Universidad Nacional del Litoral

Proyecto Final de Carrera

Diseño de un sistema de detección de anomalías en redes de computadoras.

Informe de avance 3

Pineda Leandro
Córdoba
11 de marzo de 2018

1. Introducción

En este informe se mostrarán las mejoras realizadas al segundo incremento del proyecto titulado *Diseño de un sistema de detección de anomalías en redes de computadoras*. En primer lugar se discutirán las diferentes formas de comunicación entre los componentes de un sistema y las ventajas de la utilización de comunicación indirecta en el mundo de los microservicios. Se evaluarán diferentes alternativas en lo que respecta a las tecnologías más usadas en la actualidad y se comparará el desempeño del sistema provisto en el segundo incremento con la del tercer incremento. Además, se describirán los cambios realizados en la arquitectura y su justificación, así como la información que el sistema es capaz de proveer. Finalmente, se mostrará una primera versión de una interfaz web de usuario.

2. Comunicación entre componentes

La arquitectura de un sistema es la descripción de su estructura en términos de componentes específicos y sus interrelaciones. Mediante esta modularización, podemos asegurarnos que dicha estructura satisface las demandas actuales (es decir, resuelve el problema para el cual fue construido) y puede ser adaptada para satisfacer demandas futuras. Podemos definir a los sistemas distribuidos como aquellos compuestos por varios componentes que no comparten el mismo espacio de memoria[1]. Cuando se diseñan sistemas distribuidos es conveniente considerar dos cuestiones:

- ¿Cuales son las entidades que se comunican entre si?
- ¿Cómo van a comunicarse, o para ser mas específicos, que paradigma de comunicación va a usarse?

Estas preguntas son centrales para entender los sistemas distribuidos; qué se está comunicando y cómo esas entidades se comunican entre si, definen una gran cantidad de variables a ser consideradas a la hora de construir estos sistemas.

Las entidades que se comunican en un sistema distribuido son típicamente procesos, lo que nos permite entender a los sistemas distribuidos como procesos que se relacionan mediante los paradigmas de comunicación *entre procesos* apropiados. Podemos nombrar tres paradigmas de comunicación:

- Comunicación *entre procesos*.
- Invocación remota.
- Comunicación indirecta.

Comunicación entre procesos La comunicación *entre procesos* refiere al soporte de bajo nivel para la comunicación entre procesos en sistemas distribuidos, incluyendo primitivas para manejo de mensajes, acceso directo a las API provistas por protocolos de Internet (esto es, usando Sockets) y soporte para comunicación *multicast*.

Para comunicarse, un proceso envía un mensaje (una secuencia de bytes) a un receptor y un procesos ejecutandose allí recibe el mensaje. Esta actividad involucra el pasaje de datos de un proceso emisor a un proceso receptor y puede significar la sincronización de ambos procesos, generando una dependencia muy marcada entre los mismos.

Invocación remota La invocación remota representa el paradigma de comunicación más común en sistemas distribuidos. El intercambio de mensajes entre las entidades comunicantes es bidireccional, de forma que operaciones remotas, procedimientos y métodos, pueden ser invocados como se define a continuación:

- Protocolos *request-reply*: estos protocolos involucran el intercambio de mensajes desde el cliente al servidor y luego del servidor al cliente, donde el primer mensaje representa la operación que será ejecutada en el servidor (con los parámetros necesarios) y el segundo contiene cualquier resultado de dicha operación. Este paradigma es mas bien primitivo, y es utilizado generalmente en sistemas embebidos donde la *performance* es de suma importancia.
- *Remote procedure calls* (RPC) o llamadas a procedimientos remotos: este concepto, atribuido inicialmente a Birrel and Nelson [1984], representó un gran cambio en los paradigmas de computación distribuida. En RPC, los procedimientos de los procesos ejecutandose en computadoras remotas pueden ser invocados como si se encontraran el espacio local de memoria. De esta manera, el sistema abstrae aspectos acerca de la distribución, como la codificación de los parámetros, resultados y mecanismo de pasaje de mensajes. Este esquema soporta comunicación cliente-servidor pero depende de servidores que ofrezcan un conjunto de operaciones a través de una interfaz de servicio para que los clientes puedan llamar esas operaciones como si estuviesen disponibles localmente.

- *Remote method invocation* (RMI) o invocación remota de métodos: RMI es similar a RPC pero utiliza objetos distribuidos. Bajo este paradigma, un objeto cliente puede invocar métodos de un objeto remoto. De la misma forma que con RPC, ciertos detalles de como se implementa la comunicación quedan ocultos al usuario. Algunas implementaciones de RMI pueden incluir, además, soporte para darle a los objetos identidad y la habilidad de usar esos identificadores de objetos en llamadas remotas.

Comunicación indirecta Las técnicas discutidas hasta aquí tienen una cosa en común: la comunicación representa una relación en ambos sentidos entre el emisor y receptor, con los emisores enviando explícitamente mensajes/invocaciones a los receptores asociados. Los receptores generalmente deben saber sobre la identidad de los emisores y, en la mayoría de los casos, ambas partes deben existir al mismo tiempo para que la comunicación sea exitosa. Lo descrito anteriormente no puede garantizarse en ciertos escenarios. Por esto, surgieron numerosas técnicas donde la comunicación es indirecta a través de una tercera entidad, permitiendo un gran grado de desacople entre emisores y receptores. En particular:

- Los emisores no necesitan saber a quien le están enviando datos.
- Emisores y receptores no necesitan existir al mismo tiempo.

Las técnicas más usadas para comunicación indirecta incluyen:

- Sistemas *publish-suscribe*: en estos sistemas, un gran número de productores (o *publishers*) distribuyen eventos (elementos de información de interés) a un número similar de consumidores (o *suscribers*). Usar cualquier de los paradigmas discutidos anteriormente hubiera sido complejo e ineficiente y por lo tanto los sistemas *publish-suscribe* (a veces llamados sistemas basados en eventos) surgieron para cubrir esta demanda[1]. Todos los sistemas *publish-suscribe* comparten la característica crucial de proveer un servicio intermedio que asegura que la información generada por los productores es enrutada eficientemente a los consumidores que deseen dicha información.
- Colas de mensajes: de la misma forma que los sistemas *publish-suscribe* proveen un estilo de comunicación uno a muchos, las colas de mensajes ofrecen un servicio punto a punto mediante el cual los procesos de los productores pueden enviar mensajes a una cola especifica y los procesos consumidores pueden recibir los mensajes o ser notificados de la llegada de nuevos mensajes a la cola. Las colas, entonces, ofrecen una indirección entre los procesos productores y consumidores.

En la actualidad, existen numerosas tecnologías que permiten implementar comunicación indirecta en arquitecturas distribuidas. Se consideraron las siguientes alternativas:

- RabbitMQ: es un *message broker* de proposito general, sólido y maduro, que soporta la mayoría de los protocolos estándares como AMQP. Su diseño de centra en entrega consistente de mensaje a consumidores que leen mensajes a una velocidad similar a la que el *message broker* monitorea su estado (acusa recibo).
- Kafka: está diseñado para procesar grandes volumenes de mensajes tipo *publish-suscribe* y *streams*. Pensado para ser durable, rápido y escalable. Provee persistencia de mensajes y se ejecuta en cluster de servidores que almacenan flujos de eventos en categorías llamadas *tópicos*.
- ActiveMQ: es similar a Kafka pero a diferencia de este, los *message brokers* son los responsables de mantener el mensaje hasta que los consumidores lo hayan procesado. Esto hace que los *brokers* sean un poco más complejos y degrada la perfomance a medida que los el número de consumidores aumenta.
- Mosquitto: es una implementación de alta perfomance para el protocolo MQTT. Si bien está pensado para aplicaciones IoT, su thoughput para mensajes de poco tamaño es muy buena.

Actualmente existen 3 protocolos ampliamente aceptados (AMQP, MQTT y STOMP), que son soportados por las alternativas descritas anteriormente. Dado que solo se quiere enviar mensajes pequeños y se necesita evitar el agregado de *overhead* a los mismos para no degradar la *performance*, es necesario utilizar un protocolo liviano como MQTT. Si bien este protocolo es soportado por varias tecnologías, Mosquitto es el único diseñado exclusivamente como *message broker* MQTT. Además, las otras soluciones brindan una serie de características como la persistencia y reproducción de eventos, que no son de utilidad para esta implementación: esto agregaría innecesariamente complejidad y demanda de recursos de procesamiento, impactando de forma negativa a las prestaciones del servicio.

Dada la introducción sobre los aspectos de la comunicación de los diferentes componentes en sistemas distribuidos, estamos en condiciones de analizar los cambios realizados en la arquitectura del sistema con el objetivo de incrementar sus prestaciones en cuanto a *performance*.

2.1. Implementación previa del sistema: REST API

En la implementación inicial se había optado por registrar la ocurrencia de eventos usando una interfaz REST API, mayormente por la sencillez de su implementación y el amplio soporte y adopción que tiene esta tecnología en el mundo de los *webservices*. Sin embargo, se observa que cuando se intentan procesar cantidades masivas de eventos en tiempo real, el sistema evidencia una *performance* pobre dada la naturaleza síncrona de la comunicación.

Los resultados de las pruebas que serán mostradas a continuación, se realizaron en un sistema con las siguientes característica: procesador AMD FX6300 a 2,5 GHz, 8GB de memoria RAM, sistema operativo *Ubuntu Linux 16.04*, kernel *GNU/Linux 4.13.0.32-generic*.

Para determinar el volumen real de eventos que el sistema puede manejar, se generaron datos sintéticos y se registraron en el sistema de manera consecutiva por un lapso de 10 minutos, sin espera entre la registración de cada evento. El gráfico 1 muestra la cantidad de eventos procesados a lo largo del tiempo.

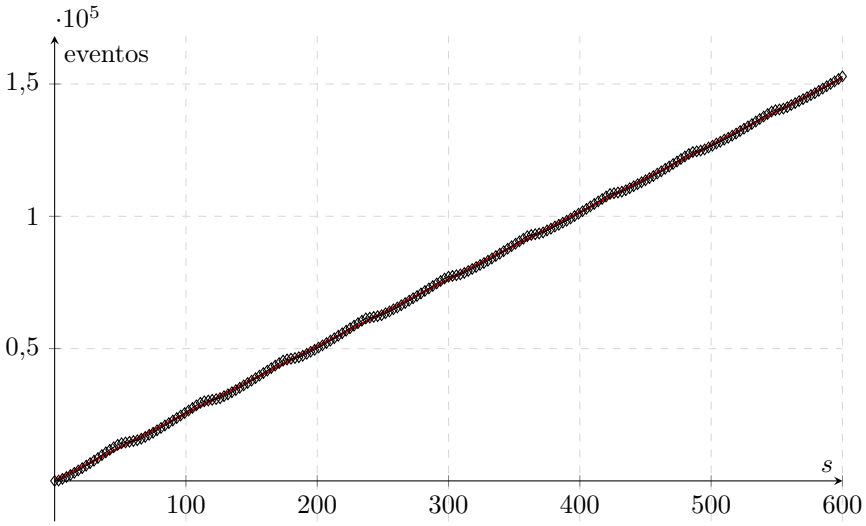

Figura 1: Eventos procesados a lo largo del experimento (REST API).

La recta que mejor ajusta los datos está dada por la ecuación $y = 253{,}71x + 32{,}1$. La pendiente de la recta sirve entonces para estimar la performance de la implementación REST del sistema en alrededor de 250 eventos por segundo.

2.1.1. Nueva implementación: Messaging Queue

Para esta nueva implementación es necesario un nuevo componente en la arquitectura del sistema. Se agregó una instancia de una *message queue* y se hicieron las modificaciones necesarias para que el *webservice* se suscriba a un tópico determinado y procese los eventos que son allí

publicados: esto implica cambiar la interfaz para la creación de nuevos eventos e implementar los callbacks necesarios.

Las pruebas realizadas son similares a las presentadas anteriormente y se realizan en el mismo sistema. Se generaron eventos aleatorios y se publicaron en un canal determinado de la cola de mensajes. Muestreando la cantidad de eventos que el sistema procesa a intervalos regulares de tiempo, se confecciono la gráfica 2. A simple vista se puede observar un aumento pronunciado en la performance del sistema. La recta que ajusta a los datos está dada por la ecuación $y = 6907{,}2x - 48449$, por lo que podemos estimar un volumen de procesamiento de alrededor de 6900 eventos por segundo. La única desventaja de este método de envío de mensajes es que requiere que los clientes publiquen los mensajes en un canal dado, haciendo uso de las librerías correspondientes para su implementación. Además, podemos observar que la gráfica es una curva perfecta, mientras que en el caso previo se pueden observar pequeñas oscilaciones regulares a lo largo del tiempo. Esto puede deberse a procesos que JVM ejecuta en segundo plano, como el *garbage collector*, dado el volumen de objetos que se crean y destruyen en memoria.

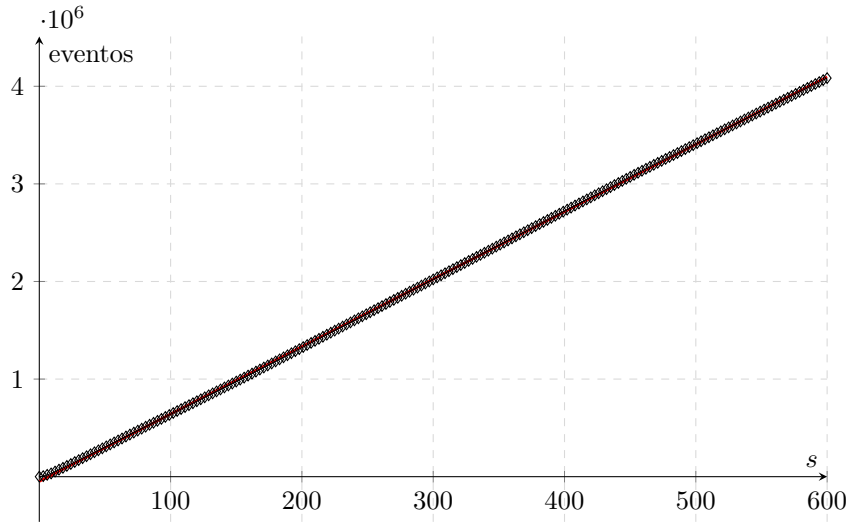

Figura 2: Eventos procesados a lo largo del experimento (Message Queue).

Con la implementación actual, podemos observar que la performance del sistema es 27 veces mejor que la implementación del segundo incremento. Aunque esto implica el despliegue de un servicio de cola de mensajes y escribir clientes que publiquen mensajes en la misma, la ganancia en prestaciones es lo suficientemente buena como para que el beneficio justifique los costos de desarrollo.

3. Cambios en la arquitectura

Los cambios en el código del micro servicios que fueron realizados en este incremento pueden agruparse en 3 categorías, según el objetivo que cumplen:

- Incrementar la performance del sistema.
- Mejorar la legibilidad del código (refactoring).
- Exponer la información necesaria para implementar una interfaz web.

3.1. Incremento en la performance del sistema

Para esta mejora fue necesario integrar una *message queue* a la arquitectura del sistema. Las modificaciones en el micro servicio no implicaron cambios estructurales muy grandes al código del segundo incremento dado que la interfaz para registrar eventos estaba bien definida. Sin embargo, fue necesario agregar un nuevo servicio, y la lógica para integrar el mismo.

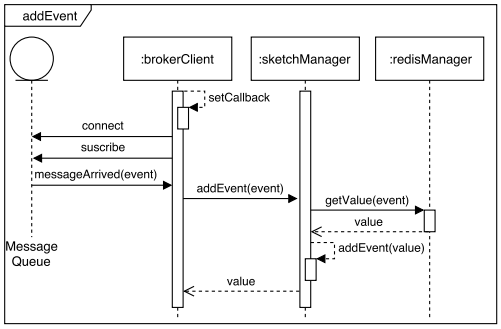

Figura 3: Diagrama de secuencia para nuevos eventos (tercer incremento)

En la implementación anterior, los mensajes eran registrados directamente por un actor externo, mientras que ahora el sistema se suscribe a un canal particular de la cola de mensajes, escuchando nuevos eventos publicados en el mismo(es decir, usando comunicación indirecta). Mediante una función de *callback* se especifica qué acción debe realizarse cada vez que el tópico notifica la llegada de un nuevo mensaje: este es agregado a un *sketch* mediante *sketchManager* y procesado de igual forma que en la implementación del segundo incremento. La figura 3 muestra cómo el objeto *brokerClient* implementa la lógica descrita anteriormente. Cuando es instanciado, el cliente debe conectarse a la cola de mensajes, suscribirse a un tópico y configurar el *callback*. También encapsula la lógica necesaria para manejar la pérdida o interrupción de la conexión al servicio.

3.2. Reestructuración de código

Otro aspecto importante a mejorar es la calidad del código del sistema. Esto no solo ayuda a mejorar ciertos aspectos funcionales como la eficiencia, sino que permite entender de manera sencilla los distintos módulos del sistema y la responsabilidad de cada uno de ellos. Así, el código es mucho más simple de interpretar, mantener y modificar. El diagrama 4 muestra los diferentes componentes del sistema y la relación que existe entre ellos.

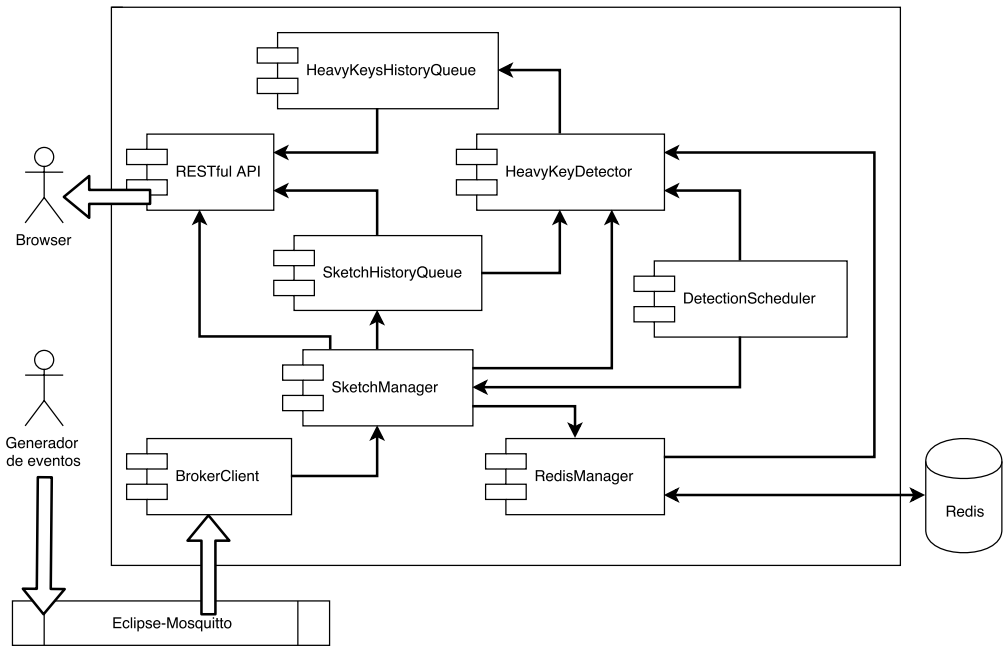

Figura 4: Componentes del sistema y su interacción

Podemos describir la responsabilidad de cada módulo para comprender cómo funciona el servicio. El componente *RedisManager* encapsula los métodos para conectar, consulta y almace-

nar datos en la base de datos clave-valor. Expone dos métodos para obtener los valores enteros asociados a un evento, y los eventos asociados a un conjunto de valores enteros. *BrokerClient* envía los eventos que son publicados en un tópico a *SketchManager*. Este último mantiene un objeto *Sketch* activo en todo momento y actualiza su estado con cada nuevo evento procesado. Además, mediante *RedisManager*, asigna un valor entero a cada evento registrado y persiste el par evento-valor en la base de datos.

Una vez transcurrida una ventana de tiempo o época, deben ejecutarse los métodos necesarios para detectar los eventos anómalos que están siendo procesados por el sistema. El componente *DetectionScheduler* es el encargado de orquestar los diferentes pasos del algoritmo de detección implementado. Primero, indica al componente *SketchManager* que una época a finalizado: este genera un nuevo objeto *sketch* y almacena el objeto *sketch* anteriormente activo en la cola *SketchHistoryQueue*. Al mismo tiempo, el componente *HeavyKeyDetector* es notificado del fin de la época: este toma los dos últimos *sketch* disponibles en *SketchHistoryQueue* y ejecuta los algoritmos de detección de eventos anómalos. Como la representación de los eventos en esta instancia son números enteros, deben recuperarse de la base de datos los eventos asociados a estos mediante *RedisManager*. Finalmente, asocia los eventos detectados con la época de detección correspondiente, que es obtenida mediante *SketchManager*. Al finalizar la detección, *HeavyKeyDetector* guarda los eventos detectados en la cola *HeavyKeysHistoryQueue* para ser consultados convenientemente. Finalmente, se expone mediante una API REST los resultados de la detección junto con información del estado general del sistema.

3.3. Exposición de los resultados de la detección

A medida que los eventos son procesados el algoritmo de detección de eventos anómalos genera resultados en intervalos regulares de tiempo. Estos deben ser expuestos de manera conveniente para generar un visualización apropiada mediante una interfaz web. De esta forma, el sistema o *backend* queda totalmente desacoplado de la visualización de la información o *frontend*. Esto permite cambiar y mejorar los métodos de detección sin afectar su visualización, permitiendo de la misma manera cambiar la forma en que los datos son presentados sin necesidad de modificar el *backend* en absoluto. La información provista usando una REST API se enumera a continuación:

- Estado general del sistema: valores de configuración, época actual de detección (número entero) y cantidad total de eventos procesados por el sistema.
- Información de cada época: cantidad de elementos procesados y fecha de comienzo.
- Eventos detectados: fecha de finalización de cada época, cantidad de eventos anómalos detectados por época y los eventos detectados. Estos eventos son una lista de cadenas de caracteres o *strings* para cáda época de detección.

Así, la interfaz web que será descrita en la siguiente sección, solo debe consumir estos datos y presentarlos de manera conveniente. Si lo pensamos desde el punto de vista de un modelo MVC (Model-View-Controller), el *backend* resuelve el Model y el Controller, y el *frontend* web resuelve la vista.

4. Interfaz web

Con las API correspondiente ofreciendo los resultados de la detección, solo resta implementar una vista de estos datos. Como la visualización de los datos pretende ser simple (emulando un *dashboard*), se optó por implementar una aplicación web de página única, o SPA por *Single-page Applicacion*. La ventaja principal de las SPA es que son dinámicas, lo cual evita refrescar la página para actualizar los datos mostrados, dando así una experiencia de aplicación de escritorio pero permitiendo ser usadas mediante un navegador, sin descargar programas adicionales.

Respecto a las tecnologías disponibles, existen dos estándares:

- AngularJS: es un framework desarrollado y mantenido por Google Inc. Está escrito en JavaScript y hace uso del patrón de diseño MVC.

- ReactJS: es una librería para construir interfaces de usuario. Desarrollada y mantenida por Facebook, está pensada para diseñar vistas simples para cada estado de una aplicación. La librería es la encargada de actualizar y re-dibujar los componentes que sean necesarios cuando los datos cambian.

Si bien ambas tecnologías son ampliamente usadas y su desarrollo es sumamente activo, ReactJS parece ser la opción más conveniente dado que sus características se ajustan al problema que se intenta resolver, es decir, mostrar los datos provistos por el *web-service* de manera dinámica, mientras que AngularJS ofrece un framework MVC completo que agregaría complejidad al desarollo.

En cuanto a cómo mostrar los datos, la primera aproximación consistió en el uso de una tabla con 5 columnas para fecha de detección, cantidad de *Heavy Hitters*, los *Heavy Hitters* detectados, cantidad de *Heavy Changers* y los *Heavy Changers* detectados. Sin embargo, cómo la cantidad de eventos detectados por fila es variable, hubiese sido necesario mostrar más de una dato por celda, por lo que la utilización de una tabla parecería no ser la adecuada (se asume que cada celda es atómica, es decir, debe mostrar un único dato).

Otra alternativa mas conveniente dada la naturaleza temporal de los datos, y la cantidad de ocurrencia de eventos, es utilizar un histograma con desplazamiento. Con cada desplazamiento, los datos para la nueva época son agregados. Así, puede observarse cuando ocurren ciertos eventos, y compararlos a simple vista con otros intervalos de tiempo pasados y futuros. En la imagen 5 puede observase una primera implementación de la interfaz web.

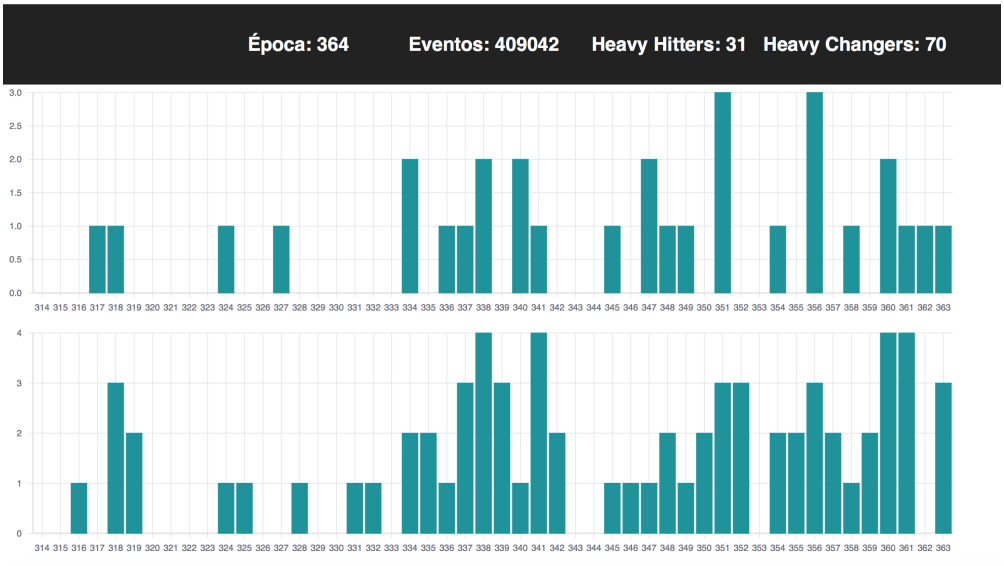

Figura 5: Interfaz web del sistema

En el panel superior pueden observarse los datos generales del sistema: época que está siendo computada, cantidad de eventos totales procesados y la cantidad de *heavy keys* encontradas hasta el momento. El primer histograma muestra cuantos *heavy hitters* ocurrieron en cada época y el segundo muestra cuando *heavy changers* fueron identificados. Queda pendiente elegir definir es la mejor forma de visualizar las etiquetas de los eventos detectados y mejorar la descripción de los datos en la interfaz.

5. Instalación

El sistema y sus dependencias se implementan usando *docker-compose*[1]. Esto permite su distribución de manera sencilla y asegura independencia del sistema operativo y servidor donde es ejecutado. Las instrucciones para su instalación, configuración y prueba pueden encontrarse

[1] https://docs.docker.com/compose/

en https://github.com/leandropineda/sketch-service. Las imagenes Docker utilizadas para esta entrega están etiquetadas, lo que permite la evolución del código sin afectar la entrega actual.

6. Conclusiones

En este incremento se realizaron una cantidad considerable de modificaciones al código. Como es usual en los desarrollos ágiles, primero se parte de una prueba de concepto para mostrar el valor que puede entregar un sistema o algoritmo, y luego se trabaja en mejorar aspectos técnicos para hacer de la misma una idea productizable. En particular, podemos observar que las mejoraras en *performance* son más que satisfactorias. Si bien para el presente informe sólo se usaron datos sintéticos y de tráfico de red, el mismo método puede aplicarse en diferentes aplicaciones: e-commerce (determinar en tiempo real artículo es el mas buscado) o análisis de texto en tiempo real (cuales son las palabras más usadas en todo momento), por nombrar algunos. Por otro lado, la integración de una *message queue* en el sistema fue fundamental, no solo para incrementar las prestaciones del servicio, sino para permitir escalabilidad: ante grandes volúmenes de eventos enviados al servicio, la carga es soportada por la *message queue*, lo cual evita agregar esta lógica al servicio desarrollado. Finalmente, la interfaz web muestra, en una sola vista, toda la actividad detectada por el sistema y permite a usuario observar cual es el estado general de los eventos que está analizando, sin necesidad de interactuar con la interfaz.

Referencias

[1] G. Mühl, L. Fiege y P. Pietzuch, *Distributed Event-Based Systems*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006, ISBN: 3540326510.

[2] G. Cormode y M. Hadjieleftheriou, «Finding Frequent Items in Data Streams», *Proc. VLDB Endow.*, vol. 1, n.º 2, págs. 1530-1541, ago. de 2008, ISSN: 2150-8097. DOI: 10.14778/1454159.1454225. dirección: http://dx.doi.org/10.14778/1454159.1454225.

[3] S. Muthukrishnan, «Data Streams: Algorithms and Applications», *Found. Trends Theor. Comput. Sci.*, vol. 1, n.º 2, págs. 117-236, ago. de 2005, ISSN: 1551-305X. DOI: 10.1561/0400000002. dirección: http://dx.doi.org/10.1561/0400000002.

[4] D. Terry, D. Goldberg, D. Nichols y B. Oki, «Continuous Queries over Append-only Databases», *SIGMOD Rec.*, vol. 21, n.º 2, págs. 321-330, jun. de 1992, ISSN: 0163-5808. DOI: 10.1145/141484.130333. dirección: http://doi.acm.org/10.1145/141484.130333.

[5] D. Tong y V. Prasanna, «High Throughput Sketch Based Online Heavy Hitter Detection on FPGA», *SIGARCH Comput. Archit. News*, vol. 43, n.º 4, págs. 70-75, abr. de 2016, ISSN: 0163-5964. DOI: 10.1145/2927964.2927977. dirección: http://doi.acm.org/10.1145/2927964.2927977.

[6] G. Cormode y M. Hadjieleftheriou, «Methods for Finding Frequent Items in Data Streams», *The VLDB Journal*, vol. 19, n.º 1, págs. 3-20, feb. de 2010, ISSN: 1066-8888. DOI: 10.1007/s00778-009-0172-z. dirección: http://dx.doi.org/10.1007/s00778-009-0172-z.

[7] L. J. Guibas, «Problems», *Journal of Algorithms*, vol. 2, n.º 2, págs. 208 -210, 1981, ISSN: 0196-6774. DOI: http://dx.doi.org/10.1016/0196-6774(81)90022-5. dirección: http://www.sciencedirect.com/science/article/pii/0196677481900225.

[8] R. M. Karp, S. Shenker y C. H. Papadimitriou, «A Simple Algorithm for Finding Frequent Elements in Streams and Bags», *ACM Trans. Database Syst.*, vol. 28, n.º 1, págs. 51-55, mar. de 2003, ISSN: 0362-5915. DOI: 10.1145/762471.762473. dirección: http://doi.acm.org/10.1145/762471.762473.

[9] E. Kranakis, P. Morin e Y. Tang, «Bounds for Frequency Estimation of Packet Streams», en *In SIROCCO*, 2003, págs. 33-42.

[10] E. M. Hutchins, M. J. Cloppert y R. M. Amin, «Intelligence-driven computer network defense informed by analysis of adversary campaigns and intrusion kill chains», *Leading Issues in Information Warfare & Security Research*, vol. 1, pág. 80, 2011.

[11] A. Metwally, D. Agrawal y A. El Abbadi, «Efficient Computation of Frequent and Top-k Elements in Data Streams», en *Proceedings of the 10th International Conference on Database Theory*, ép. ICDT'05, Edinburgh, UK: Springer-Verlag, 2005, págs. 398-412, ISBN: 3-540-24288-0, 978-3-540-24288-8. DOI: 10.1007/978-3-540-30570-5_27. dirección: http://dx.doi.org/10.1007/978-3-540-30570-5_27.

[12] G. Cormode y S. Muthukrishnan, «An Improved Data Stream Summary: The Count-min Sketch and Its Applications», *J. Algorithms*, vol. 55, n.º 1, págs. 58-75, abr. de 2005, ISSN: 0196-6774. DOI: 10.1016/j.jalgor.2003.12.001. dirección: http://dx.doi.org/10.1016/j.jalgor.2003.12.001.

[13] F. Putze, P. Sanders y J. Singler, «Cache-, Hash-, and Space-efficient Bloom Filters», *J. Exp. Algorithmics*, vol. 14, 4:4.4-4:4.18, ene. de 2010, ISSN: 1084-6654. DOI: 10.1145/1498698.1594230. dirección: http://doi.acm.org/10.1145/1498698.1594230.

[14] B. H. Bloom, «Space/Time Trade-offs in Hash Coding with Allowable Errors», *Commun. ACM*, vol. 13, n.º 7, págs. 422-426, jul. de 1970, ISSN: 0001-0782. DOI: 10.1145/362686.362692. dirección: http://doi.acm.org/10.1145/362686.362692.

[15] D. Crockford, *The application/json Media Type for JavaScript Object Notation (JSON)*, RFC 4627, jul. de 2006. DOI: 10.17487/RFC4627. dirección: https://rfc-editor.org/rfc/rfc4627.txt.