad de réplicas de los operadores (véase Anexo E), logrando un procesamiento promedio de 98 eventos por segundo. En el caso de la Figura 5.6 no existe un aumento en la cantidad de réplicas, por lo tanto, no existe un aumento de la cantidad de datos procesados, alcanzando un promedio de 16 eventos procesados por segundo. Se puede observar que en esta figura en el segundo 2600 la tasa de entrada disminuye considerablemente, esto es resultado de la sobrecarga del sistema. Comparando ambos escenarios se puede observar que la utilización del modelo elástico para esta aplicación ha logrado mejorar el rendimiento del sistema, logrando incrementar 5 veces el numero total de eventos procesados.

En las Figuras 5.7 y 5.8 se presenta la cantidad total de eventos procesados

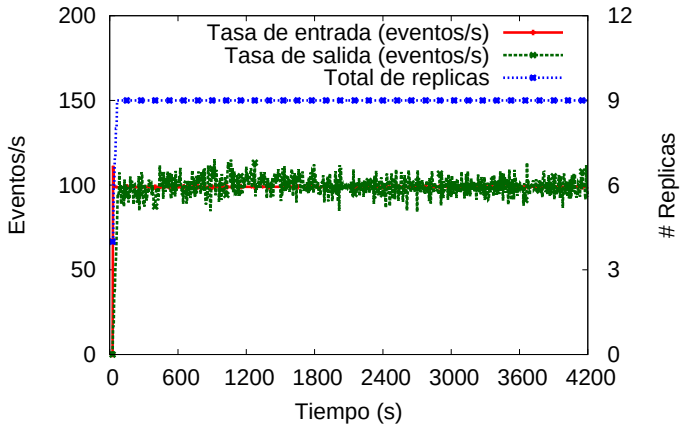

Figura 5.5: Rendimiento y cantidad de réplicas totales del grafo en la primera aplicación con envío constante de la fuente de datos con uso del modelo.

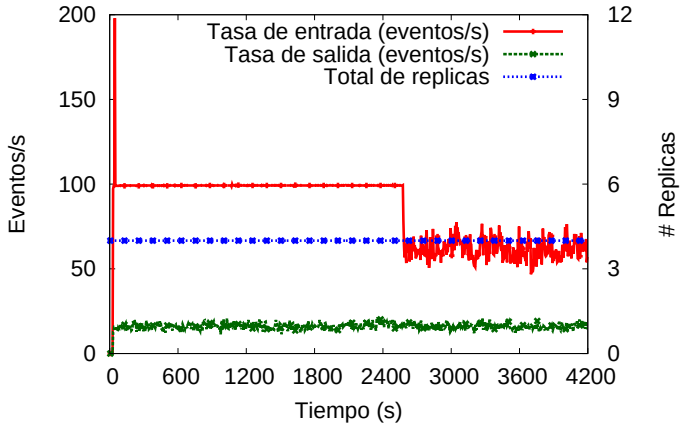

Figura 5.6: Rendimiento y cantidad de réplicas totales del grafo en la primera aplicación con envío constante de la fuente de datos sin uso del modelo.

en el transcurso de la ejecución en cada uno de los operadores, con y sin uso del modelo respectivamente. En el primer gráfico los cuatro operadores del SPS van aumentando la cantidad total de eventos procesados linealmente y con la misma pendiente, tan sólo existe una menor cantidad de eventos procesados en el tercer PE, lo cual se traslada al cuarto PE, debido a que al procesar menor cantidad de eventos en el tercer PE, llega una menor cantidad de eventos al cuarto PE. En este gráfico se alcanza un total de 401.618 eventos procesados. En cambio, en el segundo gráfico las curvas de cantidad de eventos procesados son muy distintas entre 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 que alcanza un total de 67.141 eventos. Por lo que con el uso del modelo elástico se ha procesado 6 veces más eventos que los procesados durante el mismo período de tiempo sin el uso de éste.

Las Figuras 5.9 y 5.10 corresponden al segundo experimento y muestra el

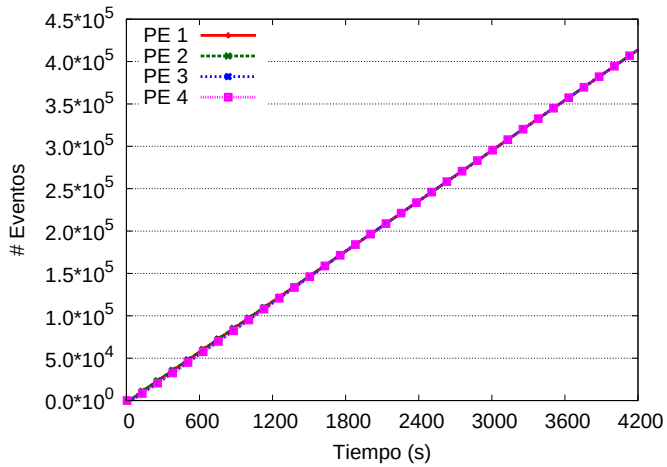

Figura 5.7: Cantidad total de eventos procesados en la primera aplicación con un envío constante de la fuente de datos con uso del modelo.

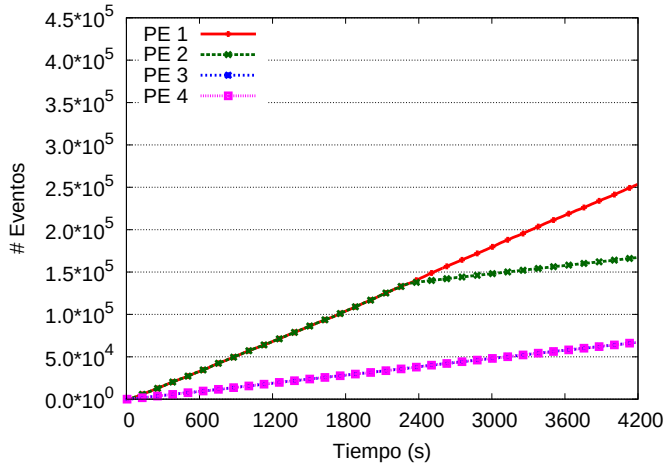

Figura 5.8: Cantidad total de eventos procesados en la primera aplicación con un envío constante de la fuente de datos sin uso del modelo.

rendimiento que posee el sistema, y como varía la cantidad de réplicas totales del grafo según el flujo de datos. En la Figura 5.9 se muestra como el sistema fue adaptando la cantidad de réplicas según la tasa de entrada, de tal manera de modificar la cantidad de réplicas según la necesidad que se posea. Por ejemplo, en el primer y segundo tercio aumenta la cantidad de réplicas, no así en el último, donde disminuye la cantidad de réplicas, debido que existe un exceso en la cantidad de recursos. El promedio en la tasa de salida en esta prueba es de 72 eventos por segundo. Por otra parte, en la Figura 5.10 existe una tasa de salida constante, lo cual refleja la nula adaptabilidad del sistema según el flujo entrante, habiendo un promedio de 19 eventos procesados por segundo. De los resultados presentados, se puede observar una clara mejora en el rendimiento, alcanzado a procesar hasta 3 veces más eventos que la versión sin el modelo elástico.

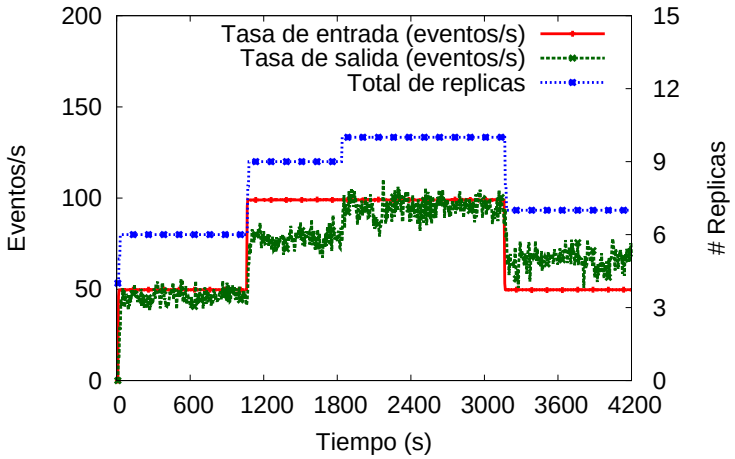

Figura 5.9: Rendimiento y cantidad de réplicas totales del grafo en la primera aplicación con envío variable de la fuente de datos con uso del modelo.

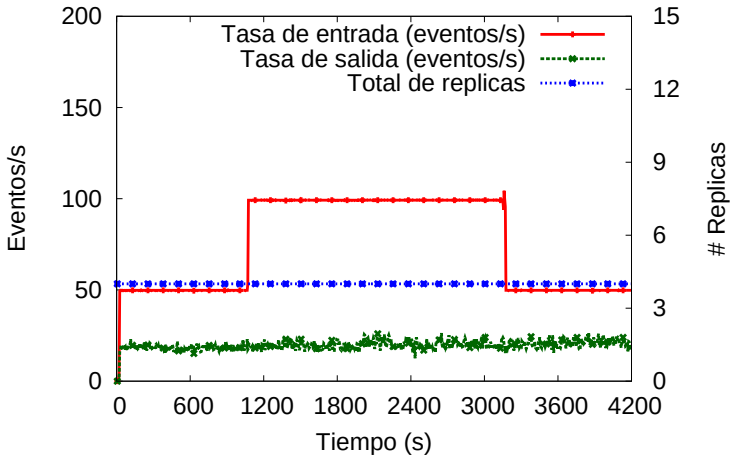

Figura 5.10: Rendimiento y cantidad de réplicas totales del grafo en la primera aplicación con envío variable de la fuente de datos sin uso del modelo.

Por otra parte, la cantidad total de eventos procesados en el experimento, se observa en las Figuras 5.11 y 5.12 con y sin uso del modelo respectivamente. En el primer gráfico se observa variaciones en la curva en el segundo 1100 y 3200, lo cual son los segundos en los cuales hubo un cambio en el flujo de la fuente de datos, por lo que aumenta y disminuye respectivamente la tasa de llegada. En cambio, en el segundo gráfico no se aprecia esto, debido que independiente de la tasa de llegada, el procesamiento del sistema es constante, sin adaptarse el sistema a la carga que posea cada operador con el transcurso del tiempo. El sistema con uso del modelo procesa un total de 303.156, contra 82.770 eventos en total sin uso del modelo, lo cual significa un aumento de 3 veces más eventos procesados.

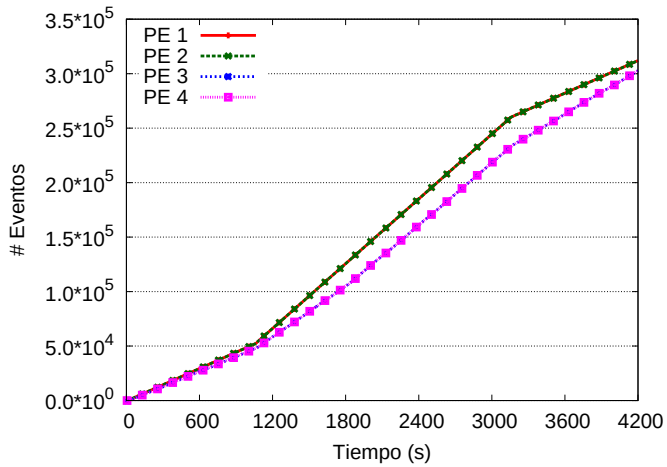

Figura 5.11: Cantidad total de eventos procesados en la primera aplicación con un envío variable de la fuente de datos con uso del modelo.

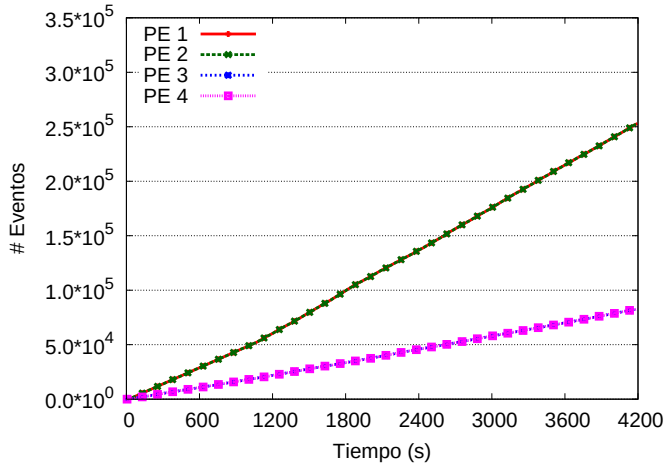

Figura 5.12: Cantidad total de eventos procesados en la primera aplicación con un envío variable de la fuente de datos sin uso del modelo.

5.3.2 Aplicación 2: Contador de palabras en muestras de textos

En la segunda aplicación se ha procedido a realizar dos experimentos, los cuales son similares a los anteriores. En el primer experimento se realiza un envío constante de los eventos, y en el segundo, un envío variable. En ambos experimentos, se han realizado dos pruebas, donde la primera consiste en la ejecución de la aplicación en un sistema SPS que usa el modelo elástico, y la segunda sin el uso de éste.

Para el análisis de los experimentos, se ha considerado el rendimiento y la cantidad de réplicas del grafo y la cantidad total de eventos procesados. En el Anexo E se presentan las estadísticas de cada uno de los operadores.

La Figura 5.13 corresponde al primer experimento y muestra el flujo de datos entrantes del sistema, y como varía la cantidad de réplicas totales del grafo según el flujo de datos. En esta figura se observa que al principio de la ejecución de la aplicación, existen altas variaciones con el flujo de eventos entrante, y eso se debe a que al crear una réplica del operador sobrecargado, el PE Counter (véase Anexo E), genera una sobrecarga a nivel físico en la máquina, habiendo retrasos en el procesamiento de los datos, lo cual no afecta en el análisis del grafo lógico. Por otra parte, en este experimento se muestra una replicación basado en el algoritmo predictivo, lo cual se debe al análisis del rendimiento del operador según la tasa de rendimiento del PE Counter.

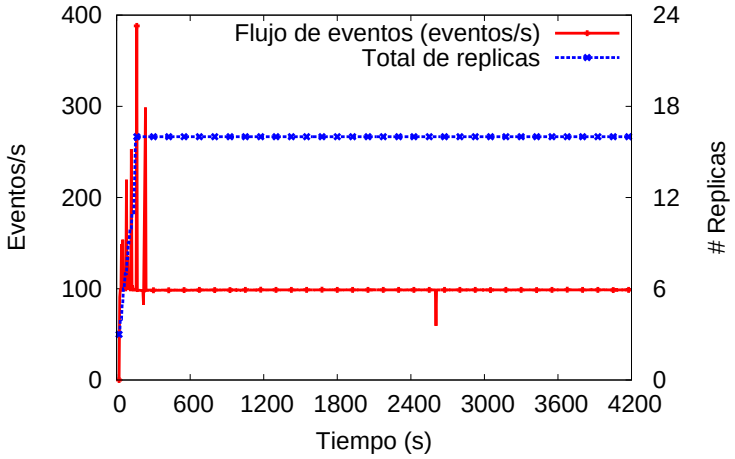

Figura 5.13: Flujo de datos y cantidad de réplicas totales del grafo en la primera aplicación con envío variable de la fuente de datos con uso del modelo.

Por otra parte, en las Figuras 5.14 y 5.15 se muestra la cantidad total de eventos procesados. En el primer gráfico se observa 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, por lo que se puede procesar mayor cantidad de eventos, alcanzando un total de 275.290 eventos con uso el modelo contra 28.152 sin uso del modelo. En este caso se logro procesar hasta 9 veces mas eventos haciendo uso del modelo elástico.

Dentro de las observaciones importantes está lo ocurrido en la Figura 5.14, donde no existe una mejora por parte del segundo PE de manera paralela al flujo de datos emanado por el primer PE. Como se había explicado anteriormente, independiente que se generen más réplicas, los operadores igualmente van a encolar debido a la sobrecarga surgida por parte de la máquina física.

En el PE 3 (PE Merge) se presenta una baja cantidad de eventos procesados, lo cual se observa en las Figuras 5.14 y 5.15. Esto se debe a su condición de operador auxiliar. Cabe recordar que se denomina operador auxiliar a los PEs predecesor y sucesores de un operador

con estado, de esta manera, estos operadores se dedican a dividir la información para enviarla a las distintas réplicas del operador, y posteriormente juntar la información obtenida por las distintas réplicas de éste. La baja cantidad de eventos entrantes se debe a que los eventos entrantes sólo son enviados cada 10 segundos por el PE 2 (PE Counter), y en caso de procesar mayor cantidad de datos y poseer mayor cantidad de réplicas de este operador, mayor es la cantidad de eventos procesados por el PE 3. Por último, en la aplicación con uso del modelo elástico se han procesado 3.491 eventos, contra 34 eventos procesados sin el modelo elástico.

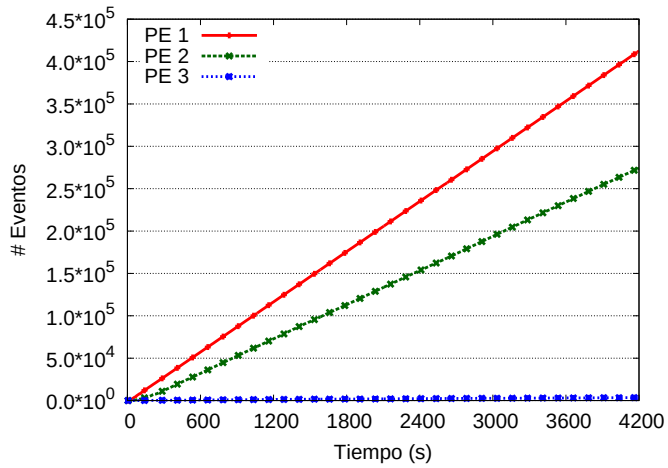

Figura 5.14: Cantidad total de eventos procesados en la segunda aplicación con un envío constante de la fuente de datos sin uso del modelo.

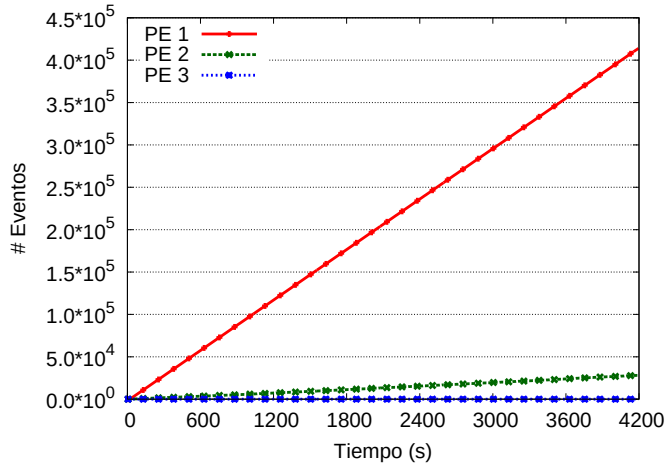

Figura 5.15: Cantidad total de eventos procesados en la segunda aplicación con un envío constante de la fuente de datos con uso del modelo.

La Figura 5.16 corresponde al primer experimento y muestra el flujo de datos entrantes del sistema, y como varía la cantidad de réplicas totales del grafo según el flujo de datos. En esta figura se puede observar como el número de réplicas se adapta al tráfico recibido. En el primer y segundo tercio se muestra un aumento de réplicas, pero posteriormente en el último

disminuye, aprovechando los recursos que se disponen. A diferencia del experimento anterior, en éste no se activa el algoritmo predictivo, y eso se debe a qué no existe una posible sobrecarga a futuro, dado que se estabiliza el sistema antes de este análisis debido al algoritmo reactivo.

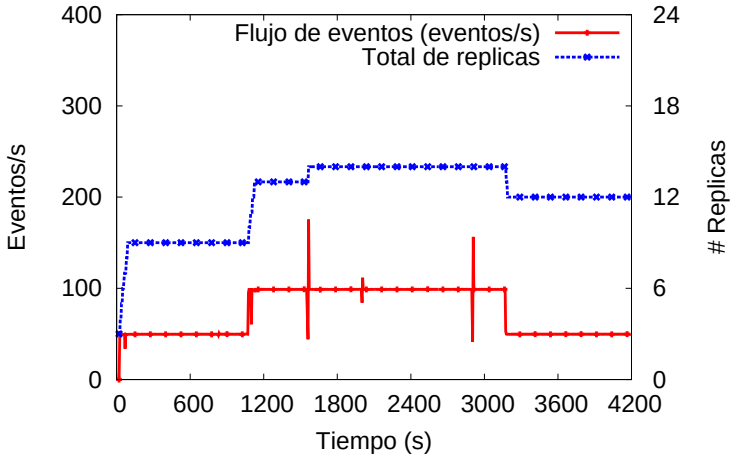

Figura 5.16: Flujo de datos y cantidad de réplicas totales del grafo en la primera aplicación con envío variable de la fuente de datos con uso del modelo.

Por otro lado, en las Figuras 5.17 y 5.18 se muestra la cantidad total de eventos procesados. En el primer gráfico se aprecia que la curva del primer PE y el segundo PE es más cercana que las del segundo gráfico, y esto se debe a la replicación que se ha efectuado en el sistema. Cabe destacar que el sistema con uso del modelo ha procesado un total de 228.942 eventos, mientras que el sistema sin uso del modelo ha procesado 27.751 eventos. Por lo que la solución permite aumentar en más de 8 veces la cantidad de eventos procesados.

El análisis que se realiza en el PE 3 (PE Merge) es el mismo explicado en el anterior experimento, donde la cantidad de eventos procesados va a depender de la cantidad de réplicas existentes en el PE 2 (PE Counter). Por último, se ha realizado un procesamiento de 1.578 eventos con uso del modelo, y 27 eventos sin uso del modelo.

5.3.3 Aplicación 3: Aplicación sintética

En la tercera aplicación se ha procedido a realizar un experimento que consta de dos pruebas, al igual que en las anteriores pruebas, donde una de ellas hace uso del modelo elástico y la otra no. Ambas pruebas se han realizado con un envío constante de 100 eventos por segundo desde la fuente de datos.

Para el análisis de los experimentos, se ha considerado el consumo de memoria, el uso de la CPU, el rendimiento, la cantidad de réplicas del grafo y la cantidad total de eventos

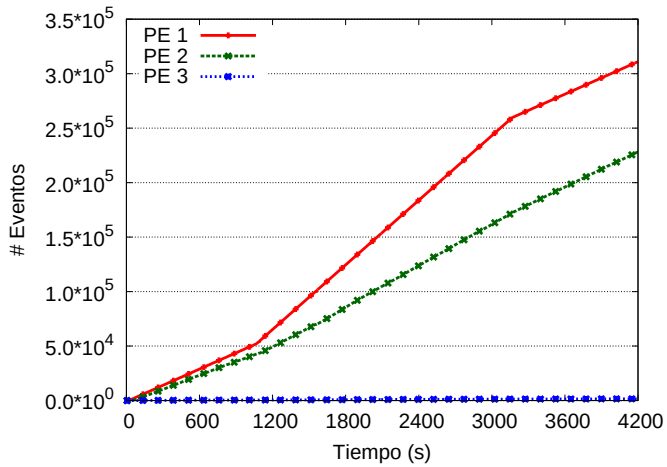

Figura 5.17: Cantidad total de eventos procesados en la segunda aplicación con un envío variable de la fuente de datos con uso del modelo.

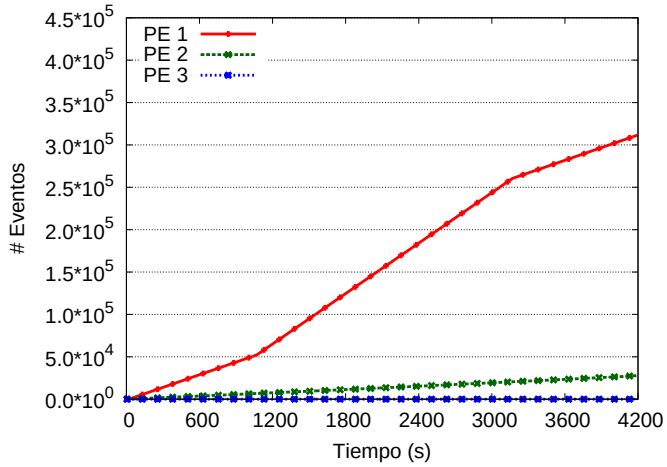

Figura 5.18: Cantidad total de eventos procesados en la segunda aplicación con un envío variable de la fuente de datos sin uso del modelo.

procesados. En el Anexo E se presentan las estadísticas de cada uno de los operadores.

Respecto a la utilización de CPU, se puede observar en las Figuras 5.19 y 5.20 el porcentaje de uso con y sin uso del modelo respectivamente. Si bien la diferencia no es significante en el uso de la CPU por parte de la aplicación, en el primer caso existe un promedio de 0.62% de utilización de CPU, contra un promedio de 0.61% de uso en el segundo caso, habiendo un aumento del $0,01\%$ de utilización promedio de CPU. Dentro de los primeros 10 segundo existe un alto uso de CPU en ambos gráficos, lo que se debe al despliegue y compilación de la aplicación en el sistema de S4.

Por otra parte, el consumo de memoria RAM es presentado por las Figuras 5.21 y 5.22, donde la primera es con uso de modelo y la segunda sin uso. En el primer gráfico existe un consumo promedio de 264 MB versus 268 MB del segundo gráfico, habiendo una disminución del

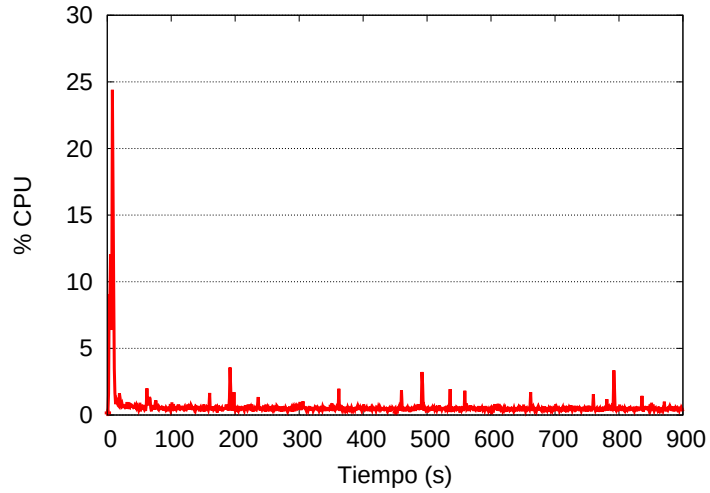

Figura 5.19: Porcentaje de utilización de la CPU en la tercera aplicación con uso del modelo.

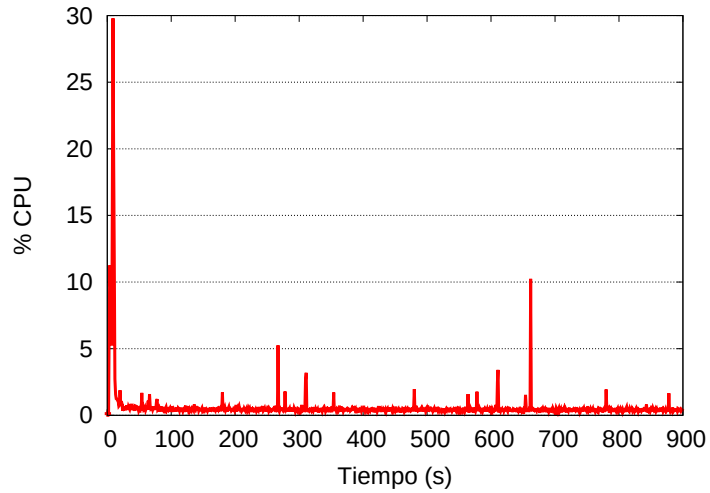

Figura 5.20: Porcentaje de utilización de la CPU en la tercera aplicación sin uso del modelo.

$1,5\%$ de consumo de memoria RAM. En el primer gráfico se observa un aumento del consumo de memoria en el segundo 200, a diferencia del segundo gráfico, debido al mayor consumo de eventos y creación de nuevos operadores. Posterior al segundo 600, por parte del sistema sin uso del modelo, se muestra un aumento del consumo de memoria, lo cual se debe a la cantidad de eventos en la cola que existen en el sistema, a diferencia del primer gráfico que hubo una convergencia en el consumo de la memoria, por lo que posteriormente al reducir las colas, reduce igualmente el consumo de memoria principal en la ejecución.

En las Figuras 5.23 y 5.24 corresponden al rendimiento que posee el sistema, y como éste varía la cantidad de réplicas totales del grafo según la tasa de entrada. En la Figura 5.23 se observa que existe un incremento en la cantidad de réplicas, debido a la sobrecarga de los operadores del grafo (véase Anexo E), aumentando el rendimiento de éste, de tal manera de

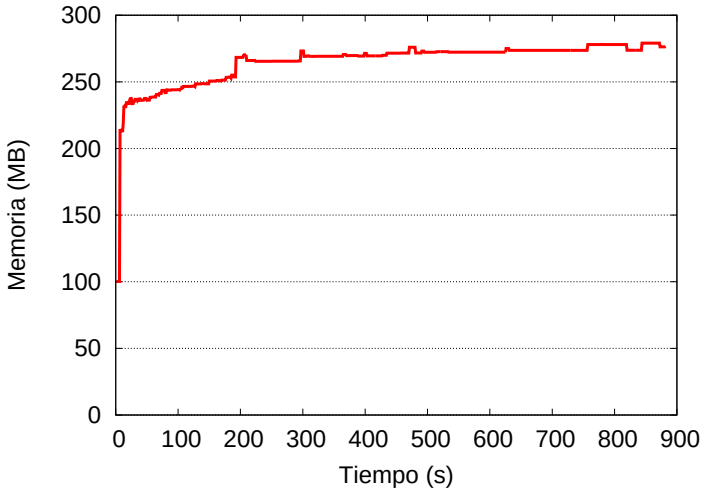

Figura 5.21: Consumo de memoria RAM en la tercera aplicación con uso del modelo.

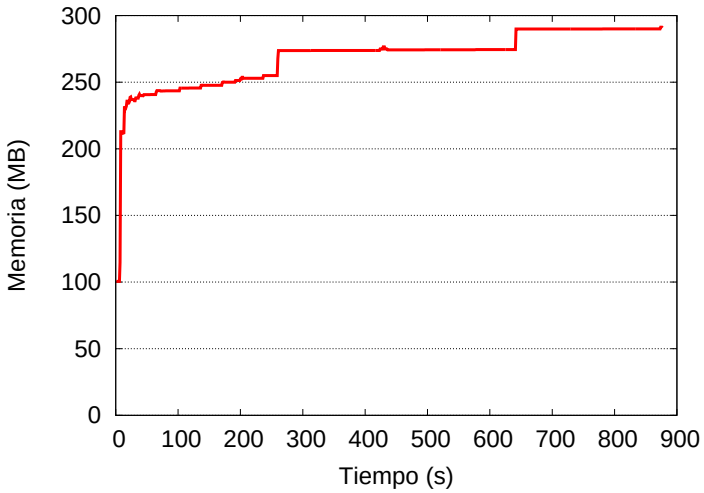

Figura 5.22: Consumo de memoria RAM en la tercera aplicación sin uso del modelo.

procesar 97 eventos por segundo en promedio con el uso del modelo. En cambio, en la Figura 5.24 no existe un aumento en el rendimiento, teniendo una tasa de salida promedio de 33 eventos por segundo. De esta manera, existe una mejora de 2 veces más eventos procesados con el uso del modelo propuesto.

En cuanto a la cantidad total de eventos procesados, en las Figuras 5.25 y 5.26 se aprecia que la cantidad de eventos procesados es mayor en el primero gráfico, el cual hace uso del modelo elástico. En el primer gráfico existe un total de 88.169 eventos procesados y en el segundo un total de 29.714 eventos procesados, existiendo una mejora de 3 veces la cantidad de eventos procesados.

Finalmente, en la Tabla 5.2 se muestra el tiempo promedio de ejecución de cada uno de los algoritmos para el análisis de un operador. Se puede observar que si bien el algoritmo

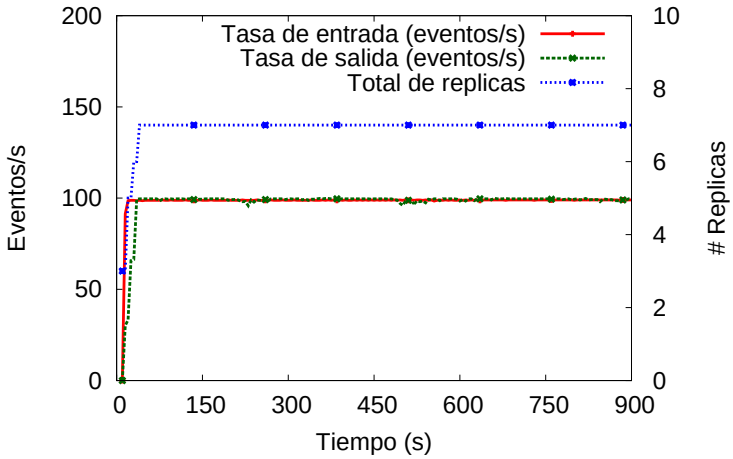

Figura 5.23: Rendimiento y cantidad de réplicas totales del grafo en la tercera aplicación con uso del modelo.

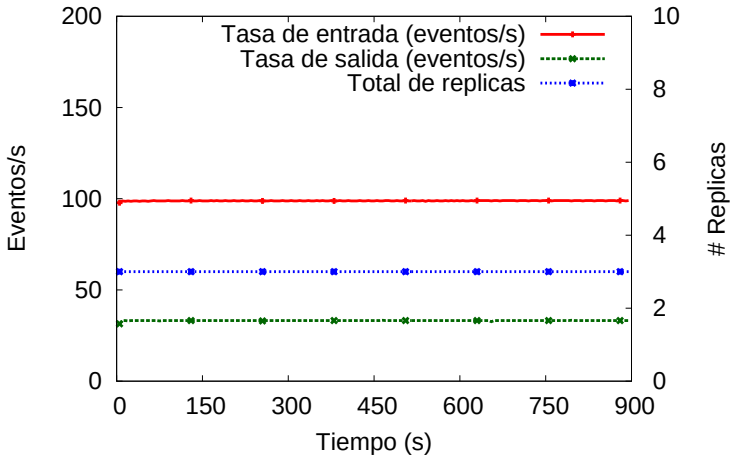

Figura 5.24: Rendimiento y cantidad de réplicas totales del grafo en la tercera aplicación sin uso del modelo.

predictivo posee un tiempo de ejecución mayor que el algoritmo reactivo, no produce un alto costo computacional su implementación.

Tabla 5.2: Tiempos de ejecución de los algoritmos del modelo elástico.

Algoritmo	Tiempo (ms)
Reactivo	0.03
Predictivo	4.63

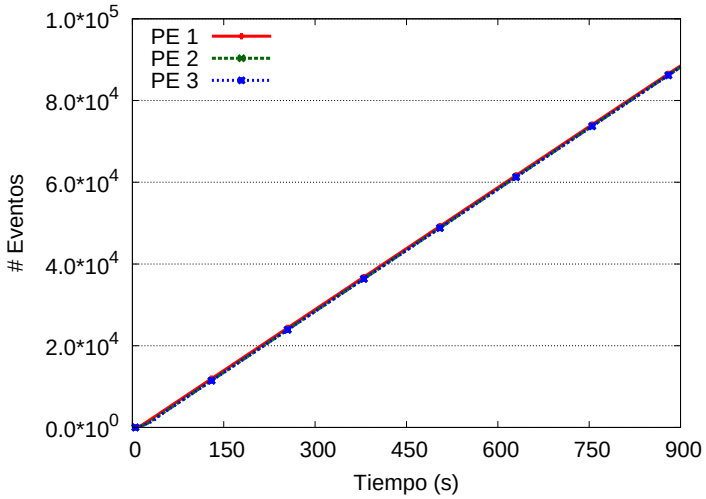

Figura 5.25: Cantidad de eventos procesados en la tercera aplicación con uso del modelo.

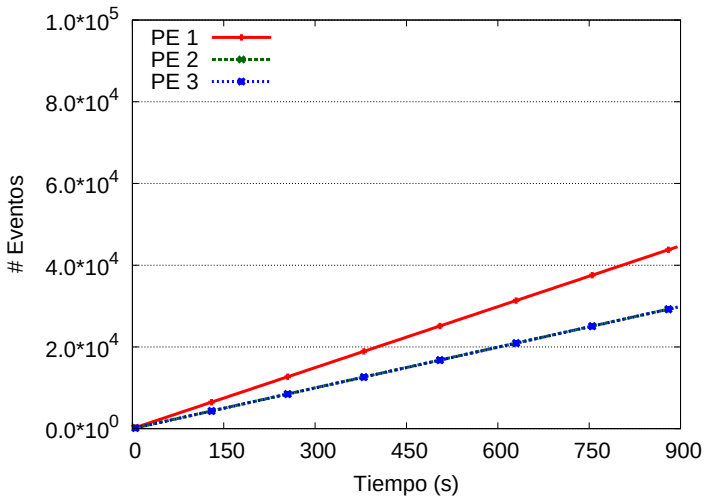

Figura 5.26: Cantidad de eventos procesados en la tercera aplicación sin uso del modelo.

CAPÍTULO 6. CONCLUSIONES

En este trabajo se ha propuesto la generación de un sistema elástico, cuyo objetivo es lograr una mejor utilización de los recursos disponibles en un sistema y con ello un aumento en su capacidad de procesamiento y escalabilidad.

6.1 DETALLES DE LA CONTRIBUCIÓN

Dentro de las contribuciones de este trabajo se encuentra el diseño e implementación de un modelo elástico que es capaz de lidiar con el dinamismo del flujo de eventos o tráfico de datos. En este modelo se diseñaron cuatro módulos, los cuales estaban compuestos por un módulo de monitoreo, que recolecta las estadísticas, un módulo reactivo y predictivo, que estima la carga de cada operador en el presente y a futuro respectivamente, y un módulo de administración de réplicas, que aumenta o disminuye el número de réplicas de un operador según la carga de éste.

El módulo reactivo que se ha diseñado e implementado tiene como función analizar la carga actual del operador, cumpliendo así el primer objetivo de este trabajo, el cual consiste en diseñar e implementar un algoritmo reactivo que permita analizar en el momento la carga de los operadores.

Por otra parte, se ha diseñado e implementando el módulo predictivo, cuya función es estimar la carga de un operador en una ventana de tiempo futura, cumpliendo el segundo objetivo, el cual consiste en diseñar e implementar un algoritmo de predicción que permita estimar la carga de los operadores.

Así mismo, se ha diseñado e implementado un módulo de administración de réplicas, el cual se encarga de administrar la cantidad de réplicas de los operadores del SPS de forma elástica, vale decir, que aumenta o disminuye el número de réplicas acorde al tráfico recibido.

Para validar el modelo elástico se han construido tres escenarios para la experimentación, donde el primero consiste en una aplicación que realiza operaciones sin estados, la segunda una aplicación que realiza operaciones con estados, y la tercera una aplicación sintética. De esta manera, el objetivo es evaluar el rendimiento del sistema utilizando el modelo elástico.

Haciendo uso de los escenarios, se ha evaluado y analizado el rendimiento del sistema con y sin uso del modelo elástico. Los resultados muestran que para todos los escenarios se ha mejorado la cantidad de eventos procesados. Dependiendo del tipo de escenario, se ha detectado un aumento de hasta 9 veces en el *throughput* de la aplicación. Este es un resultado importante, puesto que al poseer una mayor cantidad de datos procesados, se puede lograr una

mayor precisión en la información obtenida. Por otra parte, el costo asociado a la implementación del modelo elástico en relación a la CPU es de un aumento del $0,01\%$, sin embargo la memoria RAM utilizada ha disminuido en un $1,51\%$.

De esta manera, podemos ver que se han alcanzado todos los objetivos planteados, y la hipótesis establecida se ha validado. El modelo cumple con hacer elástico el SPS, aumentando la cantidad de datos procesos acorde al tráfico recibido. Además, se ha concluido que el sistema posee un bajo costo de implementación.

6.2 DISCUSIONES

Uno de los problemas detectados en el desarrollo de este trabajo es la limitación de recursos físicos para la implementación de la aplicación diseñada. Esto se debe a que la cantidad de eventos entrantes no son todos procesados, independientemente si se genera una mayor cantidad de réplicas o no. Este problema fue detectado en la fase de experimentación, donde al tratar de realizar pruebas con un tiempo de ejecución mayor, existe una disminución considerablemente de la tasa de rendimiento de un operador después de un largo tiempo de ejecución, debido a que el *buffer* del operador se llena al no procesar todos los datos entrantes, bloqueando el envío de eventos a éste.

Dentro de las limitaciones del trabajo está el cálculo de la tasa de procesamiento, la cual se considera homogénea en todo en transcurso de la ejecución del sistema. Esto significa que se calcula un valor al principio de la ejecución del sistema, el cual indica cual es la cantidad de eventos que procesa por segundo, de tal manera de considerar esa tasa de procesamiento en los cálculos de la tasa de rendimiento de los operadores. En caso que no se considere esto, se tiene que calcular una tasa de procesamiento en ventanas de tiempo, lo cual puede producir un porcentaje de error, debido que la tasa de procesamiento de la ventana de tiempo anterior no sea la misma que la actual.

Si bien el modelo diseñado realiza un análisis del sistema lógico, no considera un análisis de los recursos físicos disponibles por parte del sistema, es decir, uso de la CPU, capacidad de la memoria RAM, entre otras métricas. Cabe destacar, que en el caso que la carga de un operador aumente en conjunto con la replicación, existe la limitación física de la máquina, debido a la capacidad de la CPU y memoria que éste posea, limitando la capacidad de procesamiento del SPS.

Por otra parte, el sistema no es capaz de detectar patrones estacionarios que puedan existir en el día, lo cual es una desventaja en la implementación de éste. Esto se debe a que el algoritmo predictivo analiza procesos estocásticos, y no un aprendizaje del comportamiento del

flujo de datos, como lo realizan así modelos predictivos tipo *machine learning* (Mohri et al., 2012). Sin embargo, estas soluciones son de alto costo y limitan la escalabilidad del SPS.

Finalmente, el sistema diseñado presenta como ventaja poseer un bajo cómputo para el cálculo del número de réplicas. También se destaca el rápido análisis de los operadores, ya sea en la distribución de carga en cada uno de los operadores, o en el estado que se encuentra el operador, de tal manera de modificar la cantidad de réplicas existentes. De esta manera, al poseer un sistema elástico, se logra optimizar el uso de los recursos existentes y adquirir un dinamismo en el grafo de la aplicación ejecutada sobre el SPS.

6.3 TRABAJO FUTURO

Dentro de los problemas abiertos que pueden resolverse a futuro en el modelo propuesto, se encuentran: el diseño de un predictor que indique dinámicamente cuántas son las réplicas necesarias según el historial y la implementación del modelo diseñado en otro SPS.

Actualmente el modelo propuesto asume de manera estática un número fijo de réplicas a generar, se podría realizar un análisis más detallado de la historia, de tal manera que según el comportamiento que éste posea, estimar cuantas serían las réplicas a aumentar o disminuir. Así mismo, se podría realizar un estudio respecto a los *peaks* de tráfico que se encuentren de forma estacionaria según cierto flujo de datos, y que el sistema se adapte dinámicamente a estos. Para realizar estos estudios, se puede añadir la capacidad de aprendizaje utilizando *machine learning* (Mohri et al., 2012) al modelo propuesto. De esta manera, éste aprendería los comportamientos respectivos del tráfico, de manera de mejorar la predicción y con ello la utilización de recursos.

Por otra parte, sería interesante poder implementar el modelo elástico en otro SPS, ya sea Storm (Storm, 2014) o StreamIt (Thies et al., 2002), debido a los distintos problemas que surgieron al utilizar el S4. De esta manera, se podría realizar una comparación de cual son los distintos pro y contra de los SPS con el sistema implementando, y en que casos es mejor utilizar uno u otro dependiendo del modelo o escenario que estos utilicen.

REFERENCIAS BIBLIOGRÁFICAS

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).

Arasu, A., Cherniack, M., Galvez, E. F., Maier, D., Maskey, A., Ryvkina, E., Stonebraker, M., & Tibbetts, R. (2004). Linear road: A stream data management benchmark. In *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004*, (pp. 480–491).

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).

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

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.

Cooper, R. B. (1972). *Introduction to queueing theory*. Macmillan.

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

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

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*. Fakultat fur Mathematik und Informatik.

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.

Goetsch, K. (2014). *eCommerce in the Cloud - Bringing Elasticity to eCommerce*. O'Reilly.

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 *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).

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).

Jacob, B., Lanyon-Hogg, R., Nadgir, D. K., & Yassin, A. F. (2004). *A Practical Guide to the IBM Autonomic Computing Toolkit*. IBM Corporation, International Technical Support Organization.

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.

Kephart, J., Kephart, J., Chess, D., Boutilier, C., Das, R., Kephart, J. O., & Walsh, W. E. (2005). An architectural blueprint for autonomic computing. *IBM White paper*.

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.

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.

Menin, E. (2002). *The Streaming Media Handbook*. Pearson Education.

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).

Mohri, M., Rostamizadeh, A., & Talwalkar, A. (2012). *Foundations of Machine Learning*. Adaptive computation and machine learning. MIT Press.

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., & Pillai, U. (1984). *Probability, Random Variables, and Stochastic Processes*. McGraw 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/`.

Sampieri, R. H., Collado, C. F., & Lucio, P. B. (2010). Metodología de la investigación. *México: Editorial Mc Graw Hill*.

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).

Services, A. W. (2015). Amazon elastic compute cloud (ec2). [Online] `https://aws.amazon.com/ec2/`.

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*. Sams publishing.

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.

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

Wang, X., & Loguinov, D. (2007). Load-balancing performance of consistent hashing: asymptotic analysis of random node join. *IEEE/ACM Trans. Netw.*, *15*(4), 892–905.

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).

Wu, S., Kumar, V., Wu, K., & Ooi, B. C. (2012). Parallelizing stateful operators in a distributed stream processing system: how, should you and how much? In *Proceedings of the Sixth ACM International Conference on Distributed Event-Based Systems, DEBS 2012, Berlin, Germany, July 16-20, 2012*, (pp. 278–289).

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: $P \leftarrow Matriz[3x3]$ //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: $P_{1,1}++$
6: $\tau 1++$
7: **else if** $\rho_i < 0.5$ **and** $0.5 \leqslant \rho_{i+1} \leqslant 1$ **then**
8: $P_{1,2}++$
9: $\tau 1++$
10: **else if** $\rho_i < 0.5$ **and** $\rho_{i+1} > 1$ **then**
11: $P_{1,3}++$
12: $\tau 1++$
13: **else if** $0.5 \leqslant \rho_i \leqslant 1.5$ **and** $\rho_{i+1} < 0.5$ **then**
14: $P_{2,1}++$
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: $P_{2,2}++$
18: $\tau 2++$
19: **else if** $0.5 \leqslant \rho_i \leqslant 1.5$ **and** $\rho_{i+1} > 1.5$ **then**
20: $P_{2,3}++$
21: $\tau 2++$
22: **else if** $\rho_i > 1$ **and** $\rho_{i+1} < 0.5$ **then**
23: $P_{3,1}++$
24: $\tau 3++$
25: **else if** $\rho_i > 1$ **and** $0.5 \leqslant \rho_{i+1} \leqslant 1.5$ **then**
26: $P_{3,2}++$
27: $\tau 3++$
28: **else**
29: $P_{3,3}++$
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: $P_{i,j} \leftarrow {}^{P_{i,j}}/_{\tau_i}$
37: **end for**
38: **end if**
39: **end for**
40: **return** P //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.

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

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

ANEXO E. ESTADÍSTICAS DE LOS OPERADORES

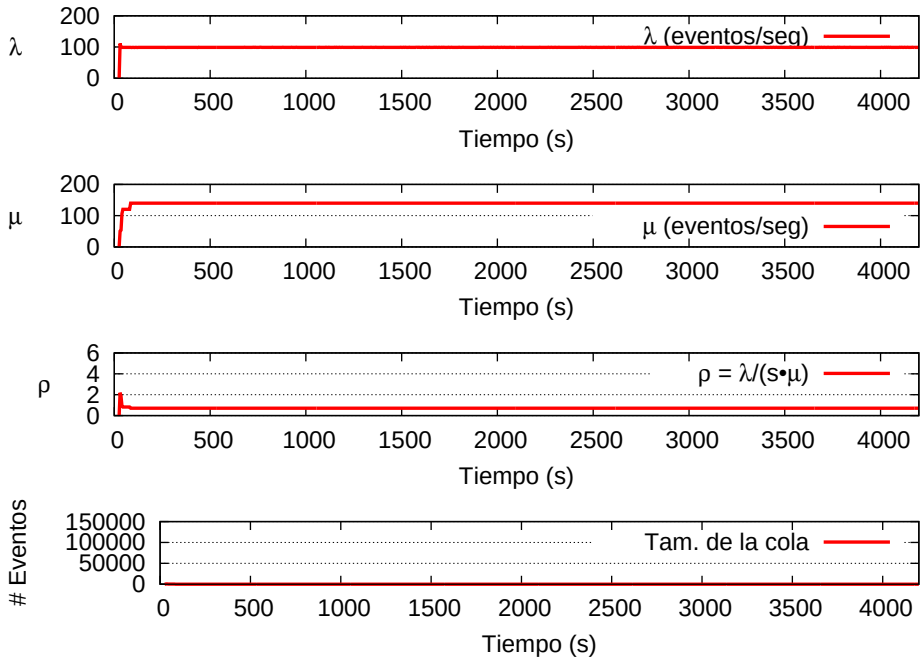

Figura E.1: Estadísticas del PE Stopword en la primera aplicación con un envío constante de la fuente de datos con uso del modelo.

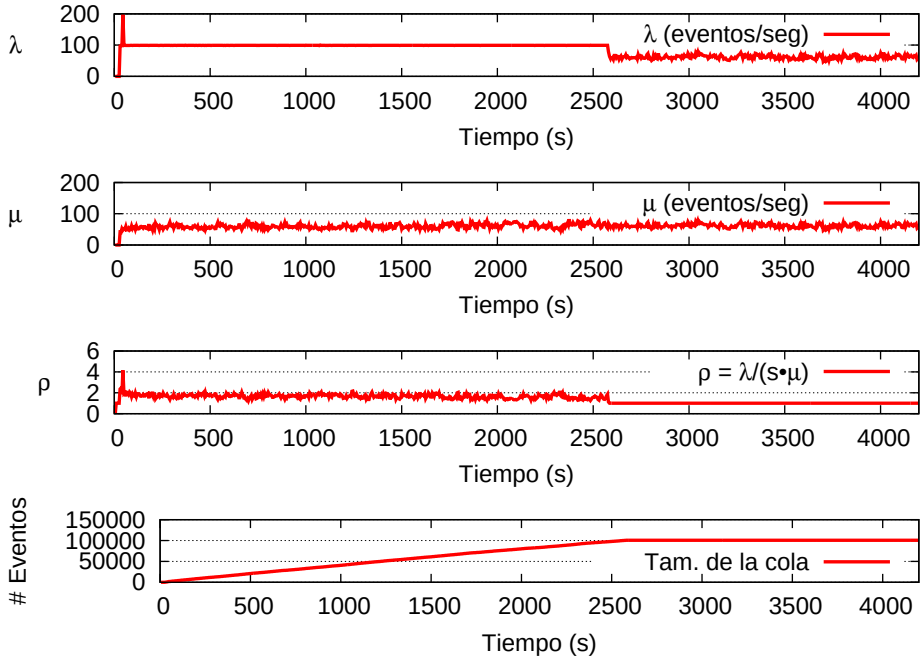

Figura E.2: Estadísticas del PE Stopword en la primera aplicación con un envío constante de la fuente de datos sin uso del modelo.

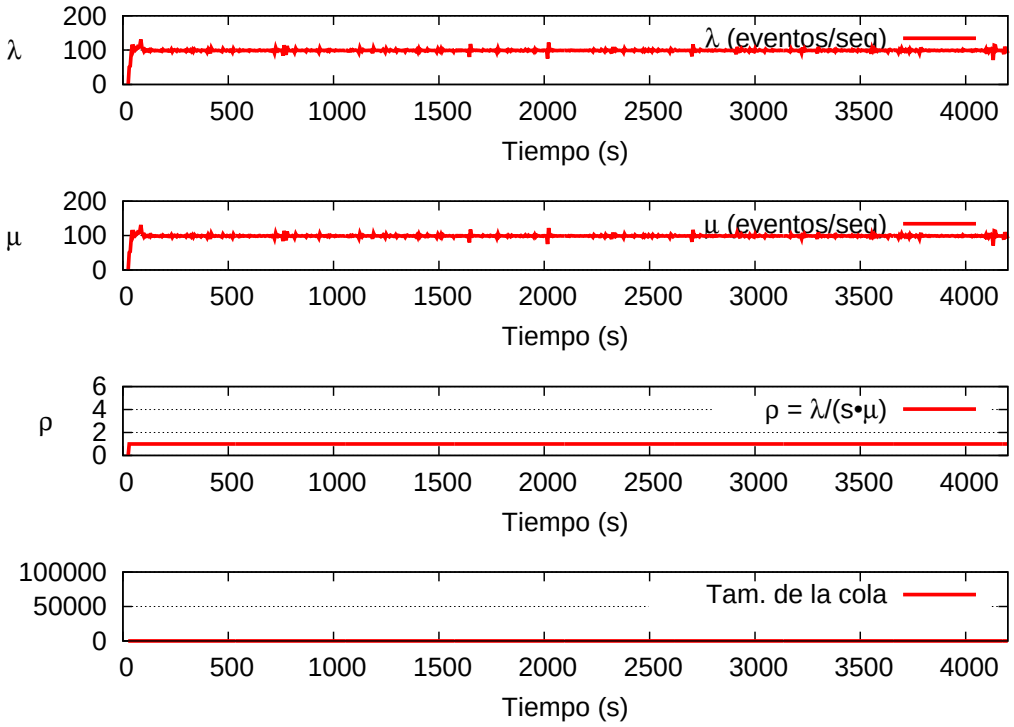

Figura E.3: Estadísticas del PE Language en la primera aplicación con un envío constante de la fuente de datos con uso del modelo.

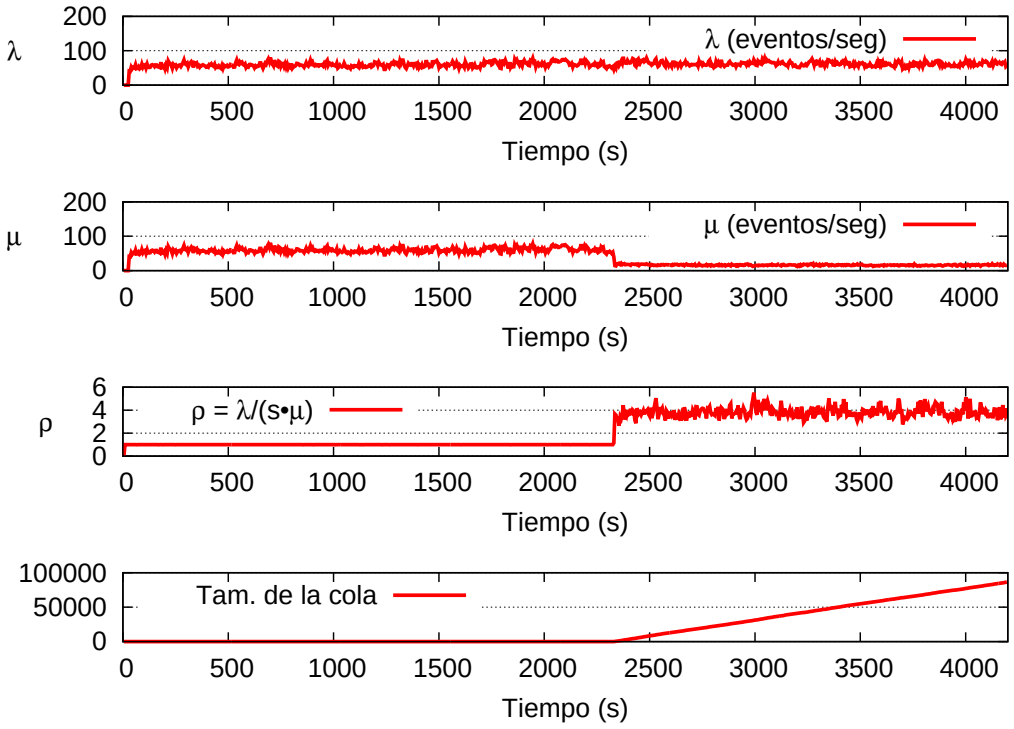

Figura E.4: Estadísticas del PE Language en la primera aplicación con un envío constante de la fuente de datos sin uso del modelo.

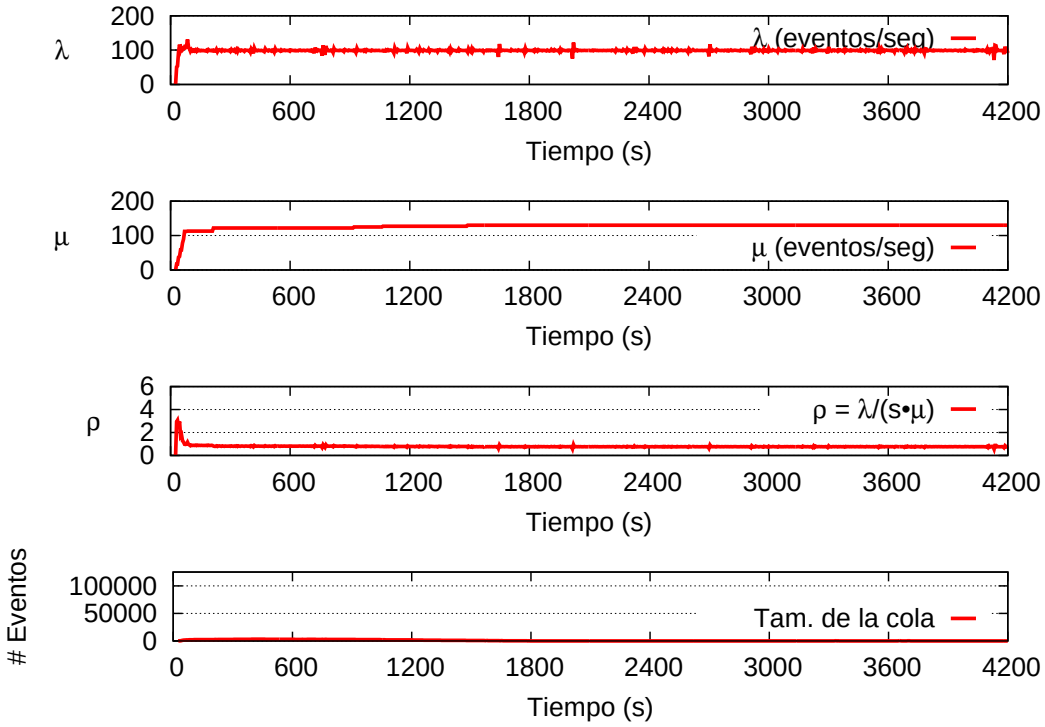

Figura E.5: Estadísticas del PE Counter en la primera aplicación con un envío constante de la fuente de datos con uso del modelo.

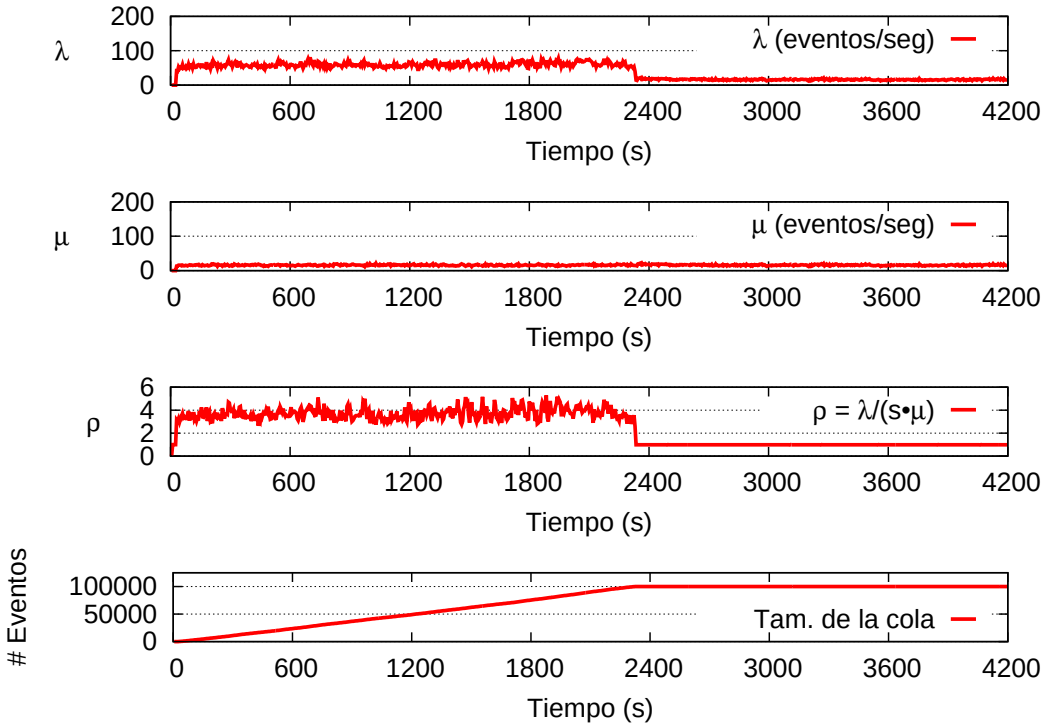

Figura E.6: Estadísticas del PE Counter en la primera aplicación con un envío constante de la fuente de datos sin uso del modelo.

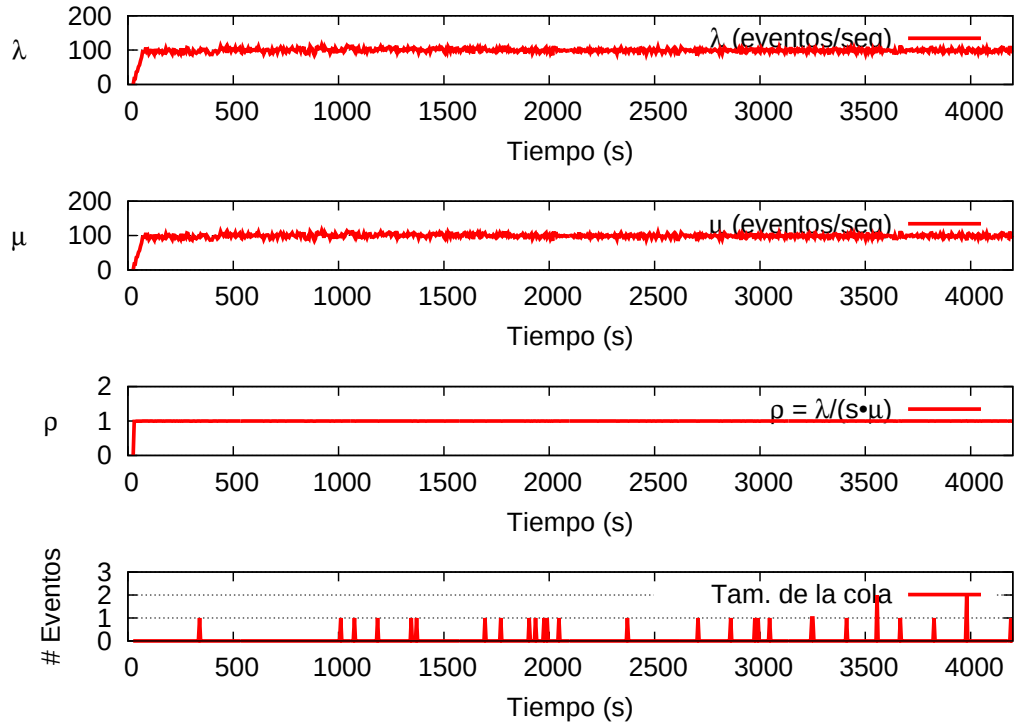

Figura E.7: Estadísticas del PE Mongo en la primera aplicación con un envío constante de la fuente de datos con uso del modelo.

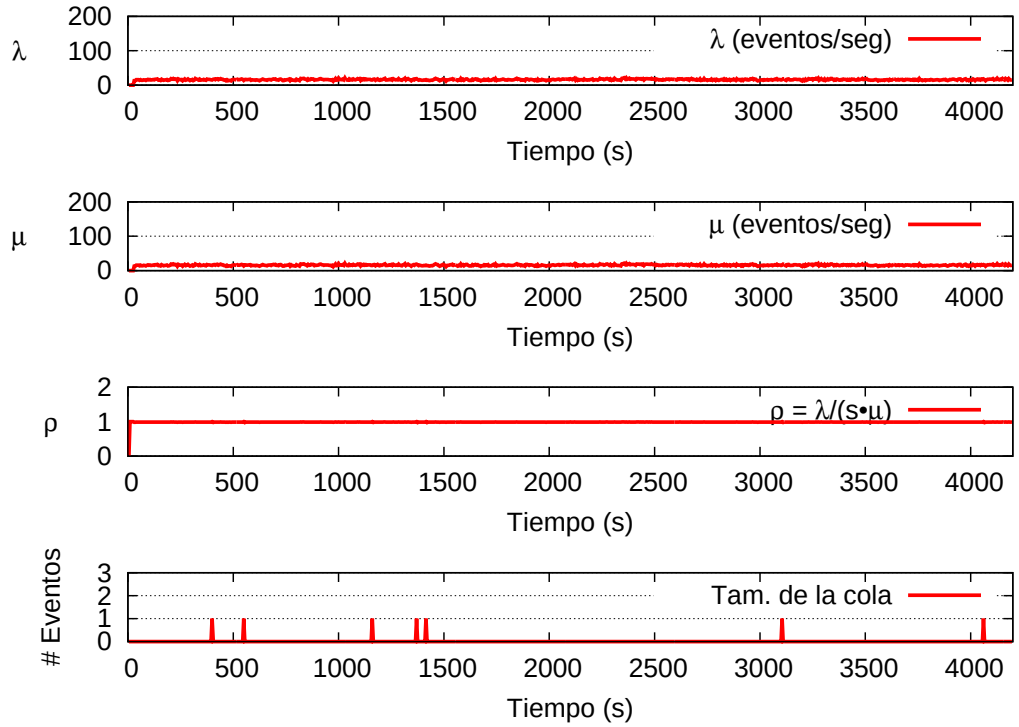

Figura E.8: Estadísticas del PE Mongo en la primera aplicación con un envío constante de la fuente de datos sin uso del modelo.

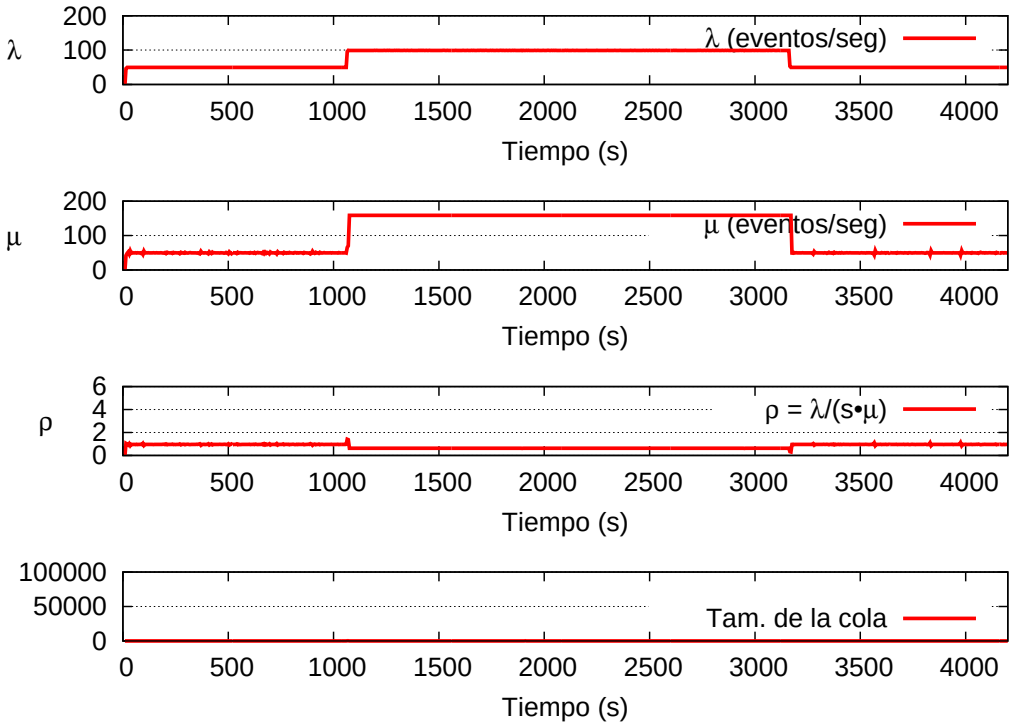

Figura E.9: Estadísticas del PE Stopword en la primera aplicación con un envío variable de la fuente de datos con uso del modelo.

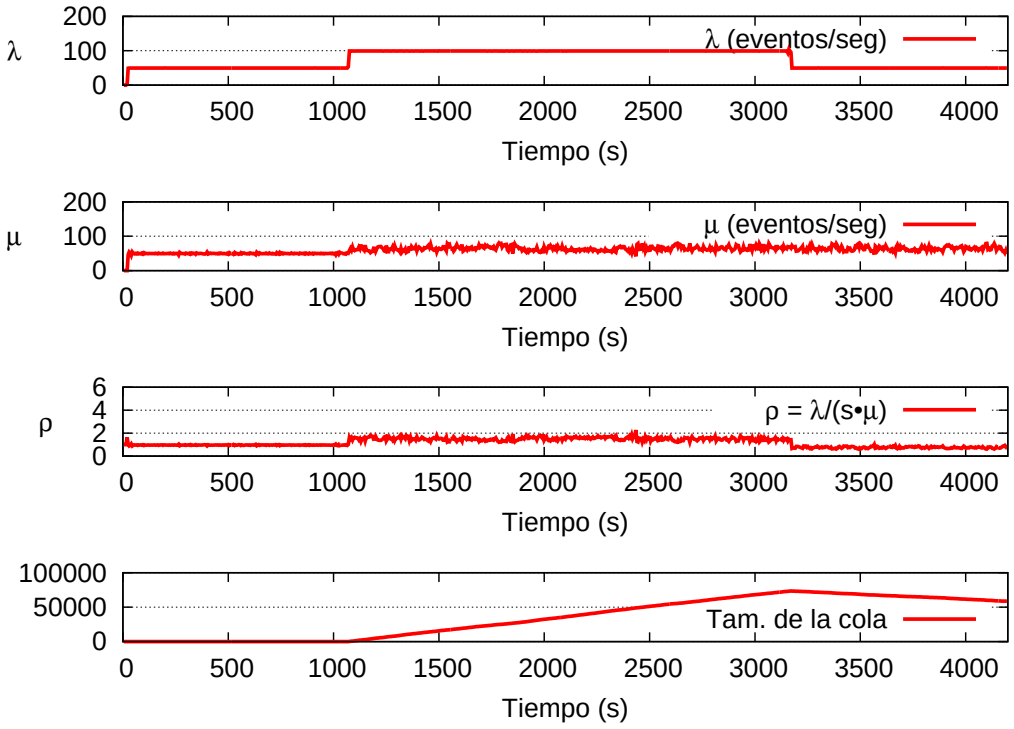

Figura E.10: Estadísticas del PE Stopword en la primera aplicación con un envío variable de la fuente de datos sin uso del modelo.

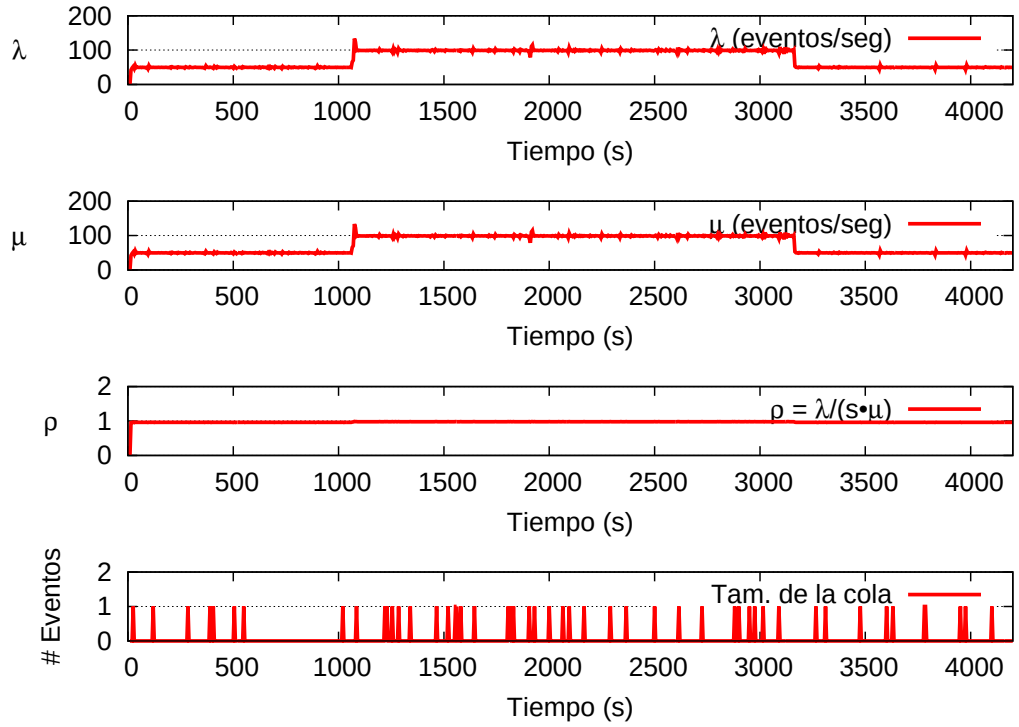

Figura E.11: Estadísticas del PE Language en la primera aplicación con un envío variable de la fuente de datos con uso del modelo.

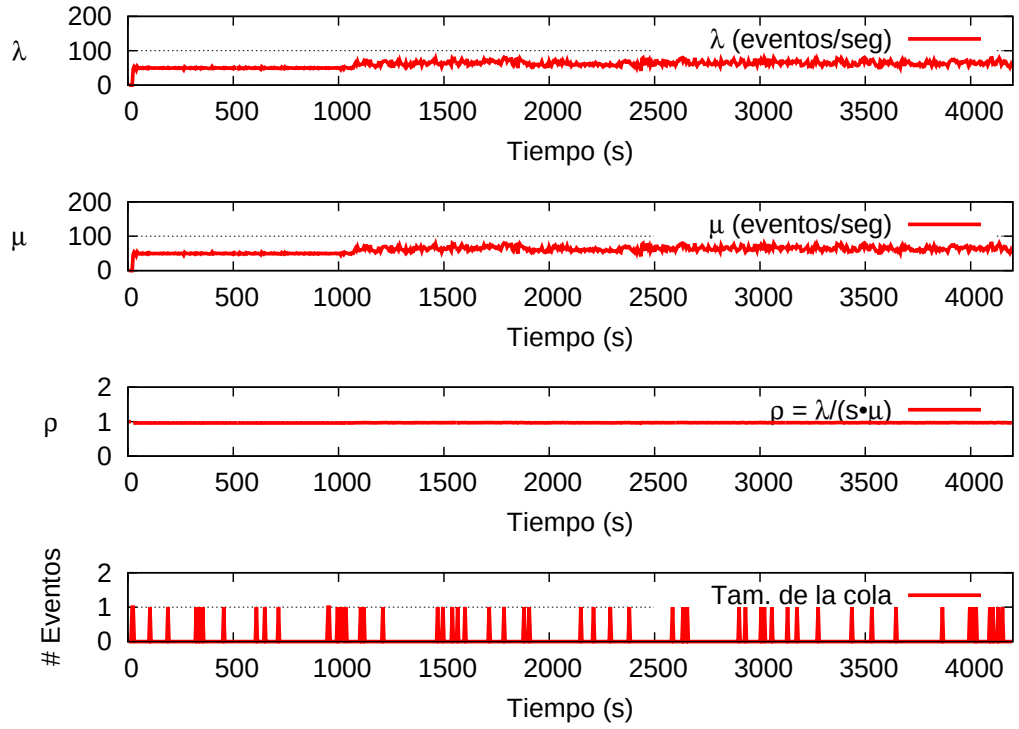

Figura E.12: Estadísticas del PE Language en la primera aplicación con un envío variable de la fuente de datos sin uso del modelo.

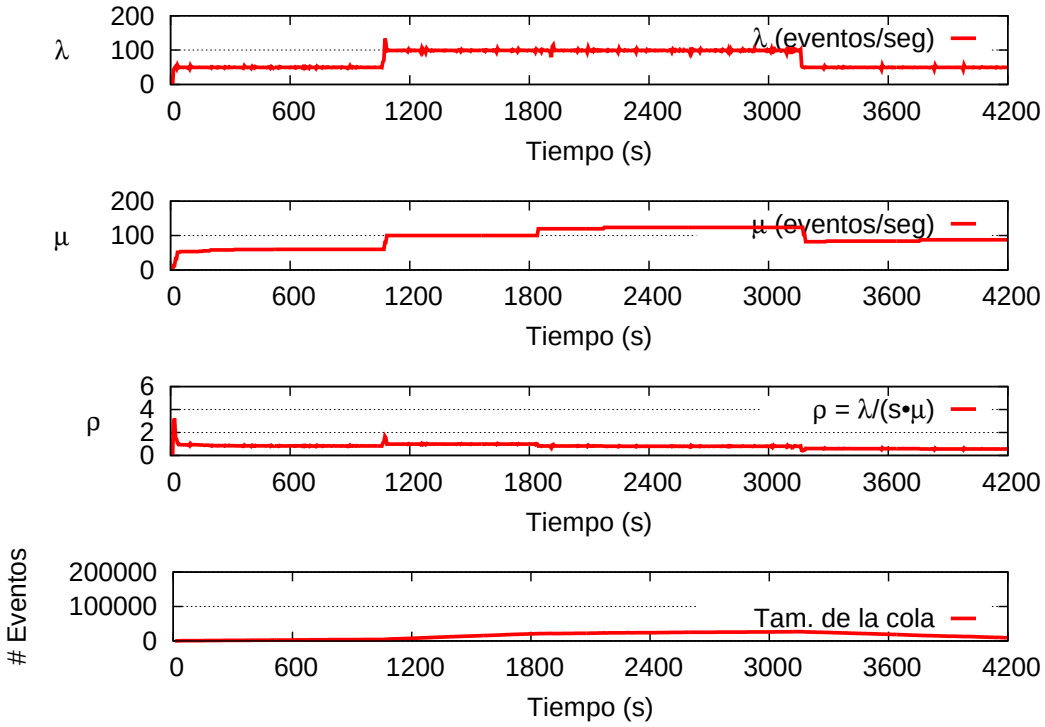

Figura E.13: Estadísticas del PE Counter en la primera aplicación con un envío variable de la fuente de datos con uso del modelo.

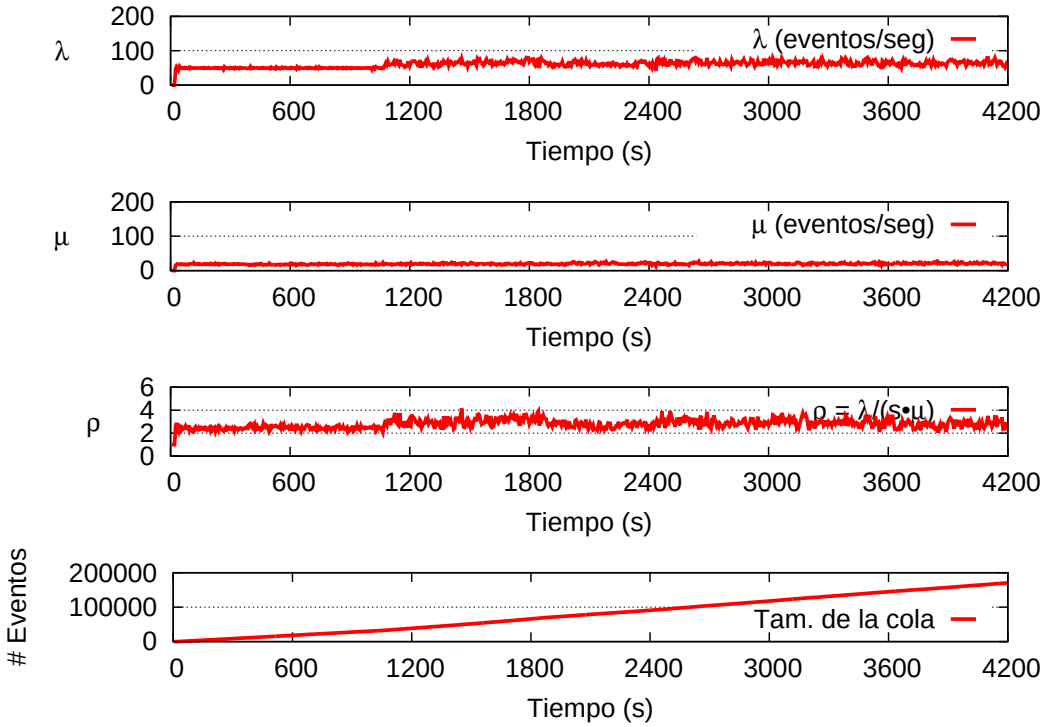

Figura E.14: Estadísticas del PE Counter en la primera aplicación con un envío variable de la fuente de datos sin uso del modelo.

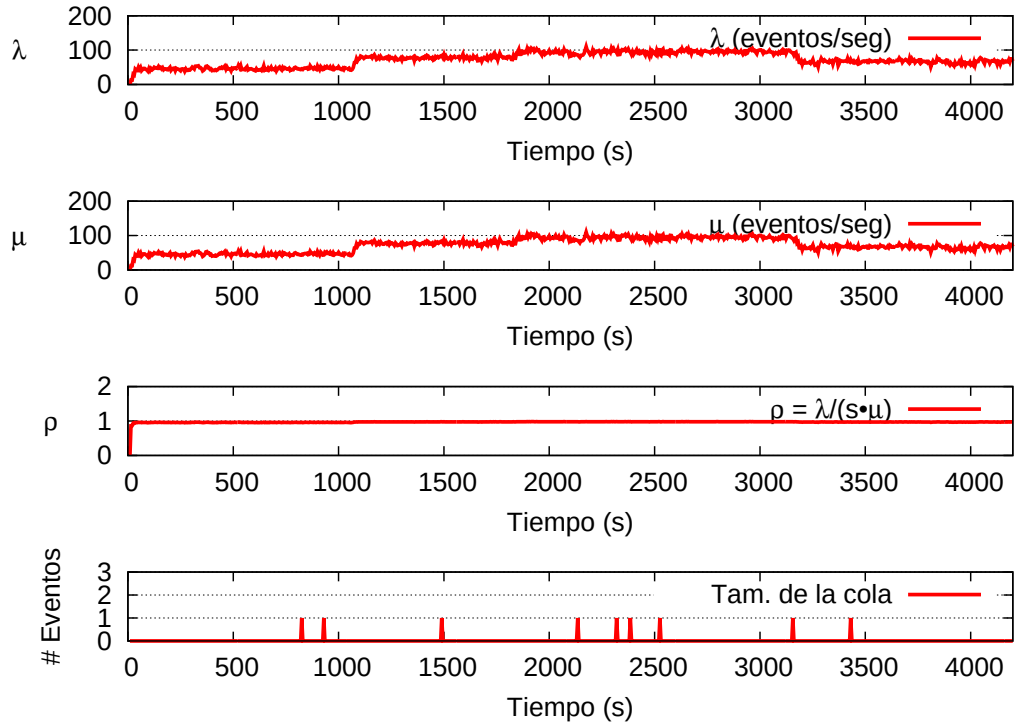

Figura E.15: Estadísticas del PE Mongo en la primera aplicación con un envío variable de la fuente de datos con uso del modelo.

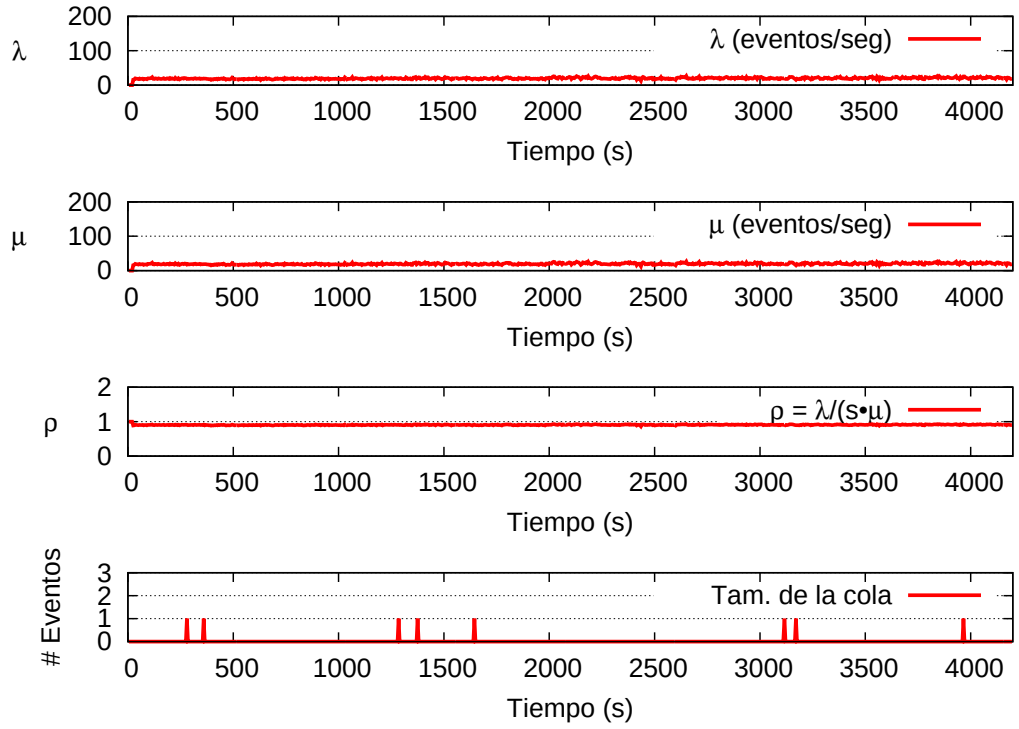

Figura E.16: Estadísticas del PE Mongo en la primera aplicación con un envío variable de la fuente de datos sin uso del modelo.

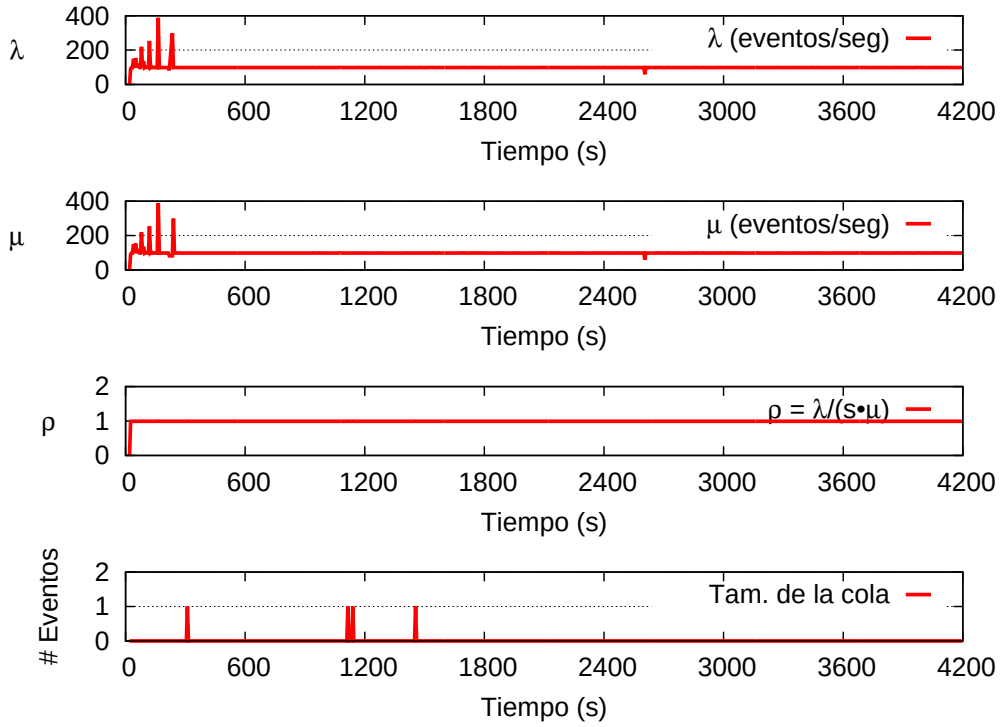

Figura E.17: Estadísticas del PE Split en la segunda aplicación con un envío constante de la fuente de datos con uso del modelo.

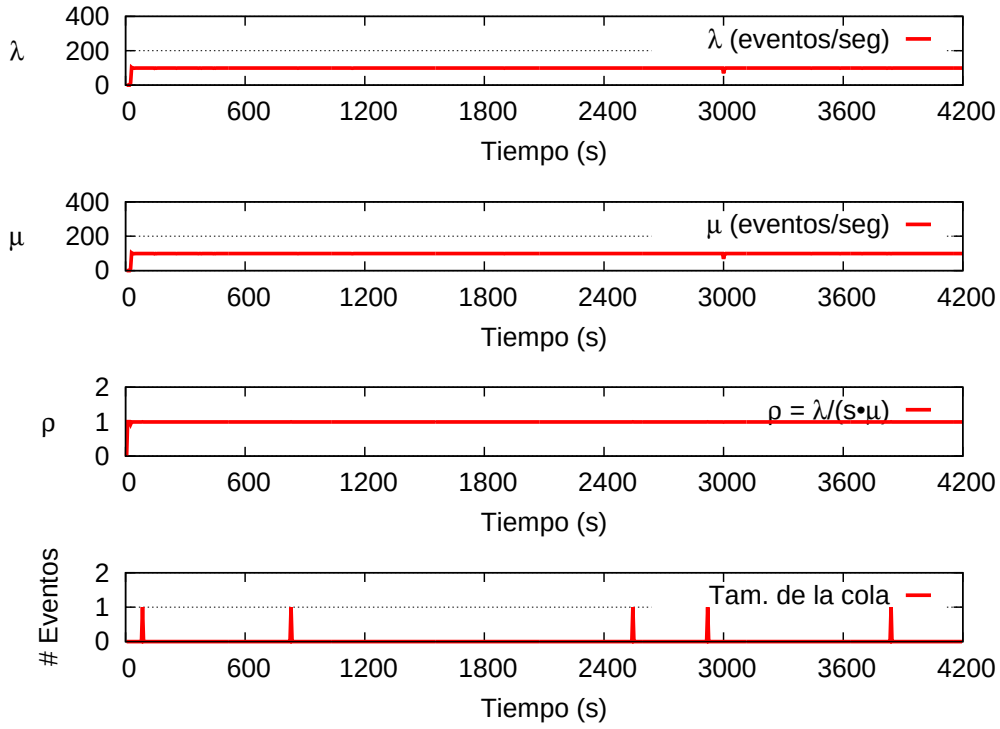

Figura E.18: Estadísticas del PE Split en la segunda aplicación con un envío constante de la fuente de datos sin uso del modelo.

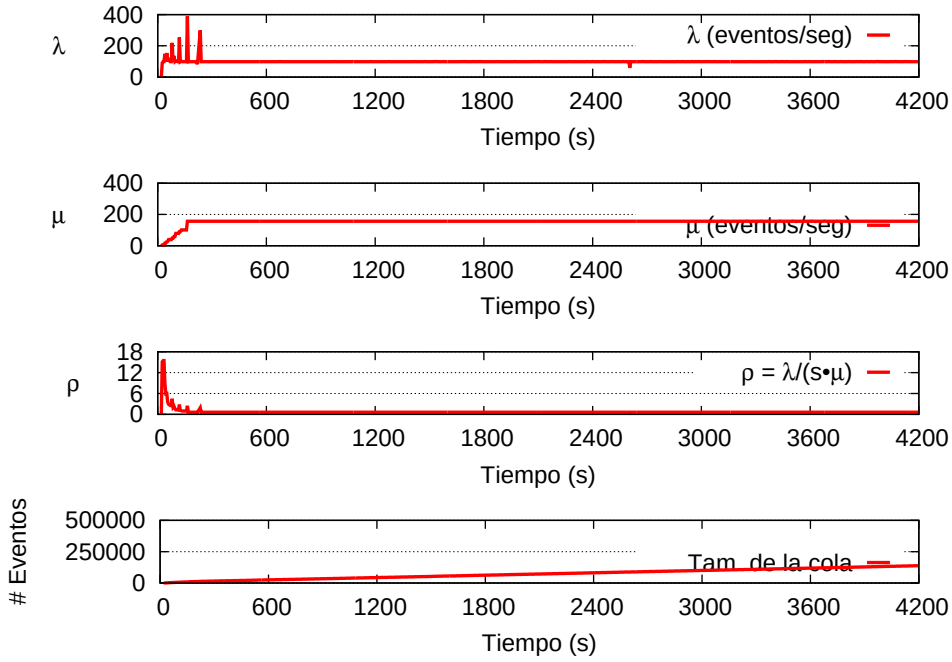

Figura E.19: Estadísticas del PE Counter en la segunda aplicación con un envío constante de la fuente de datos con uso del modelo.

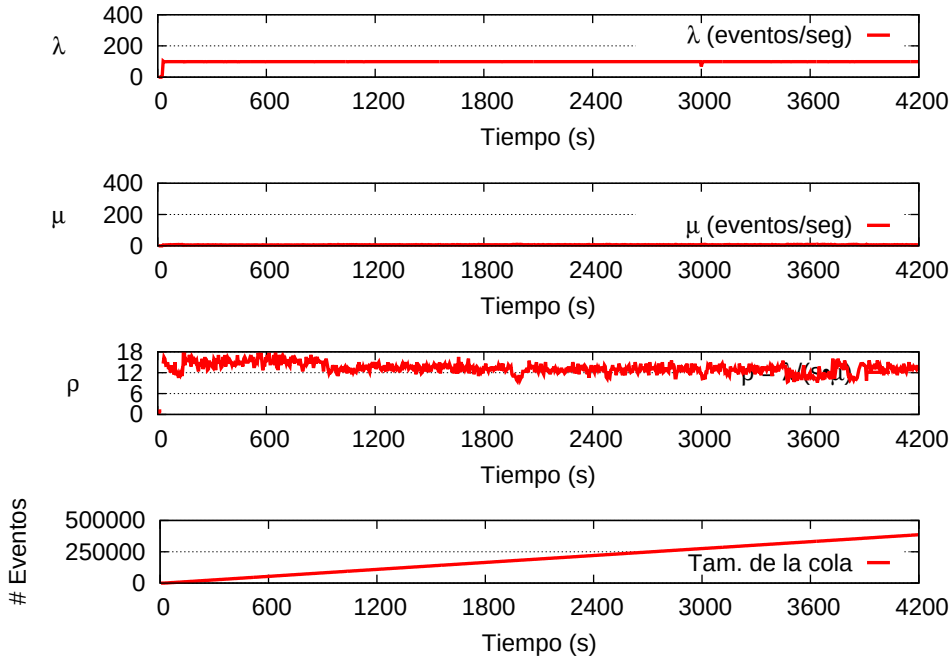

Figura E.20: Estadísticas del PE Counter en la segunda aplicación con un envío constante de la fuente de datos sin uso del modelo.

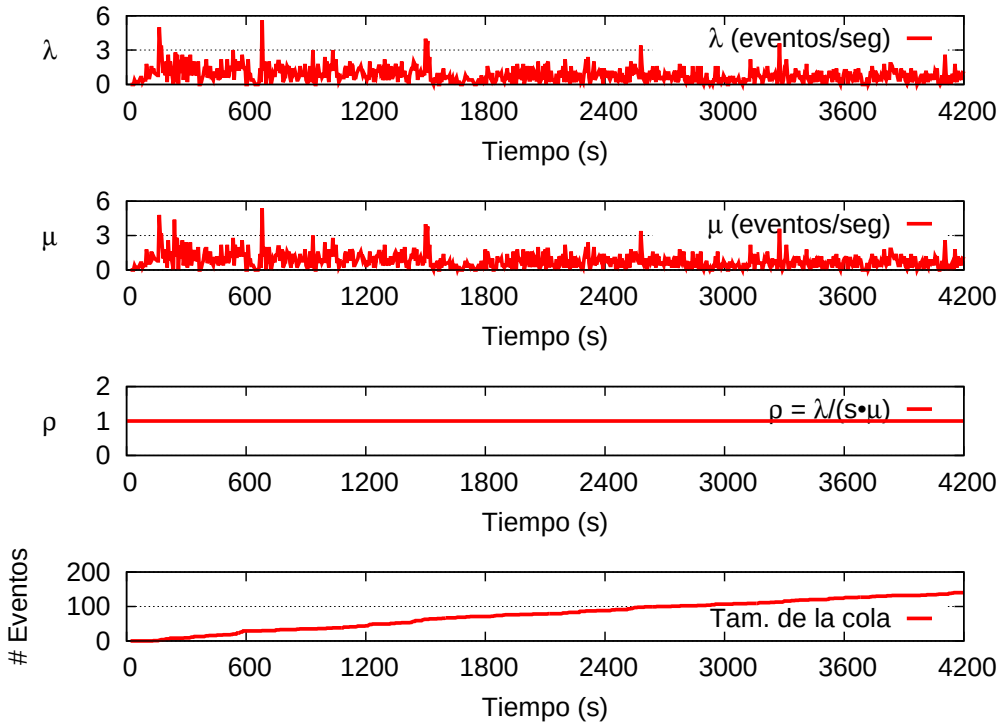

Figura E.21: Estadísticas del PE Merge en la segunda aplicación con un envío constante de la fuente de datos con uso del modelo.

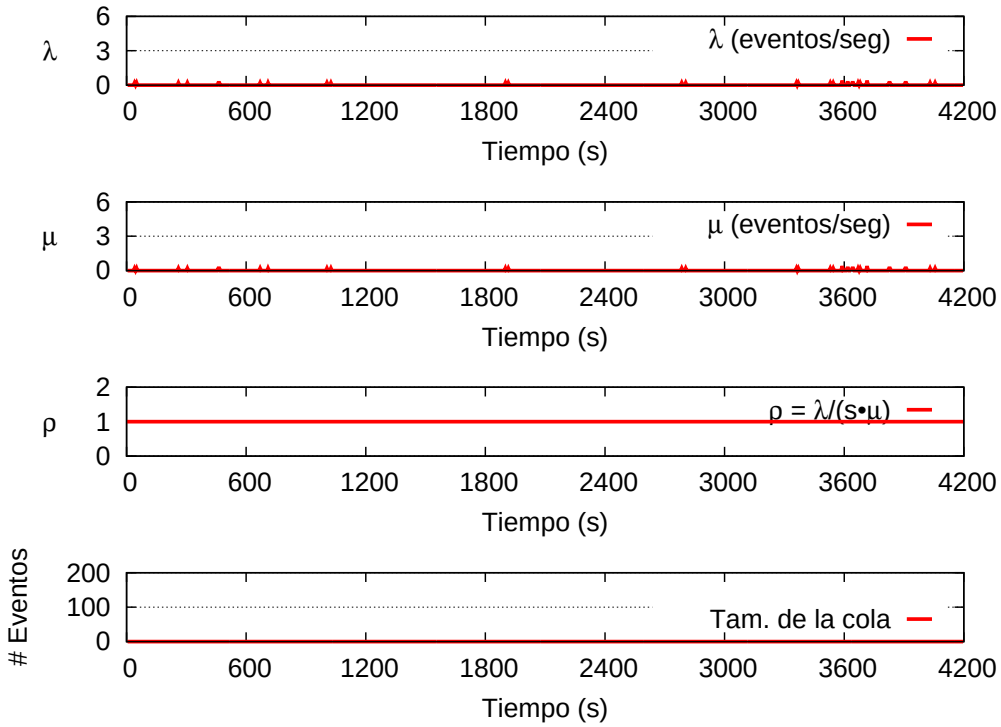

Figura E.22: Estadísticas del PE Merge en la segunda aplicación con un envío constante de la fuente de datos sin uso del modelo.

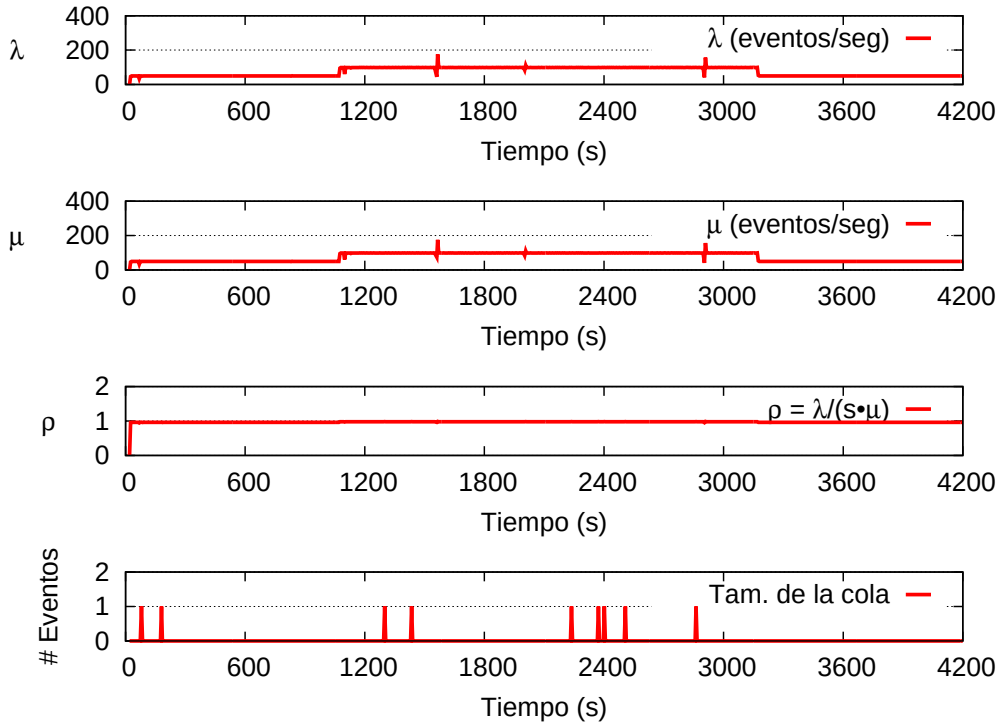

Figura E.23: Estadísticas del PE Split en la segunda aplicación con un envío variable de la fuente de datos con uso del modelo.

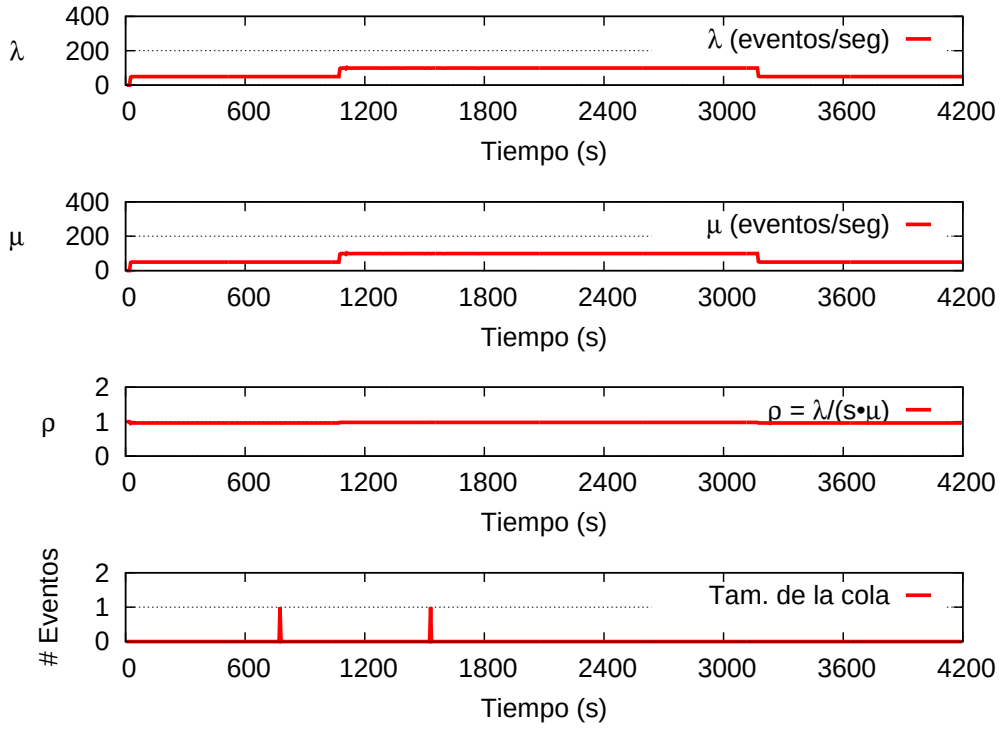

Figura E.24: Estadísticas del PE Split en la segunda aplicación con un envío variable de la fuente de datos sin uso del modelo.

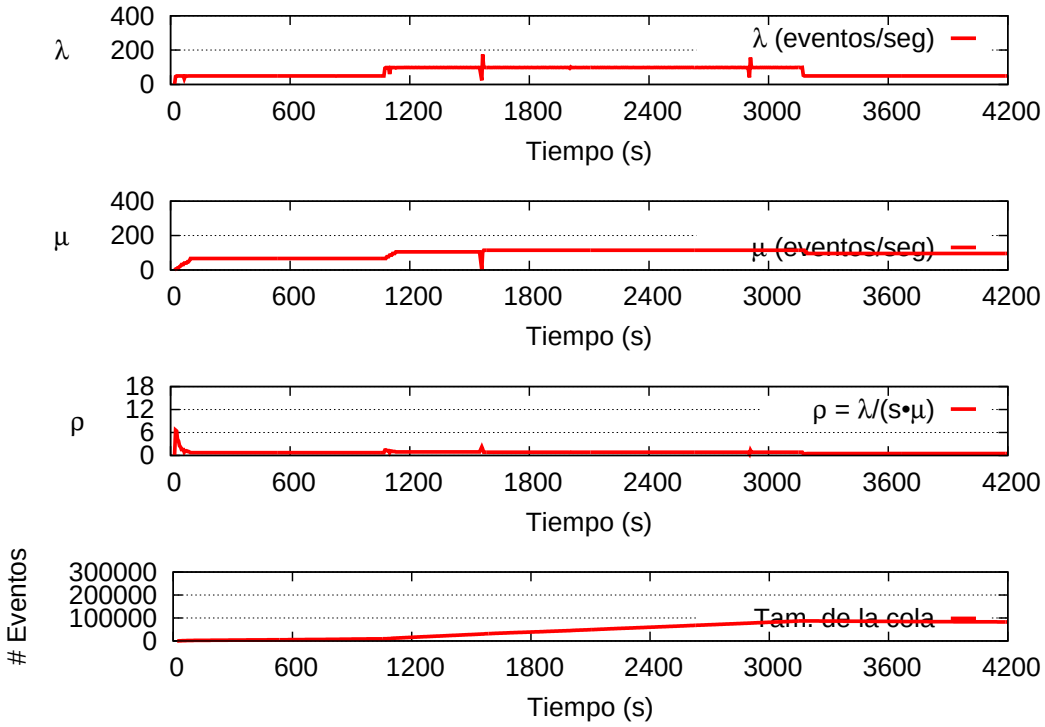

Figura E.25: Estadísticas del PE Counter en la segunda aplicación con un envío variable de la fuente de datos con uso del modelo.

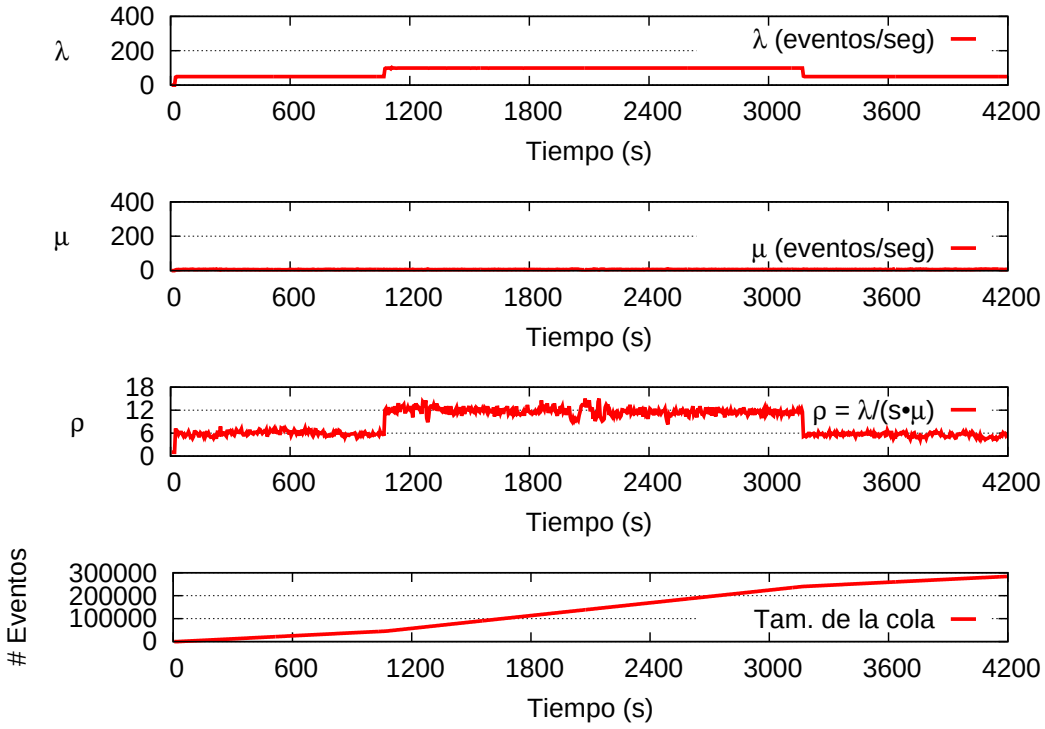

Figura E.26: Estadísticas del PE Counter en la segunda aplicación con un envío variable de la fuente de datos sin uso del modelo.

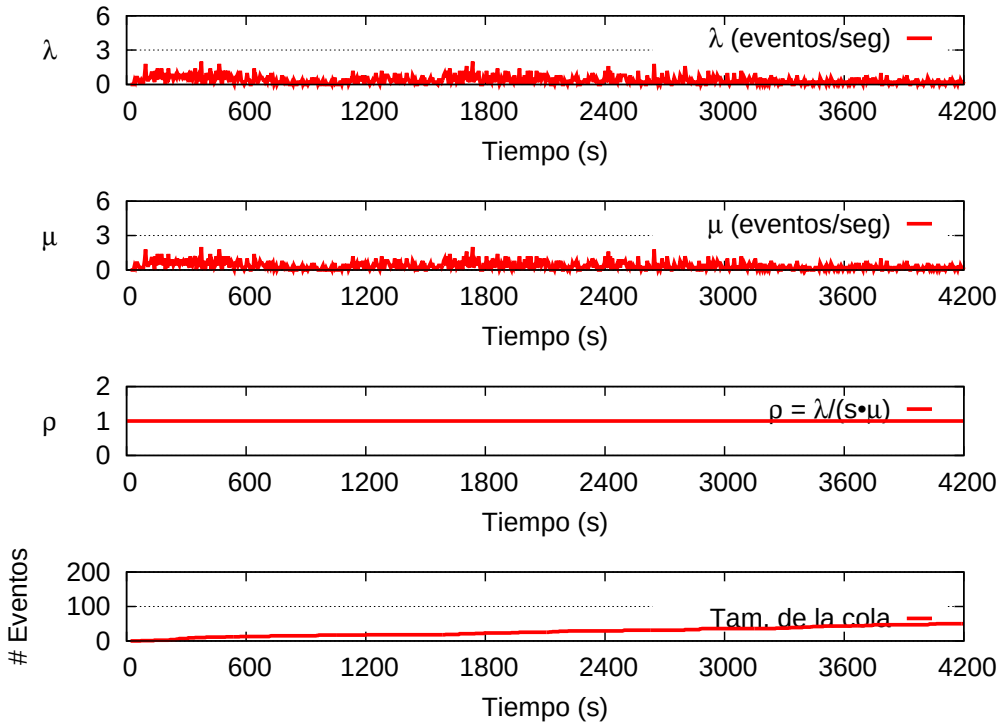

Figura E.27: Estadísticas del PE Merge en la segunda aplicación con un envío variable de la fuente de datos con uso del modelo.

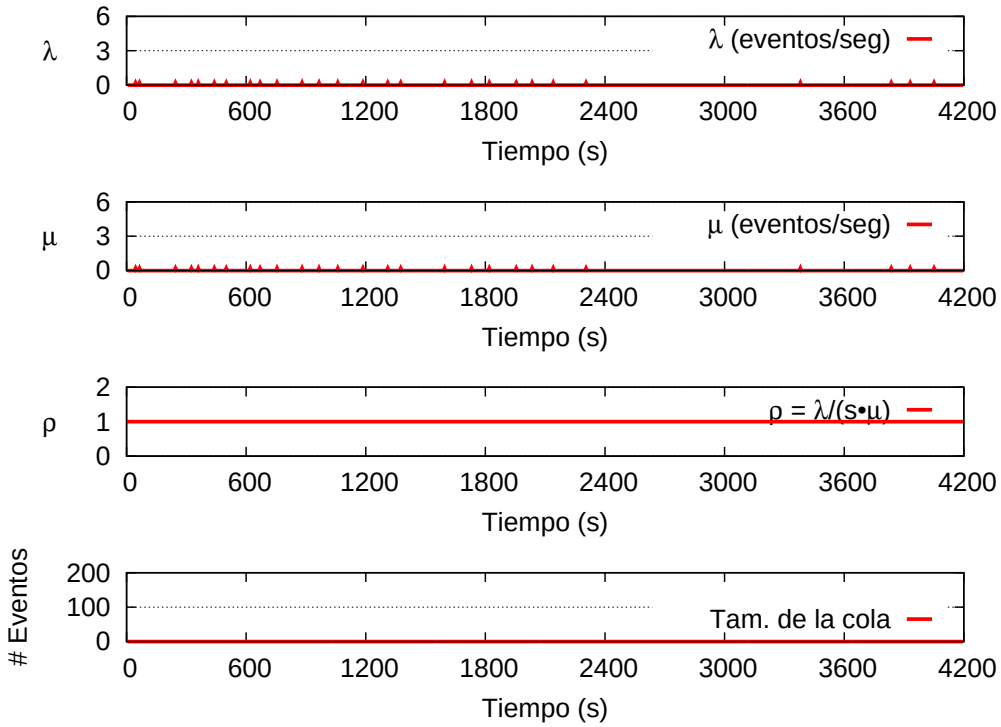

Figura E.28: Estadísticas del PE Merge en la segunda aplicación con un envío variable de la fuente de datos sin uso del modelo.

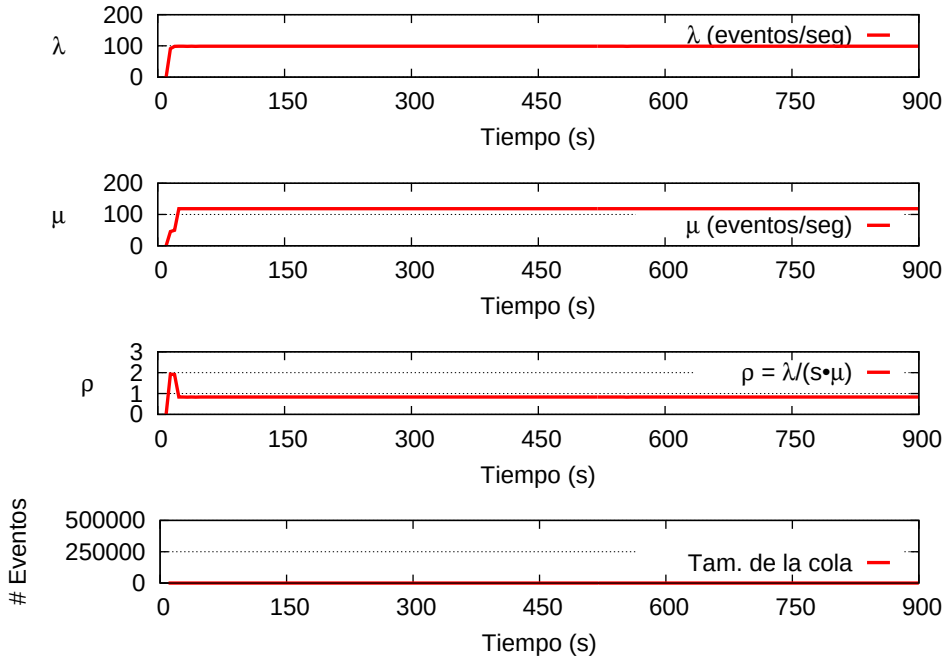

Figura E.29: Estadísticas del primer PE en la tercera aplicación con un envío constante de la fuente de datos con uso del modelo.

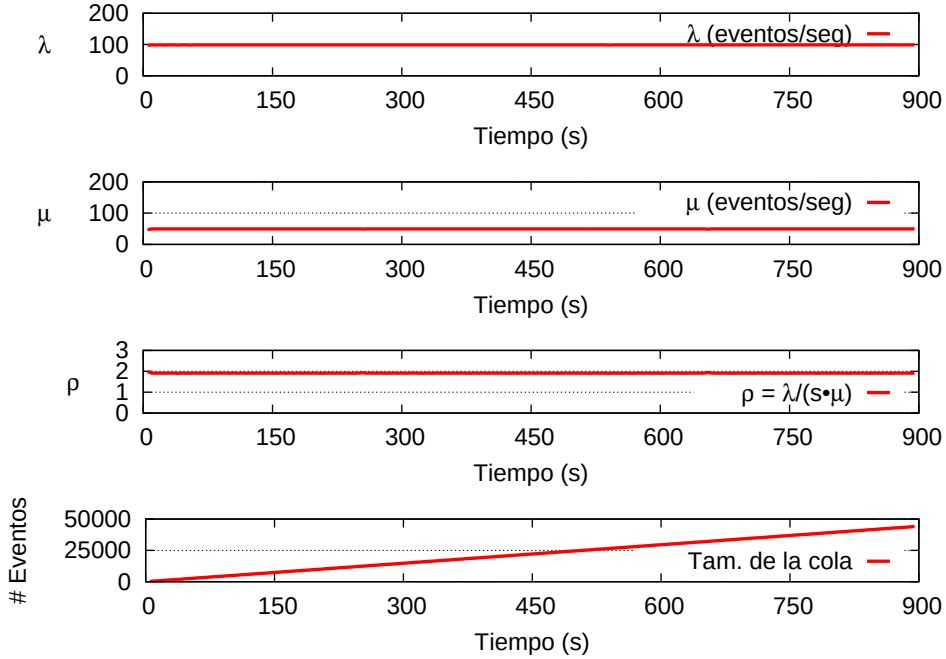

Figura E.30: Estadísticas del primer PE en la tercera aplicación con un envío constante de la fuente de datos sin uso del modelo.

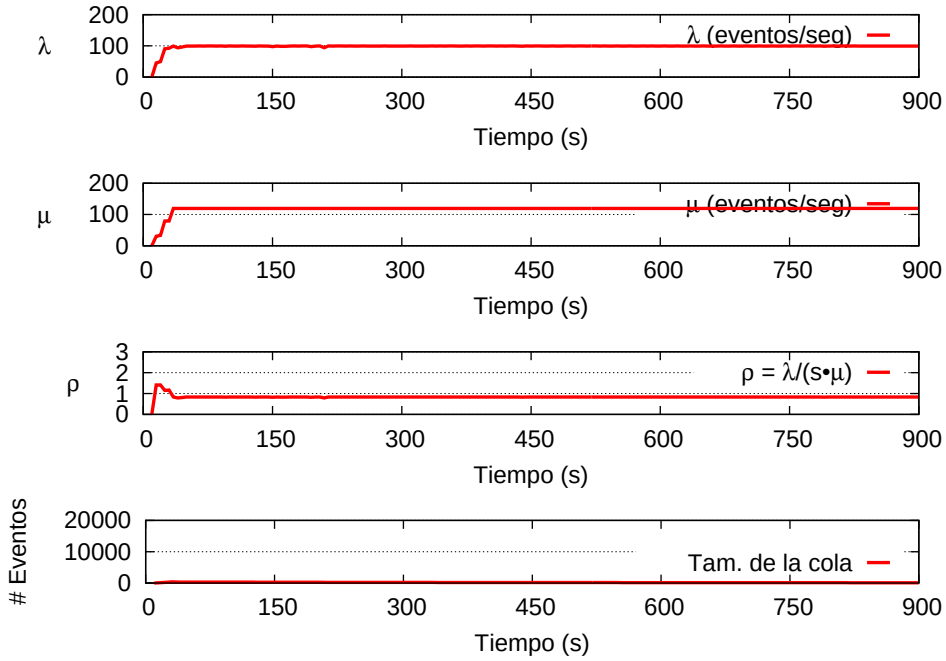

Figura E.31: Estadísticas del segundo PE en la tercera aplicación con un envío constante de la fuente de datos con uso del modelo.

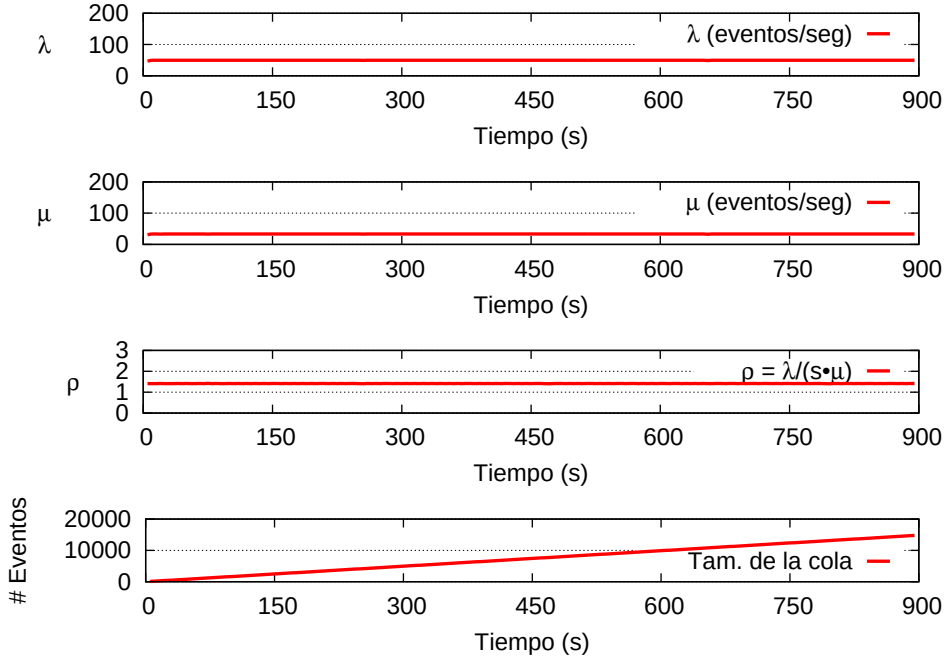

Figura E.32: Estadísticas del segundo PE en la tercera aplicación con un envío constante de la fuente de datos sin uso del modelo.

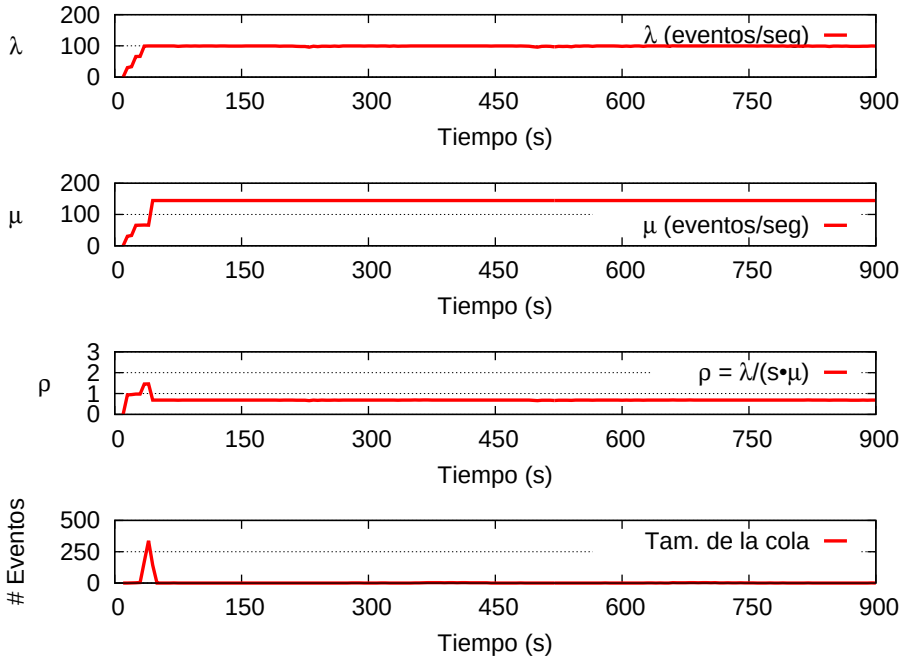

Figura E.33: Estadísticas del tercer PE en la tercera aplicación con un envío constante de la fuente de datos con uso del modelo.

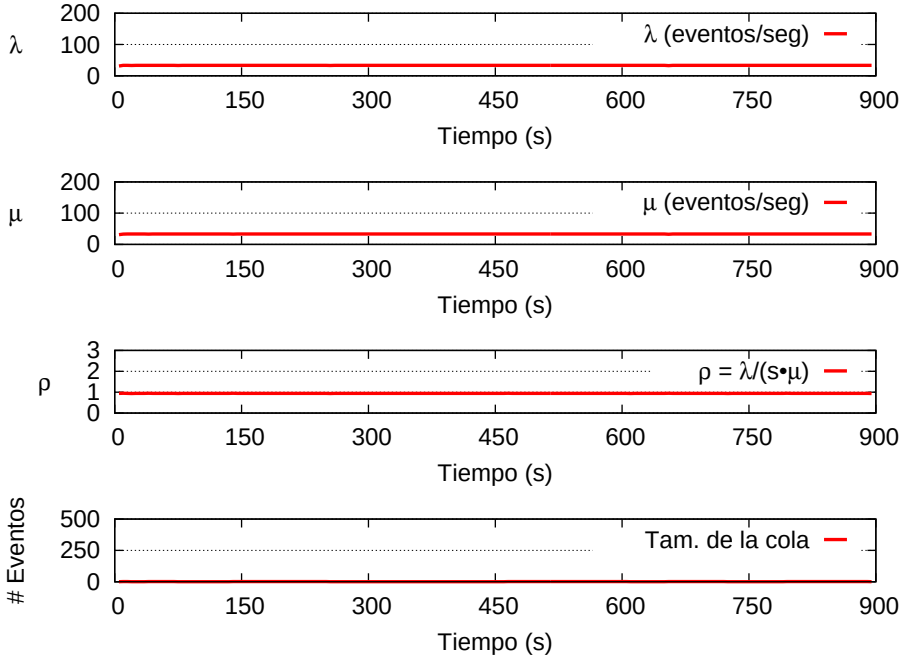

Figura E.34: Estadísticas del tercer PE en la tercera aplicación con un envío constante de la fuente de datos sin uso del modelo.