
Universidad Nacional del Litoral

Proyecto Final de Carrera

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

## Informe de avance 2

*Pineda Leandro*
Córdoba
3 de marzo de 2018

# 1. Introducción

Determinar anomalías en una serie temporal de eventos permite detectar fenómenos que pueden ser de interés para ciertas aplicaciones. Por ejemplo, monitorear la demanda de recursos en un servicio de red, o determinar que anuncio publicitario es la más visto en un sitio web.

Es interesante observar anomalías en el comportamiento de ciertos componentes en distintos puntos de una red de datos. Por ejemplo, pueden analizarse los flujos de datos en los puntos de acceso de la red. En la capa de transporte, el protocolo TCP (que provee conexión host a host sobre IP) provee información suficiente para identificar estos patrones. Además, los diferentes servicios que funcionan en una red, como servidores web, producen grandes cantidades de registros de auditoría (o logs) que pueden ser analizado en busca de patrones anómalos. Para llevar a cabo el análisis en tiempo real de eventos (tanto los segmentos TCP que son transportados como los logs de los distintos servicios o eventos generados por cierta aplicación), utilizaremos el modelo de *data streaming*: este se basa esencialmente en algoritmos que procesan una única vez los datos, dado que transferirlos y almacenarlos no es posible en la práctica, y permiten determinar cuales son los elementos más comunes en el stream de datos, así como algunos estadísticos (como la media, mediana, histograma), entre otros.

# 2. El modelo de data streaming

En este modelo, las entradas que van a ser procesadas no están disponibles para ser accedidas aleatoriamente desde disco o memoria, sino que llegan como uno o mas flujos continuos de datos. Los *streams* de datos son diferentes de los modelos relacionales convencionales[1] en varios aspectos:

- Los elementos del stream deben ser procesados de manera *online*.
- Los sistemas que procesan los datos no tienen control sobre el orden en los elementos de la entrada.
- Los stream de datos pueden ser infinitos.
- Una vez que un elemento es procesado, este se descarta.[2]

Con respecto a los datos podemos hacer una distinción entre *consultas únicas* y *consultas continuas*[1]. Las consultas únicas (como aquellas que se hacen mediante un DBMS tradicional) son evaluadas una vez sobre un *snapshot* de un conjunto de datos. La consultas continuas, por otro lado, son evaluadas continuamente mientras el *stream* de datos esta siendo procesado. Las respuestas a estas consultas pueden ser almacenadas y actualizadas en la medida que llegan nuevos flujos de datos, o pueden ser origen de otro *stream* de datos.

Para el análisis de tráfico de red, podemos pensar a los flujos TCP que pasan por un *gateway* como los eventos o *keys* a ser procesados. Las *keys* pueden ser agrupadas mediante la 5-tupla dirección de IP de origen, destino, número de puerto de origen y destino y protocolo. De esta manera se puede analizar el comportamiento del tráfico de red de cada una de las sesiones en busca de anomalías como ser sesiones que hacen uso intensivo de recursos de red en un momento del tiempo. Sin embargo, el espacio de las *keys* es tan grande que llevar registro de todos los eventos no es viable. Este tipo de problemas y similares llevaron al desarrollo de los llamados *modelos de streaming* y la utilización de diferentes técnicas de conteo: bajo esta abstracción, los algoritmos procesan la entrada una única vez y deben calcular de manera precisa varios resultados usando recursos (espacio y tiempo por elemento) de forma estrictamente sublineal al tamaño de la entrada[2]. Existen diferentes algoritmos para procesar y obtener información acerca de los eventos usando estructuras de datos que utilizan el espacio de memoria eficientemente. Sin embargo, estos métodos no calculan la frecuencia exacta de cada evento sino que la estiman: en general, para cantidades masivas de eventos basta con tener una buena aproximación de las frecuencias para identificar anomalías.

[1] Datos en almacenados usando un gestor de base de datos

[2] En algunas aplicaciones los elementos pueden ser almanacenados para ser procesados posteriormente, pero en un modelo de streaming "puro"los elementos son procesados una única vez.

El problema de los eventos frecuentes consiste en procesar una serie consecutiva de elementos y encontrar aquellos que ocurren más frecuentemente en un período de tiempo. Es un problema muy estudiado en la minería de *streams* de datos debido a que la resolución de muchos problemas se basan directa o indirectamente en la identificación de eventos frecuentes.

Los algoritmos para encontrar elementos frecuentes pueden dividirse en dos clases. Aquellos basados en técnicas de conteo llevan registro de un subconjunto de elementos y monitorean los contadores asociados con los mismos. Por cada entrada nueva, el algoritmo decide si guardar el elemento o no, y con que valor hacerlo. Por otro lado, los algoritmos basados en *sketchs* realizan proyecciones lineales aleatorias de las entradas[3] (vistas como vectores de características) y por lo tanto, no almacenan explícitamente los elementos de entrada. Estos últimos tienen ciertas propiedades que son de gran utilidad para procesamiento de múltiples *streams* de datos.

# 3. Detección de anomalías

Cómo se mencionó anteriormente, determinar eventos frecuentes puede dar indicios de ciertos escenarios. Una práctica muy común en la etapa reconocimiento[3], es el escaneo de puertos; uno o varios atacantes envían paquetes a un rango de puertos para determinar que servicios están activos, generando así grandes volúmenes de tráfico en la red. Otro escenario crítico es el de *command and control (C2)* en donde el atacante tiene control de un conjunto de equipos ya infectados y utiliza sus recursos para atacar otro objetivo. Esto también genera tráfico anómalo y es un indicador crítico ante el cual se deben tomar medidas inmediatamente. Otro indicador son las fluctuaciones repentinas en los flujos de datos, que pueden indicar ataques de denegación de servicio (DoS).

La lista de ataques podría extenderse, pero de forma general pueden definirse dos tipos de comportamientos anómalos que son útiles para identificar amenazas. A continuación se describe el modelo de *data streaming* y se definen los tipos de comportamiento anómalo.

## 3.1. Modelado del problema

Consideremos un conjunto de eventos consecutivos o *stream* de eventos donde cada evento es representado usando una tupla $(x, v_x)$. El elemento $x$ pertenece a un dominio $T = \{0, 1, 2, \ldots, n-1\}$ con $|T| = n$, y $v_x$ es un valor asociado a $x$. El valor $v_x = 1$ representa cantidad de eventos identificados por $x$. Definimos ***heavy hitters*** como aquellos elementos que aparece más frecuentemente en el *stream* de eventos. Los ***heavy changers*** son aquellos elementos que presentan inconsistencias significativas entre el comportamiento observado y el comportamiento normal del flujo de datos (el cual se basa en lo ocurrido en el pasado) en un período de tiempo acotado[5]. En un algoritmo de detección de ***heavy keys***[4] típicamente se realizan dos procedimientos: en el de actualización el valor de cada elemento es procesado y se almacenan los resultados en una estructura de datos intermedia; en el de detección se examina la estructura de datos en cada época y se determinan los *heavy keys*.

Para detectar *heavy keys* se realizan estimaciones de frecuencias en ventanas de tiempo o épocas. En cada época, sea $S(x)$ la suma de los valores $v_x$ del elemento $x$. Sea $D(x)$ la diferencia (en valor absoluto) de $S(x)$ en la época actual y la anterior. Además, sea $U = \sum_{x \in T} S(x)$ la suma total de todos los elementos $x$ en una época. El problema de detectar *heavy keys* consiste en encontrar aquellos elementos cuya suma o diferencias excedan, en valor absoluto, al parámetro $\phi$ en una época. Formalmente, definimos cómo *heavy hitters* a aquellos elementos $x$ con $S(x) \geq \phi_1$, y *heavy changers* a los elementos $x$ con $D(x) \geq \phi_2$. En adelante, referiremos indistintamente a los parámetros $\phi_1$ y $\phi_2$ como $\phi$, teniendo en cuenta que pueden ser diferentes.

# 4. Métodos para determinar elementos frecuentes

La forma clásica de procesar grandes volúmenes de datos y obtener información relevante asume que la totalidad de los datos están almacenados y pueden ser consultados en cualquier

[3]El modelo de seguridad informática llamado *cyber kill chain* describe las 7 etapas que todo atacante ejecuta para lograr su objetivo.[4]

[4]Este término suele utilizase para referirse a ambos tipos de anomalías

orden. Otro enfoque utiliza herramientas del *data streaming* basadas en algoritmos livianos sumamente eficientes (sublineales en el uso de memoria), y estructura de datos probabilistas que pueden usarse para responder ciertas preguntas sobre los datos, de manera precisa y con una probabilidad razonablemente alta. Es decir, en lugar de almacenar la totalidad de los datos para se procesados, se utiliza una representación simplificada de los mismos.

Los *sketch* son estructuras de datos que pueden ser pensadas como proyecciones lineales de los datos de entrada. Si representamos las entradas como vectores, estos pueden ser multiplicados por una *matriz sketch*. El *vector sketch* resultado contiene la información suficiente para responder de forma aproximada ciertas preguntas sobre los datos. Por ejemplo, si codificamos los elementos de un *stream* de datos como vectores cuya $i$-ésima entrada es su frecuencia $f_i$, el *sketch* es el producto de este vector y una matriz: existen diferentes formas de definir la *matriz sketch*, cada una con aplicaciones particulares. Para conteo de eventos se usan familias de funciones de *hash* con ciertas propiedades para definir la proyección lineal.

Los algoritmos basados en *sketches* resuelven el problema de estimación de frecuencia, pero necesitan información adicional para resolver el problema de los elementos frecuentes. Por esto, se suele aumentar la estructura de datos de los *sketch* con algún método de conteo para encontrar elementos frecuentes de manera eficiente[5].

Antes de describir los algoritmos basados en *sketches* es conveniente introducir los filtros *Bloom*, que son una versión simplificada de los primeros, para ganar intuición acerca del funcionamientos de estas estructuras de datos.

## 4.1. Filtros Bloom

El término *filtro Bloom* refiere a una estructura de datos compacta que se usa para representar conjuntos. Sin embargo, no es un conjunto ordinario dado que puede reportar falsos positivos cuando se consulta por la existencia de cierto elemento [6].

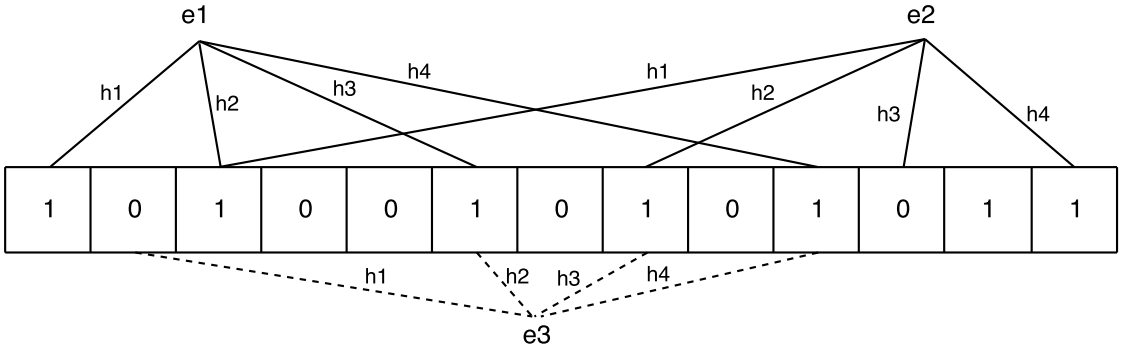



Figura 1: Filtro Bloom

La imagen muestra un filtro de Bloom luego que los elementos $e_1$ y $e_2$ fueron insertados. Podemos observar que $e_3$ no es parte del conjunto dado que al menos uno de los bits en la posición que indican las funciones de hash es 0. La pertenencia de un elemento cualquiera $e'$ al conjunto puede ser reportada erróneamente si las $k$ funciones de hash $h_i(e')$ apuntan a bits del filtro con valor 1.

La ventaja principal de esta estructura de datos sobre las tradicionales es la eficiencia en el uso de memoria. Para representar un conjunto de a lo sumo $n$ elementos, un *filtro Bloom* clásico hace uso de un vector de $m$ bits. Al comienzo todos los bits están en 0. Para insertar un elemento $e$ en el filtro, $k$ bits son actualizados en base a la evaluación de $k$ funciones de hash independientes $h_1(e)$, $h_2(e)$, ..., $h_k(e)$. Para consultar si un elemento es parte del conjunto se deben calcular las $k$ funciones de hash. Si todos los bits que indican las funciones de hash están en 1 es probable que el elemento sea parte del conjunto, de lo contrario es seguro que el elemento no forma parte del conjunto. Sea $c = m/n$, la elección óptima para $k$ está dada por

[5]Dos algoritmos conocidos son usualmente referidos como Majority y Frequent.

$k = int(\ln 2\ c) = int(\ln 2\ m/n)$ y la probabilidad de obtener un falso positivo (en promedio) está dada por $f_{std}(m,n,k) \approx (1 - e^{-kn/m})^k$[7]. Por ejemplo, para un universo de $n = 65536$ elementos y usando $c = 4$ bits por elemento tenemos que $f_{std} = 0{,}0174$.

## 4.2. CountMin Sketch

Los *sketches* son menos populares que los *filtros Bloom* pero comparten ciertas similitudes. Son estructuras de datos que permiten sumarizar un *stream* de datos y pueden utilizarse para resolver el problema de los elementos frecuentes. Esto puede ser llevado a cabo usando menos espacio del que se utilizaría almacenando un contador por elemento, pero permitiendo que los contadores tengan cierto error en algunas ocasiones.

El *sketch* COUNTMINT[8] se utiliza para sumarizar *streams* de datos. Consiste en un arreglo de $d \times w$ contadores y $d$ funciones de hash independientes $h_j$ con $1 < j < d$ que mapean un elemento $i$ a un entero $z \in \{0, 1, \ldots w-1\}$. Por cada elemento procesado se actualiza $d$ contadores en la posición $(j, h_j(i))$ (uno por cada fila) incrementando su valor en 1.

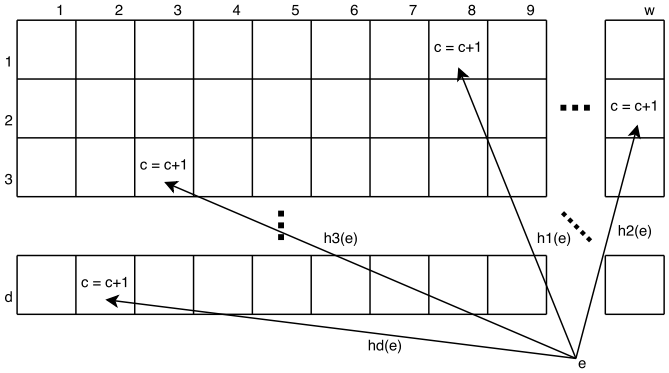



Figura 2: CountMin Sketch

Sea $\mathbf{a}$ un vector de frecuencias, de dimensión $n$, cuyo estado en el tiempo $t$ es $\mathbf{a}(t) = [a_1(t), a_2(t), \ldots, a_n(t)]$. Inicialmente $\mathbf{a}$ es el vector $\mathbf{0}$, es decir $a_i(0) = 0\ \forall i$. Las actualizaciones a los elementos de $\mathbf{a}$ se representan mediante un *stream* de tuplas. De forma general, la tupla $(i_t, c_t)$ representa el $t$-ésimo elemento procesado:[6]

$$a_{i_t}(t) = a_{i_t}(t-1) + c_t$$
$$a_{i_{t'}}(t) = a_{i_{t'}}(t-1)\ \forall t' \neq t$$

Usando *sketches* podemos estimar la frecuencia de ocurrencia $a_i$ de un elemento $i$ mediante la fórmula $\hat{f}_i = min_{1 \leq j \leq d}\{C[j, h_j(i)]\}$. Se puede demostrar que $a_i \leq \hat{f}_i$ y que $\hat{f}_i \leq a_i + \varepsilon||\mathbf{a}||_1$ con probabilidad mayor o igual a $1 - \delta$. Los parámetros $\varepsilon$ y $\delta$ definen las dimensiones del *sketch* como $w = \lceil \frac{e}{\varepsilon} \rceil$ y $d = \lceil \ln \frac{1}{\delta} \rceil$.

**Ejemplo** Para procesar 10 millones de eventos y producir estimaciones con un error hacia arriba menor o igual a 100 con una probabilidad de al menos 0,99 necesitamos $\varepsilon||\mathbf{a}||_1 = 100$ y $\delta = 0{,}01$. En términos las dimensiones del *sketch* $w = 271829$ y $d = 5$.

Finalmente, COUNTMIN es $O(\frac{1}{\epsilon} \log \frac{1}{\delta})$ en términos de memoria y para realizar operaciones de actualización es $O(\log \frac{1}{\delta})$

# 5. Primer incremento: detección local de anomalías

Se diseñó y se implementó un *webservice* que expone los recursos necesarios para actualizar las estructuras de datos según el modelo descrito (de forma transparente para el usuario) y mo-

[6] Para conteo de elementos $c_t = 1$.

nitorear los evento detectados por el sistema. Estos últimos son representados usando un *string* de longitud variable, pudiendo así adaptarse a cualquier uso, y son almacenados en una base de datos central de tipo *clave-valor*. La detección de eventos ocurre automáticamente en intervalos de tiempo regulares y se mantiene un historial de los eventos detectados. La configuración del tamaño de las estructuras de datos, los parámetros de detección y el intervalo de detección pueden ser modifican mediante un archivo de configuración. La implementación que será descrita a continuación puede encontrarse en `https://github.com/leandropineda/sketch-ws/releases/tag/report2`.

## Implementación

El estado del *sketch* se actualiza ejecutando una API REST por cada evento que se quiere ingresar al *webservice*. El cuerpo de la consulta es de tipo *application/json*[9]: contiene el *string* que representa el evento. Si bien en términos de performance esta no es la implementación más eficiente, proveer una interfaz REST para dialogar con el servicio hacer que el sistema pueda ser integrado en cualquier aplicación existente de manera sencilla.

El objeto *sketchManager* es el encargado de crear y actualizar los *sketches*. Cada nuevo evento se asocia a un número entero, que luego es utilizado para actualizar el *sketch*, mediante *RedisManager*. Finalmente, la respuesta del recursos es el valor al que el evento fue asociado. El diagrama de secuencia se muestra en la figura 3.

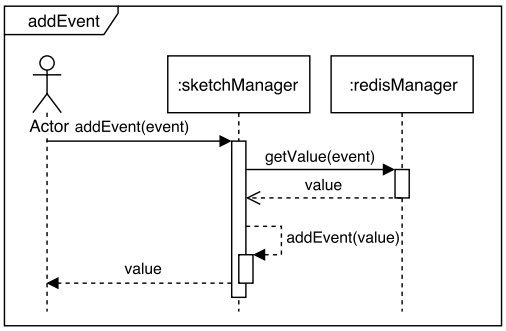



Figura 3: Diagrama de secuencia para nuevos eventos

La asociación de una valor numérico a los eventos se respalda con una base de datos *clave-valor*. El método *getOrSet(event)* es el encargado de proveer el valor asociado al evento. Si no existe en la base de datos, se obtiene un valor para ser asociado al nuevo evento y se guardan el par *<evento, valor>* y su inverso *<valor, evento>* en la base de datos. Finalmente se retorna el valor asociado al evento.

En la figura 4 se muestra el diagrama de secuencia para la asociación de un valor a los eventos:

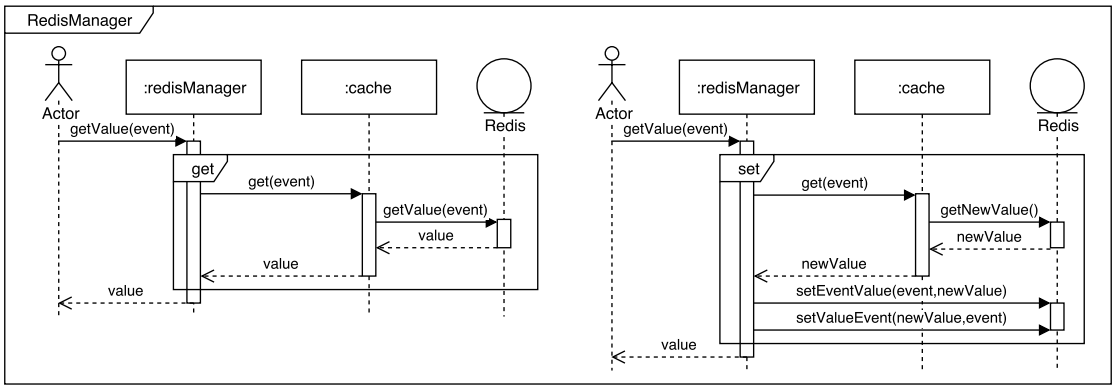



Figura 4: Diagrama de secuencia del flujo de acceso a la base de datos.

Como las consultas a la base de datos usualmente son costosas y generan retrasos (es decir, introducen latencia), se colocó una *cache* en el acceso a la base de datos. Con este diseño se busca solucionar los siguientes problemas:

- Disminuir la carga de procesamiento del *microservicio*: es mucho más eficiente operar con valores enteros que con cadenas de caracteres.
- Los algoritmos de *streaming* están basados en funciones de *hashing* y por lo tanto hacen uso intensivo de aritmética de enteros.
- Ante un escenario de detección distribuida, permite homogeneizar los eventos usando un repositorio centralizado, como una base de datos.

Para realizar la detección de *heavy keys* se necesitan dos *sketches* adyacentes en el tiempo. El mecanismo de rotación de *sketches* está encapsulado en la clase *SketchManager* y se implementa utilizando una *cache* de un objeto que añade el *sketch* activo a *SketchHistory* cada vez que se invalida el mismo. El flujo descrito se puede observar en el diagrama de secuencia de la figura 5.

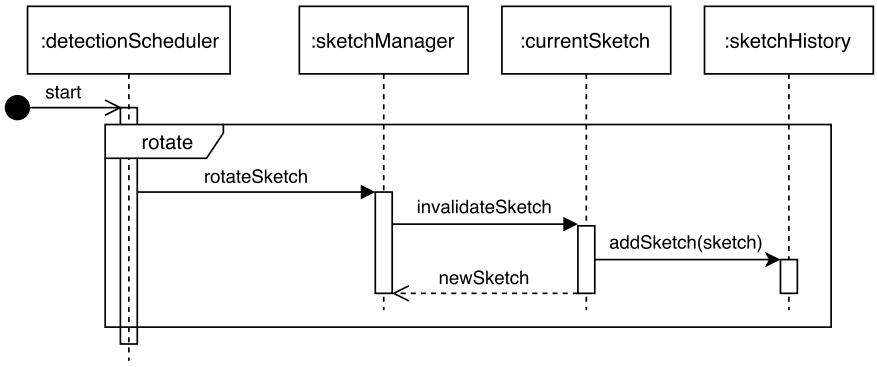



Figura 5: Diagrama de secuencia para la rotación de *sketches*.

Para realizar la detección de *heavy keys* la clase *HeavyKeyDetector* usa los 2 últimos *sketches* disponibles en *sketchHistory*. Dado que los resultados de los algoritmos de detección son valores numéricos, es necesario obtener los eventos a los que fueron asociados previamente. Esta transformación es realizada por *RedisManager*. Finalmente, *HeavyKeyDetector* agrega a *HeavyKeyDetectionHistory* los resultados de la detección son puestos a disposición del usuario mediante recursos en una API REST.

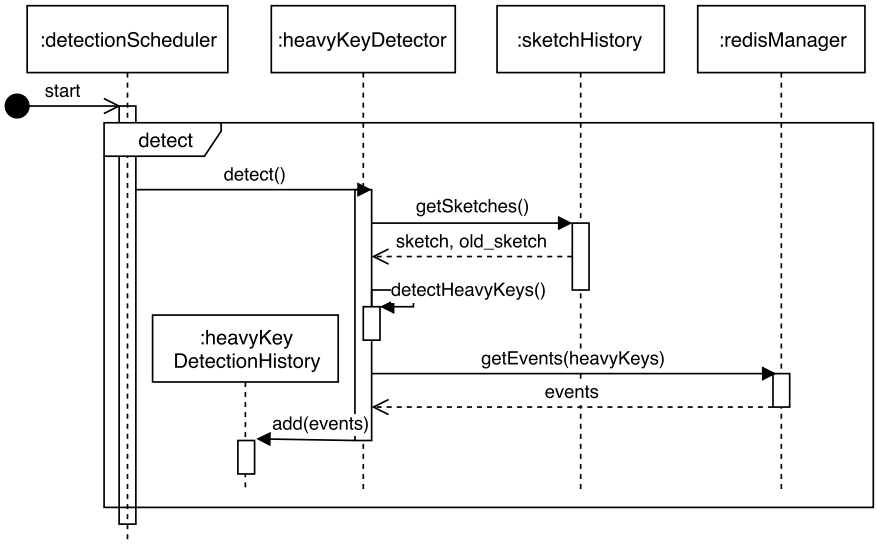



Figura 6: Diagrama de secuencia para detección de *heavy keys*.

La implementación descrita esta empaquetada usando contenedores Docker[7], y todas los componentes necesarios para su funcionamiento así como su configuración están definidas en un archivo *docker-compose*. Estos contenedores permiten empaquetar software en unidades estandarizadas agilizando el desarrollo y puesta en producción de las aplicaciones. El *webservice* provee además los recursos REST para consultar el estado del servicio y sus dependencias.

# 6. Pruebas realizadas

Se realizaron dos experimentos para comprobar el funcionamiento del servicio. En el primero se generaron una serie de eventos en el tiempo con un patrón en particular. En el segundo experimento se utilizaron los flujos TCP/IP de una captura de tráfico de red para generar los eventos generados por un equipo de red.

## 6.1. Patrones de eventos generados localmente

Para el primer experimento se generaron una serie de eventos consecutivos en el tiempo, representados como un par *(tiempo, evento)*. Para cada etiqueta, los eventos son espaciados regularmente en el intervalo de tiempo en el que se generaron, y se agregan pequeñas perturbaciones para evitar que la cantidad de eventos en ventanas adyacentes sea idéntica. Además, se agregan una serie de eventos aleatorios para simular situaciones reales donde pueden existir otras etiquetas que introducen ruido a la señal.

En la figura 7 se observa la distribución de los eventos generados a lo largo del tiempo, para un intervalo de 60 segundos. Cada barra del gráfico representa un intervalo de tiempo de 3 segundos. Los eventos se generan para los intervalos de tiempo definidos de la siguiente manera: en el intervalo $(0, 10)$ se generan 1200 eventos $E1$, en $(10, 40)$ se generan 800 eventos $E2$, en $(33, 36)$ se generan 400 eventos $E3$, en $(40, 47)$ se generan 500 eventos $E4$ y en $(54, 58)$ se generan 400 eventos $E5$. Finalmente se suman 1500 eventos aleatorios a toda la duración de la simulación.

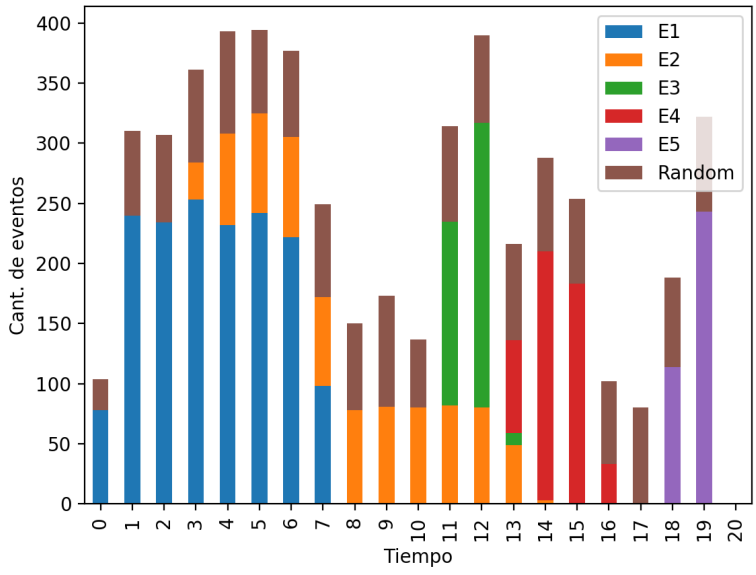



Figura 7: Distribución de los eventos en el tiempo

| Tiempo | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| Heavy Hitters | - | E1 | E1 | E1 | E1 | E1 | E1 |
| Heavy Changers | E1 | - | - | - | - | - | - |
| Tiempo | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
| Heavy Hitters | E1 | - | - | - | - | E3 | - |
| Heavy Changers | E1 | - | - | - | E3 | E3 | E3 |
| Tiempo | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| Heavy Hitters | - | E4 | E4 | - | - | E5 | - |
| Heavy Changers | E4 | - | E4 | - | E5 | E5 | - |

Figura 8: HeavyKeys detectadas

Comprobando los resultados de la detección, se puede observar un defasaje entre los eventos detectados y los mostrados en la figura 7. Esto es así debido a que la simulación no inicia necesariamente en el comienzo de una época de detección (es decir, al comienzo de la ventana de

[7] https://www.docker.com/

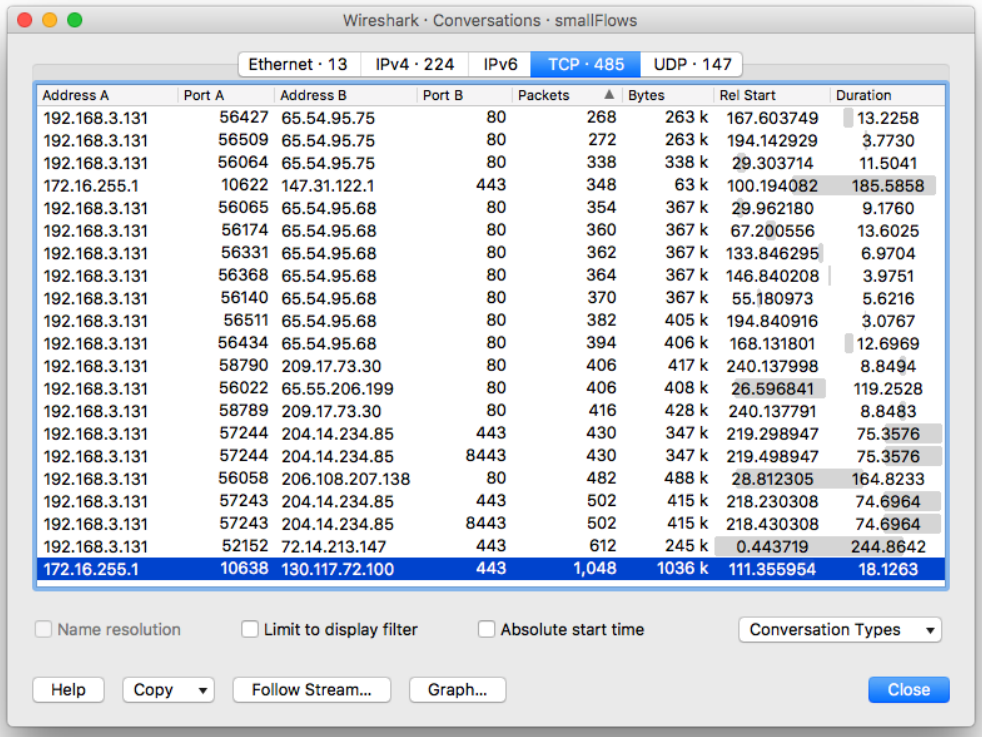
tiempo de 3 segundos configurada para este experimento). Por ejemplo, $E1$ es reportado como *heavy changer* al momento 0 cuando en realidad debería pasar inadvertido. Sin embargo, en el siguiente intervalo es reportado correctamente. De la misma manera, podemos ver que el evento $E2$ no es reportado dado que esta por debajo de los límites configurados.

Los parámetros con los que el servicio estaba configurado a la hora de realizar esta prueba son:

- Sketches
  - rows(5)
  - cols(100)
  - prime(7283)
- Detección
  - heavyHitterThreshold(150)
  - heavyChangerThreshold(100)
  - sketchRotationInterval(3)

## 6.2. Trafico de red

En esta prueba se utilizó un archivo de captura de tráfico de red para emular el comportamiento de los paquetes que atraviesan una interfaz de red. Este dataset, que es usado en la suite *Tcpreplay*[8] para probar performance en switches y adaptadores de red, esta diseñado para generar una gran cantidad de flujos usando varios protocoles, manteniendo bajo el tráfico promedio de la red. Contiene 15000 paquetes de red y una duración de 300 segundos.




| Address A | Port A | Address B | Port B | Packets | Bytes | Rel Start | Duration |
|---|---|---|---|---|---|---|---|
| 192.168.3.131 | 56427 | 65.54.95.75 | 80 | 268 | 263 k | 167.603749 | 13.2258 |
| 192.168.3.131 | 56509 | 65.54.95.75 | 80 | 272 | 263 k | 194.142929 | 3.7730 |
| 192.168.3.131 | 56064 | 65.54.95.75 | 80 | 338 | 338 k | 29.303714 | 11.5041 |
| 172.16.255.1 | 10622 | 147.31.122.1 | 443 | 348 | 63 k | 100.194082 | 185.5858 |
| 192.168.3.131 | 56065 | 65.54.95.68 | 80 | 354 | 367 k | 29.962180 | 9.1760 |
| 192.168.3.131 | 56174 | 65.54.95.68 | 80 | 360 | 367 k | 67.200556 | 13.6025 |
| 192.168.3.131 | 56331 | 65.54.95.68 | 80 | 362 | 367 k | 133.846295 | 6.9704 |
| 192.168.3.131 | 56368 | 65.54.95.68 | 80 | 364 | 367 k | 146.840208 | 3.9751 |
| 192.168.3.131 | 56140 | 65.54.95.68 | 80 | 370 | 367 k | 55.180973 | 5.6216 |
| 192.168.3.131 | 56511 | 65.54.95.68 | 80 | 382 | 405 k | 194.840916 | 3.0767 |
| 192.168.3.131 | 56434 | 65.54.95.68 | 80 | 394 | 406 k | 168.131801 | 12.6969 |
| 192.168.3.131 | 58790 | 209.17.73.30 | 80 | 406 | 417 k | 240.137998 | 8.8494 |
| 192.168.3.131 | 56022 | 65.55.206.199 | 80 | 406 | 408 k | 26.596841 | 119.2528 |
| 192.168.3.131 | 58789 | 209.17.73.30 | 80 | 416 | 428 k | 240.137791 | 8.8483 |
| 192.168.3.131 | 57244 | 204.14.234.85 | 443 | 430 | 347 k | 219.298947 | 75.3576 |
| 192.168.3.131 | 57244 | 204.14.234.85 | 8443 | 430 | 347 k | 219.498947 | 75.3576 |
| 192.168.3.131 | 56058 | 206.108.207.138 | 80 | 482 | 488 k | 28.812305 | 164.8233 |
| 192.168.3.131 | 57243 | 204.14.234.85 | 443 | 502 | 415 k | 218.230308 | 74.6964 |
| 192.168.3.131 | 57243 | 204.14.234.85 | 8443 | 502 | 415 k | 218.430308 | 74.6964 |
| 192.168.3.131 | 52152 | 72.14.213.147 | 443 | 612 | 245 k | 0.443719 | 244.8642 |
| 172.16.255.1 | 10638 | 130.117.72.100 | 443 | 1,048 | 1036 k | 111.355954 | 18.1263 |

Figura 9: Conversaciones TCP del archivo de captura de tráfico de red

Analizando los flujos que se muestran en la figura 9 se puede observar que la conversación con mayor cantidad de paquetes ocurre entre las direcciones 130.117.72.100:443 y 172.16.255.1:10638. A partir del instante 111,35 se intercambian 1048 paquetes en un período de 18,12 segundos,

[8] http://tcpreplay.appneta.com/wiki/captures.html

que son detectados correctamente como *heavy changers*. Los parámetros con los que se configuró el sistema son similares a los usados en el experimento anterior, solo que se aumentaron los umbrales: *heavyHitterThreshold* es 250 y *heavyChangerThreshold* es 200.

# 7. Conclusión y trabajos futuros

Para un flujo de eventos representados como cadenas de caracteres el sistema permite identificar en tiempo real aquellos eventos considerados como *heavy keys* (ver 3.1). Codificando los eventos de esta manera el sistema puede ser integrado en diferentes aplicaciones sin dificultad, dado que expone interfaces sencillas.

Respecto a la instalación del sistema vemos que no ofrece dificultad alguna dado que esta diseñado como un microservicio. Los contenedores Docker funcionan en cualquier sistema operativo y pueden desplegarse tanto en infraestructura *on-prem* como en el *cloud*.

Es posible modificar las propiedades de los parámetros de detección del sistema usando un archivo de configuración según los datos que se quieren procesar. Los umbrales de detección deben ser configurados acorde a la aplicación, pero podría implementarse un mecanismo de ajuste automático.

Finalmente, dado que una REST API no es eficiente para el manejo de muchas peticiones, no se pueden utilizar grande volúmenes de eventos. Para esto, es necesario reemplazar el mecanismo REST por un *message broker* como *Apache ActiveMQ* e implementar el la lógica necesaria para soportar concurrencia en todos los módulos del sistema.

# Referencias


[1] 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.

[2] 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.

[3] 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.

[4] 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.

[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] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.