Análisis de red con *tshark* y *shell scripting*

Redes de Comunicaciones I

Última modificación: miércoles, 11 de octubre de 2017

Contenido

1 Introducción: automatización del análisis de red

El aumento de la importancia de las redes de comunicaciones en todos los niveles ha hecho que, paralelamente, crezcan los esfuerzos dedicados a la monitorización y gestión del tráfico transmitido. Además, dado el gran volumen de datos que se deben examinar durante estos procesos, para posibilitar la realización de análisis en profundidad se requiere la automatización de la obtención de los indicadores clave de prestaciones (KPI, *Key Performance Indicators*) de red más habituales. En este sentido, hay numerosas herramientas que facilitan las tareas de los analistas de redes, siendo particularmente interesantes aquéllas que son libres y de código abierto.

Por ello, presentamos una guía para facilitar el análisis de tráfico de red utilizando este tipo de herramientas. En particular, los objetivos que se persiguen son:

- Presentar distintos elementos de análisis descriptivo de datos, motivando su importancia y utilidad durante el análisis de tráfico y monitorización de redes de comunicaciones.
- Describir el uso de **tshark**, una potente herramienta por línea de comandos basada en **libpcap** para la captura y análisis de tráfico. Facilita el desarrollo de scripts para la obtención de distintos indicadores de prestaciones de red, tanto con un enfoque en vivo como forense –esto es, bien analizando el tráfico que atraviesa un punto particular al instante, bien utilizando capturas de tráfico realizadas con anterioridad. Sus prestaciones son similares a las de **Wireshark**, pero su manejo por línea de comandos facilita la automatización del análisis y añade cierta versatilidad al preprocesado de datos de red para obtener KPIs particulares. Adicionalmente, permite trabajar con archivos más grandes que los soportados en **Wireshark**.
- Ejemplificar el uso de una serie de comandos generales de ***shell scripting*** para sistemas GNU/Linux en el ámbito del análisis de tráfico de red. Para ello, propondremos una serie de casos prácticos que pueden servir de inspiración para la obtención de distintas estadísticas, resúmenes y representaciones de datos que permitan mejorar la comprensión de los fenómenos que aparecen durante la transmisión de tráfico de red.
- Introducir **awk**, un lenguaje de scripting orientado al procesamiento por líneas en archivos de texto. Facilita la lectura y segmentación de líneas de este tipo de archivos, y además incluye una implementación de tablas asociativas que permite utilizar relaciones clave-valor de una forma muy simple y útil para el análisis.

1.1 Estructura del documento

Para cubrir los objetivos planteados, el resto de este documento se estructura de la siguiente forma:

- La sección *Métodos para análisis de datos de red* incluye la descripción de un conjunto de métodos generales de análisis de datos, motivando su relevancia para la monitorización y análisis de tráfico de red. Además, presenta posibles aplicaciones a la

hora de caracterizar el tráfico de red, y estudiar el comportamiento de los KPI durante la gestión de las infraestructuras de telecomunicaciones.

- La sección *Analizando tráfico con tshark* presenta dicha herramienta, explicando sus distintos modos de operación. Así, se verá cómo extraer características de los paquetes, campos concretos de las cabeceras, filtros y otras opciones afianzadas.
- La sección *Shell scripting y análisis de tráfico de red* establece el conjunto de comandos típicos que se usan habitualmente en un análisis de tráfico, y plantea cómo aprovecharlos para automatizar dicho análisis.
- La sección *Una (breve) introducción a awk* introduce este lenguaje, explicando la estructura típica de un script realizado con **awk**, algunas funcionalidades típicas y el uso de las tablas asociativas.
- Finalmente, la sección *Observaciones sobre erratas y correcciones* indica cómo realizar observaciones a la presente guía para su mejora en futuras ediciones.

2 Métodos para análisis de datos de red

A continuación vamos a introducir algunos elementos metodológicos útiles para caracterizar el comportamiento de distintos indicadores que aparecen de forma natural al analizar tráfico de redes de comunicaciones. Como veremos posteriormente, estas herramientas serán muy útiles para presentar información referente a las características del tráfico y de los flujos de red, así como para evaluar los patrones de utilización de los recursos disponibles.

2.1 Función de distribución acumulada empírica y percentiles

La función de distribución acumulada empírica (ECDF) permite mostrar el comportamiento de una variable aleatoria de un simple vistazo. La definición matemática de la ECDF para una variable aleatoria X se muestra a continuación:

$$\hat{\mathbb{P}}\{\mathrm{X} \leq x\} = \frac{1}{n}\sum_{k=1}^{n} \mathbf{1}_{x_k \leq x}$$

donde $x_k, k = 1 \ldots n$ son las distintas observaciones de la variable aleatoria y $\mathbf{1}_{x_k \leq x}$ es la función indicatriz que vale 1 si se cumple $x_k \leq x$ y 0 en caso contrario. En la Figura 1 vemos cómo se construye la ECDF a partir de un conjunto de observaciones. El valor en cada punto x se obtiene como el número de valores menores o iguales que dicho valor x entre el número total de observaciones. **Así, una vez ordenados los valores de las distintas observaciones de la variable aleatoria, el valor de la ECDF se puede computar haciendo corresponder a cada valor de x su posición dentro del conjunto, dividida por el número total de observaciones.** Hay que remarcar que en el caso de valores repetidos se puede resolver el problema de los múltiples valores de la función quedándonos con el máximo de la función –aunque una representación adecuada permite, en términos de visualización, omitir este paso de resolución de la múltiple definición de la función para un mismo x.

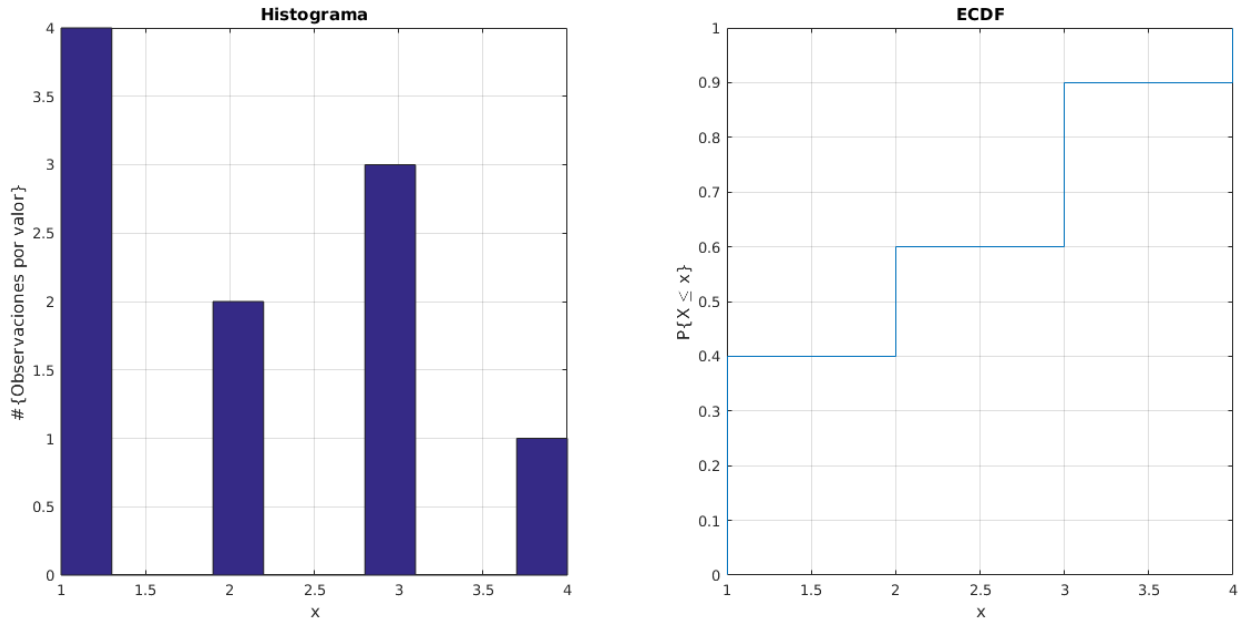

Figura 1. Histograma y ECDF de 10 observaciones de una cierta variable aleatoria.

De forma inmediata, se pueden extraer las siguientes propiedades importantes que tiene la ECDF, que siempre deben cumplirse:

- **Vale 0 cuando x tiende a $-\infty$.**
- **Vale 1 cuando x tiende a $+\infty$.**
- **Es una función no decreciente.**

En la Figura 2 vemos el sentido de estas propiedades en el caso de una serie de 1000 observaciones de una variable aleatoria que sigue una distribución Normal estándar.

A la vista de este ejemplo, podemos ver cómo utilizando la ECDF es posible caracterizar la probabilidad (en nuestro caso, al ser empírica, frecuencia de aparición) de los distintos valores que toma una variable aleatoria. En este sentido, resulta particularmente interesante tener en cuenta los valores con una alta frecuencia de aparición, los valores centrales que toma la variable estudiada, y los valores extremos. Para ello podemos considerar los **percentiles** (a los que nos referiremos de aquí en adelante como P_i). Formalmente, son una medida de posición correspondiente a los valores que dejan por debajo ciertos porcentajes de observaciones, una vez éstas han sido ordenadas en sentido creciente. Esto es:

$$P_i = \min\left\{x : \widehat{\mathbb{P}}\{\mathrm{X} \leq x\} \geq \frac{i}{100}\right\}$$

Por ejemplo, diremos que $P_{25} = \min\{x : \widehat{\mathbb{P}}\{\mathrm{X} \leq x\} \geq 0{,}25\}$

Algunos percentiles importantes son aquéllos que marcan la centralidad de la muestra (P_{50}, que es en cierto sentido una aproximación de la mediana de la muestra), y los que marcan valores extremos, siendo P_5 o P_{10} típicos para caracterizar los valores pequeños, y P_{90} o P_{95} para los grandes.

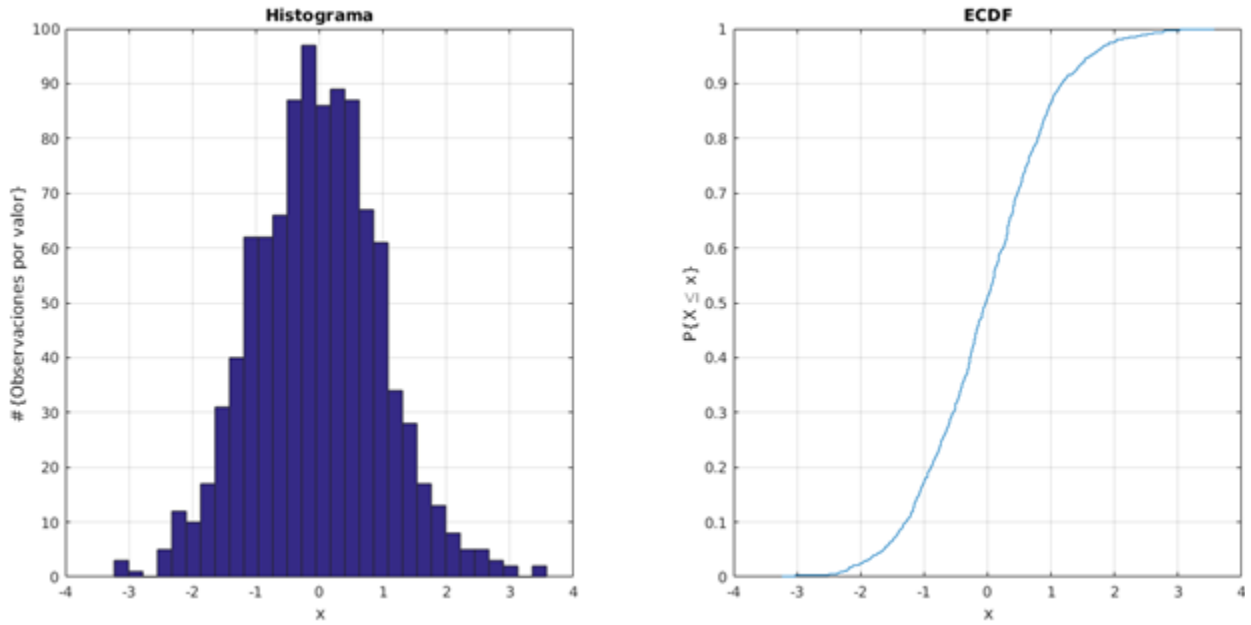

Figura 2. Histograma y ECDF correspondiente a 1000 observaciones de una variable aleatoria que sigue una distribución Normal estándar.

2.2 Valores atípicos

Los valores atípicos (en inglés *outliers*) se corresponden con eventos observados raramente. Su aparición, aunque sea extraña, puede producir perturbaciones en medidas no robustas, tales como la media muestral. Por esta razón, su localización es interesante de cara a la realización de análisis posteriores, ya que una limpieza de los datos que identifique valores atípicos puede mejorar las conclusiones extraídas, pudiendo incluso ayudar a comprender fenómenos ocultos en los datos o identificar errores experimentales o metodológicos.

La ordenación asociada a los niveles de probabilidad acumulada de los valores que toma la variable aleatoria permite definir qué se puede considerar como valores atípicos u *outliers*. Formalmente, existen diferentes definiciones asociadas a distintas formas de capturar la idea de valores alejados de lo usual. Una alternativa simple podría ser usar directamente umbrales sobre los valores extremos. Por ejemplo, podemos pensar que sólo un 10% de los datos aparecen fuera del intervalo entre percentiles $[P_5, P_{95}]$, y que esa frecuencia de aparición ya captura la idea de evento atípico.

Otras definiciones más sofisticadas tienen también en cuenta aspectos tales como la dispersión de la distribución de la muestra utilizando el rango intercuartílico (IQR, *Interquantile Range*), definido como:

$$IQR = P_{75} - P_{25}$$

En base a esto, otro posible intervalo para detectar valores atípicos sería el dado por $[P_{25} - 1.5 \times IQR, P_{75} + 1.5 \times IQR\ ]$, que incorpora la dispersión (medida mediante el valor de IQR) además de una baja frecuencia de aparición. En particular, estos límites son los que se utilizan en las representaciones de tipo diagrama de cajas o *boxplot* para indicar los valores atípicos de un conjunto de observaciones.

2.3 Series temporales

En general, entendemos como serie temporal una secuencia de observaciones o mediciones de una magnitud realizadas a lo largo de un intervalo temporal. Representan la evolución temporal de un fenómeno, de forma que permiten estudiar tendencias y comportamientos que tienen una evolución a lo largo de instantes temporales.

Para representar series temporales, se puede utilizar la siguiente notación:

$$\{X_{t_i}\}_{i=1\ldots k}$$

Esta representación relaciona los valores observados de una magnitud (X) con los valores temporales en los que se obtuvieron las observaciones (t_i, $i = 1 \ldots k$). De esta forma, podemos entender las series temporales como versiones muestreadas de funciones cuya variable independiente es el tiempo. En la Figura 3 se muestra un ejemplo de serie temporal, en el que se ve la evolución de una cierta magnitud (en el ámbito de la asignatura, típicamente, caudal consumido) con respecto al tiempo en incrementos de 1 s.

Existe un amplio abanico de métodos para estudiar distintos parámetros de estas series, y distintos modelos que permiten realizar análisis descriptivo e inferencia estadística sobre ellas. Sin entrar en mayores detalles, y atendiendo a su sentido descriptivo más básico, podemos inferir que se trata de una representación de medidas que, por inspección visual directa, ofrece a los analistas información acerca del comportamiento de una cierta magnitud a lo largo del tiempo. Esto permite la detección de cambios y la caracterización y clasificación de fenómenos que, de forma natural, observamos a través de secuencias temporales.

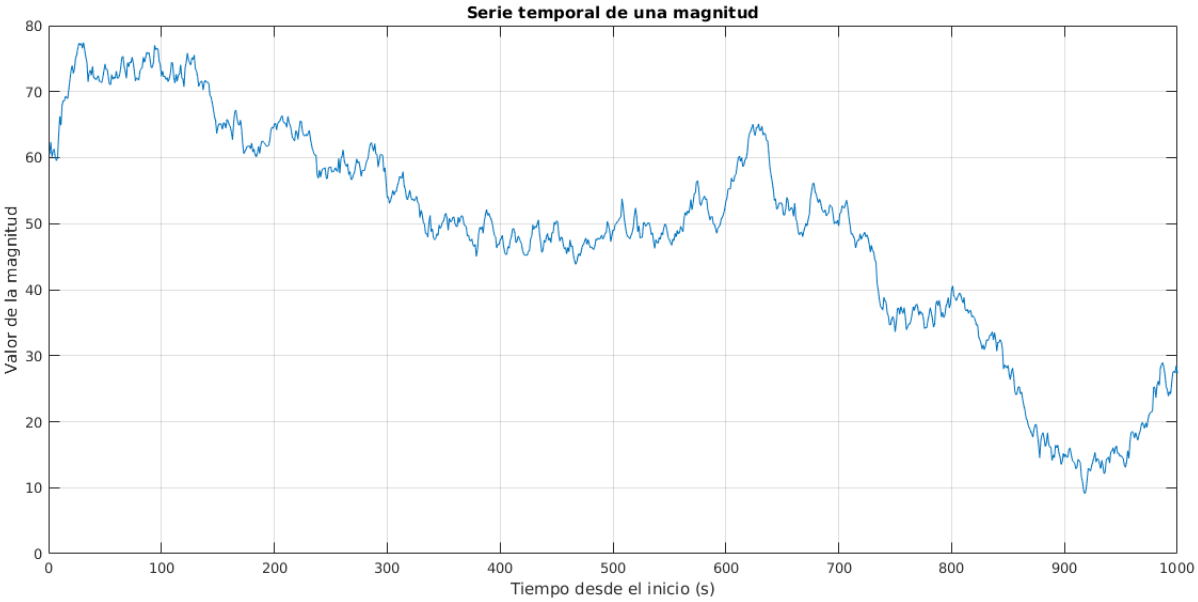

Figura 3. Ejemplo de serie temporal.

3 Analizando tráfico con tshark

A continuación presentamos la forma de invocar **tshark** desde la consola de comandos para conseguir obtener distintos parámetros a partir de capturas de tráfico. Para ello, distinguiremos entre características de los paquetes (p.ej., momento de captura o bytes capturados) y campos de cabeceras (p.ej., direcciones IP o puertos), y también veremos cómo filtrar el tráfico para que los resultados se restrinjan utilizando filtros de visualización. En general, para obtener una serie de campos de datos a partir de una captura de tráfico debemos utilizar la llamada:

```
tshark -r <nombre del archivo pcap> -T fields [-e <nombre de campo>]
```

A continuación desglosaremos algunas de las opciones de campos que se pueden extraer[1]. Como veremos, los nombres de los campos se corresponden con los que se utilizan en los filtros de visualización de **Wireshark**.

3.1 Opciones para extracción de características de paquetes

tshark permite extraer distintas características de los paquetes capturados, siendo las más interesantes para nuestros intereses las siguientes[2]:

Nombre del campo	Significado
frame.len	Longitud original del paquete
frame.time_delta	Tiempo transcurrido desde el anterior **paquete capturado**
frame.time_delta_displayed	Tiempo transcurrido desde el anterior **paquete mostrado**
frame.time_epoch	Tiempo de captura del paquete (formato EPOCH)
frame.time_relative	Tiempo de captura del paquete (desde el comienzo)

Por ejemplo, para obtener los tiempos de captura en formato EPOCH y tamaños de los paquetes de una traza, se puede utilizar el comando:

```
tshark -r <nombre del archivo pcap> -T fields -e frame.time_epoch -e frame.len
```

3.2 Opciones para extracción de campos de cabeceras de los paquetes

Por otro lado, **tshark** también permite extraer los campos de las cabeceras de los distintos protocolos presentes en el paquete. A continuación se presenta un resumen (por niveles) de algunos campos de especial relevancia:

[1] Listado completo de opciones disponibles: https://www.wireshark.org/docs/man-pages/tshark.html
[2] Listado completo de opciones disponibles: https://www.wireshark.org/docs/dfref/f/frame.html

Nivel	Nombre del campo	Significado
Ethernet[3]	**eth.addr**	Dirección MAC (destino y/u origen)
	eth.dst	Dirección MAC de destino
	eth.src	Dirección MAC de origen
	eth.type	Tipo Ethernet (protocolo encapsulado)
VLAN[4]	**vlan.etype**	Tipo VLAN (protocolo encapsulado)
	vlan.id	Etiqueta VLAN
IP[5]	**ip.addr**	Dirección IP (destino y/u origen)
	ip.dst	Dirección IP de destino
	ip.src	Dirección IP de origen
	ip.proto	Protocolo encasulado en IP
TCP[6]	**tcp.dstport**	Puerto TCP de destino
	tcp.srcport	Puerto TCP de origen
UDP[7]	**udp.dstport**	Puerto UDP de destino
	udp.srcport	Puerto UDP de origen

Por ejemplo, para obtener una salida de texto que incluya los tiempos de captura en formato EPOCH y puerto destino UDP se podría usar la siguiente invocación:

```
tshark -r <nombre del archivo pcap> -T fields -e frame.time_epoch -e udp.dstport
```

Es importante señalar que los registros que proporciona la salida de **tshark** si se usan este tipo de llamadas pueden tener un número variable de campos. Si solicitamos que se muestren campos correspondientes a la cabecera de un cierto protocolo, los paquetes que no incluyan dicho protocolo generarán registros con campos vacíos. Como veremos en el apartado siguiente, esto se puede resolver utilizando filtros de visualización que fuercen que el análisis se ejecute únicamente sobre paquetes que cumplan características como incluir un cierto protocolo.

Por otro lado, también es posible obtener una disección completa de los paquetes (similar a la proporcionada en la interfaz de **Wireshark**) utilizando **tshark**. Para ello, debemos invocar el programa usando la siguiente llamada:

```
tshark -r <nombre del archivo pcap> -T text –V
```

[3] Listado completo de opciones disponibles: https://www.wireshark.org/docs/dfref/e/eth.html
[4] Listado completo de opciones disponibles: https://www.wireshark.org/docs/dfref/v/vlan.html
[5] Listado completo de opciones disponibles: https://www.wireshark.org/docs/dfref/i/ip.html
[6] Listado completo de opciones disponibles: https://www.wireshark.org/docs/dfref/t/tcp.html
[7] Listado completo de opciones disponibles: https://www.wireshark.org/docs/dfref/u/udp.html

3.3 Aplicación de filtros de visualización

Al igual que en la interfaz gráfica de **Wireshark**, **tshark** permite la introducción de filtros de visualización para discriminar el tráfico analizado. Para ello, debemos utilizar la siguiente sintaxis (nótese el uso de apóstrofes para delimitar el filtro):

```
tshark –r <nombre del archivo pcap> -Y '<filtro de visualización>'
```

De esta forma, podemos centrarnos en paquetes que cumplan un conjunto de características: por ejemplo, para obtener el tiempo en el que fueron capturados paquetes TCP con puerto origen 5060 podríamos usar la invocación:

```
tshark -r <nombre del archivo pcap> -T fields -e frame.time_epoch -Y 'tcp.dstport eq
5060'
```

Como comentamos antes, esto permite evitar problemas asociados en el caso de solicitar campos que sólo estén presentes en paquetes correspondientes a protocolos particulares. Por ejemplo, en el caso del apartado anterior en el que obteníamos el puerto destino UDP, podríamos suprimir la salida de registros que no tengan un valor vacío en este campo modificando la invocación que indicábamos por:

```
tshark -r <nombre del archivo pcap> -T fields -e frame.time_epoch -e udp.dstport –Y
'udp'
```

Por supuesto, la sintaxis de filtros de visualización de **Wireshark** y **tshark** es la misma, así que ante la duda, la interfaz gráfica nos puede ayudar a comprobar que los filtros que estamos usando tienen una estructura correcta[8].

3.4 Análisis avanzado

Finalmente, **tshark** nos permite obtener otros resultados más avanzados a partir de archivos de tráfico. En primer lugar, **tshark** permite analizar las conversaciones definidas en distintos niveles, utilizando la llamada:

```
tshark -r <nombre del archivo pcap> -qz conv,<nivel>
```

donde nivel puede tomar cualquiera de los siguientes valores: {eth, fc, fddi, ip, ipx, sctp, tcp, tr, udp}. Las conversaciones identifican flujos de datos entre hosts, definidos de una determinada forma según el nivel al que se consideren –p.ej., en el caso de conversaciones en el nivel

[8] Además, eso quiere decir que podemos usar los mismos recursos para mejorar nuestro conocimiento de las dos herramientas: http://packetlife.net/media/library/13/Wireshark_Display_Filters.pdf

Ethernet, se utilizarían pares de direcciones MAC, pero si estamos en el nivel TCP, consideraríamos pares de direcciones IP y puertos.

También incluye mecanismos para obtener series con distinta granularidad, tal y como se pueden configurar desde la interfaz gráfica de **Wireshark**. Para ello, se puede usar el siguiente esquema de invocación:

```
tshark -r <nombre del archivo pcap> -qz
io,stat,<intervalo>[,"<filtro>"|"<métrica><filtro_de_métrica>"[,"<filtro>"|"<métric
a><filtro_de_métrica>"[…]]]
```

donde métrica puede ser cualquier tipo de estadística sobre los paquetes, incluyendo funciones de agregación varias –e.g., AVG, COUNT, SUM,… y filtro de métrica debe contener al menos el campo sobre el que se aplica la función. Estas funciones devuelven valores enteros. Puede haber tantas columnas como se indique. En las columnas que se hayan indicado sólo con filtro, se imprimirá una columna con las tramas y otra con los bytes que cumplen ese filtro en ese intervalo.

Por ejemplo, las dos invocaciones siguientes presentarían resultados semejantes:

```
tshark -r salida.pcap -qz io,stat,1,"ip"

tshark -r salida.pcap -qz io,stat,1,"COUNT(ip)ip","SUM(frame.len)frame.len&&ip"
```

4 Shell scripting y análisis de tráfico de red

En esta sección presentamos algunos comandos que pueden ser aplicados para realizar tareas de gestión y procesamiento de datos y comentamos cómo pueden utilizarse en combinación con **tshark** y **awk** (que veremos en la siguiente sección), con el fin de automatizar análisis de tráfico de red y obtener representaciones y resultados durante las labores de monitorización de red.

4.1 Comandos

A continuación comentamos, en orden alfabético, algunos comandos que nos van a permitir realizar tareas generales para procesar archivos que incluyen datos, y algunas de las opciones que incluyen.

- **cat:** permite concatenar ficheros varios ficheros, volcando el resultado en la salida estándar. Nos va a permitir, por tanto, agregar resultados procedentes del análisis de distintos conjuntos de datos. Ejemplos de uso:

```
# Concatena el contenido de resultados_origen.dat y resultados_destino.dat, y
# guarda el resultado en resultados_total.dat
cat resultados_origen.dat resultados_destino.dat > resultados_total.dat
```

- **grep:** permite seleccionar las líneas de un archivo en las que aparece un determinado patrón. Tiene una multitud de opciones y resulta útil a la hora de extraer campos en archivos de texto. Ejemplos de uso:

```
# Busca y devuelve las líneas del archivo que incluyen la cadena <cadena>
grep '<cadena>'  <nombre del archivo>
```

- **head:** envía a la salida estándar las primeras líneas de un archivo. Permite la configuración del número de líneas que son mostradas. Ejemplos de uso:

```
# Devuelve las primeras cinco líneas de un archivo
head -n 5 <nombre del archivo>
```

- **sort:** envía a la salida estándar el resultado de ordenar el archivo pasado como argumento. Permite configurar qué tipo de orden se debe aplicar (por ejemplo, alfanumérico para cadenas), el sentido del orden (de mayor a menor o viceversa), ordenamientos por campos de registro (según las columnas que indiquemos), etc. Ejemplos de uso:

```
# Devuelve el archivo ordenado alfanuméricamente –es decir, 10 < 2
sort <nombre del archivo>

# Devuelve el archivo ordenado numéricamente –es decir, 2 < 10
sort -n <nombre del archivo>
```

- **tail:** es el comando complementario a head, ya que envía a la salida estándar las últimas líneas de un archivo. Además, tiene opciones que facilitan seguir la escritura de datos en archivos mientras se están actualizando. Ejemplo de uso:

```
# Devuelve las últimas cinco líneas de un archivo
tail -n 5 <nombre de archivo>
```

- **uniq:** permite extraer los diferentes valores incluidos dentro de un archivo. Para poder realizar esta extracción es habitual realizar previamente un **sort**, de forma que valores iguales aparezcan juntos. Ejemplo de uso:

```
# Devuelve los valores únicos incluidos en el archivo y el número de
repeticiones de cada uno de ellos
uniq –c <nombre de archivo>
```

- **wc:** proporciona estadísticas del archivo pasado como parámetro, tales como el número de caracteres o el número de líneas. Ejemplo de uso:

```
# Devuelve el número de líneas del archivo
wc –l <nombre de archivo>

# Devuelve el número de líneas del archivo. La salida no incluye el nombre de
archivo, al leer de entrada estándar
wc –l < <nombre de archivo>
```

4.2 Automatización del análisis

Con los elementos vistos hasta el momento, podemos construir fácilmente herramientas bastante automatizadas para extraer las características del tráfico que atraviesa una red:

- Por un lado, hemos visto cómo obtener tablas de registros que incluyen distintos parámetros de red utilizando **tshark**. Estos resultados se escriben en la salida estándar, de modo que resulta sencillo almacenarlos en distintos archivos de texto. Así, es posible utilizar estrategias para analizar distintos conjuntos de paquetes utilizando varias invocaciones sobre un mismo archivo .pcap, unificando y consolidando posteriormente los resultados.
- Además, **sort**, **cat** y **head** nos van a permitir limpiar la información en estos archivos. Finalmente, **awk** es el complemento perfecto a todo lo anterior, como veremos en la sección siguiente.

- Posteriormente, y tras almacenar los resultados obtenidos en archivos de texto, es posible obtener representaciones gráficas de los mismos empleando bibliotecas y herramientas apropiadas –p.ej., GNUPlot, python, Matlab, etc.

5 Una (breve) introducción a awk

awk es un lenguaje de *scripting* muy versátil que facilita el procesamiento de archivos de texto por líneas. Su sintaxis es muy parecida al lenguaje de programación C, incorporando un gran número de funciones aritméticas y para manejo de cadenas, y una separación implícita de los campos que componen las distintas líneas del archivo basada en la definición de separadores de campos. Además, **awk** incluye una implementación de tablas asociativas que, como veremos, simplifica la obtención de métricas con una estructura clave-valor bien definida. Para ello, en los próximos apartados detallaremos la estructura general de un *script* en **awk**, algunas de las funciones que incluye, y cómo utilizar las tablas asociativas que incluye.

5.1 Estructura típica de un *script* de awk

Los *scripts* de **awk** tienen una estructura dividida en tres bloques:

- **Un bloque BEGIN que incluye las acciones que se realizan antes de comenzar la lectura** del archivo.
- **Un bloque que incluye las acciones que se van a realizar por cada línea** (registro) del archivo.
- **Un bloque END que especifica acciones que se ejecutan una vez se han leído todas las líneas** del archivo.

Por ejemplo, el *script* presentado a continuación inicializa una variable a 0 antes de comenzar a leer el archivo de entrada, incrementa su valor en una unidad por cada línea del archivo, e imprime el valor final de la variable una vez leídas todas las líneas del archivo:

```
awk 'BEGIN{n_lines=0;} {n_lines=n_lines+1;} END{print n_lines;}' <archivo a procesar>
```

Cabe mencionar que **awk** inicializa a 0 todas las variables la primera vez que se usan, por lo que no sería necesario añadir el bloque BEGIN en el script anterior.

También es posible escribir *scripts* en archivos de texto, pasándolos como parámetro de ejecución:

```
echo 'BEGIN{n_lines=0;} {n_lines=n_lines+1;} END{print n_lines;}' > example.awk;
awk –f example.awk <archivo a procesar>
```

5.2 Algunas funcionalidades de awk

En primer lugar, vamos a comentar algunas variables implícitas que incluye **awk** y que pueden resultar útiles durante la implementación de scripts basados en esta herramienta:

Variable	Descripción
FS	Separador de campos del archivo de entrada. Puede ser un carácter o una expresión regular. Se puede cambiar su valor por defecto bien dentro del cuerpo del script o con el parámetro de ejecución –F seguido del separador de campos (por ejemplo, si el separador es una coma: -F',')
RS	Separador de registros. Típicamente cada registro está delimitado por el fin de línea (\r\n en Unix).
FNR	Posición del registro que se está procesando dentro del archivo.
NR	Número de registros procesados desde el comienzo de ejecución.
NF	Número de campos en el registro actual.
$0	Registro actual completo.
$i	Accede al campo i-ésimo del registro actual. (p.ej: $1 es el primer campo del registro).

Además, para maximizar su usabilidad, **awk** incluye un conjunto de funciones que permiten realizar procesado avanzado de los campos encontrados en cada línea / registro.

Función	Descripción
int(valor)	Parte entera de valor.
sqrt(valor)	Raíz cuadrada de valor.
index(cadena, subcadena)	Devuelve la posición del carácter comienzo de la primera aparición de subcadena en cadena.
length(cadena)	Longitud de la cadena.
split(cadena, salida, sep)	Divide cadena en fragmentos separados por sep y guarda el resultado en salida, que es un array con los distintos fragmentos generados.
sprintf(formato,expresion1,…)	Devuelve una cadena con la estructura indicada en formato con los valores indicados en expresion1,…
substr(cadena,emp[, long])	Devuelve una subcadena de cadena, de longitud long y empezando en la posición emp .
print [variables]	Imprime el valor de variables a la salida estándar.

Finalmente, **awk** permite invocaciones incluyendo variables que pueden ser pasadas como parámetros de ejecución. Para poder pasar una variable como argumento, debemos utilizar la siguiente sintaxis:

```
awk –v <nombre de la variable>=<valor asignado a la variable> <script> <archivo a procesar>
```

Esto, combinado con los comandos antes presentados, proporciona una gran flexibilidad a la hora de manipular archivos de texto. Por ejemplo, si queremos procesar el archivo example.dat imprimiendo por cada línea del archivo el número total de líneas que tiene, podemos utilizar una combinación de **wc** y esta última funcionalidad de **awk**:

```
awk -v cont="$(wc -l example.dat)" 'BEGIN{split(cont,tokens," "); n_lines=tokens[1];}
{print n_lines} END{}' example.dat
```

Nótese que si un bloque está vacío no es necesario escribirlo. En el caso anterior, el bloque END se podría omitir.

5.3 Tablas asociativas

Para concluir, vamos a presentar las tablas asociativas que **awk** implementa de forma nativa, y que le proporcionan su mayor potencia. Una tabla asociativa es una implementación de una relación clave-valor construida sobre tablas indexadas por **cualquier** tipo de variable (incluyendo cadenas de texto). En términos prácticos, esto supone manejar *arrays* similares a los de cualquier lenguaje de programación, utilizando como índice cualquier tipo de variable.

Veamos qué quiere decir esto usando un ejemplo, en el que queremos:

- Analizar un archivo formado por registros / líneas con dos campos separados por coma.
- El primer campo ($1) indica el valor de una magnitud identificada con el segundo campo ($2).
- Queremos sumar el valor de la magnitud ($1) correspondiente a registros con el mismo identificador ($2), obteniendo la suma agregada de todos los valores.
- Imprimir, separados por comas, todos los valores de identificador (esto es, todos los valores de $2) junto con los correspondientes valores de las sumas agregadas.

Para hacer eso, lo único que tenemos que hacer es utilizar el siguiente script:

```
BEGIN {
       FS = ",";
}
{
       suma_valores[$2] = suma_valores[$2] + $1;
}
END {
       for (valor in suma_valores) {

              print valor","suma_valores[valor];
       }
}
```

Lamentablemente, la forma en que **awk** ordena y recorre el *array* no suele ser la que uno deseara. Para solventarlo, se puede hacer un **sort** sobre la salida que genere **awk**.

6 Observaciones sobre erratas y correcciones

En el caso de detectar alguna errata o error en el contenido de este documento, o para comunicar sugerencias de actualización y extensión, puede ponerse en contacto con David Muelas o Jorge López de Vergara a través de las direcciones de correo electrónico dav.muelas@uam.es y jorge.lopez_vergara@uam.es.