

Redes Multimedia- Prácticas 2020

Práctica 4: Ingeniería de tráfico

Turno y pareja: 2461_06

Integrantes:

Pablo Díez del Pozo

Alejandro Alcalá Álvarez

Fecha de entrega: 06 de mayo de 2020

Contenido

1Introducción.....	2
2Realización de la práctica.....	2
3Conclusiones.....	13

1 Introducción

El objetivo de esta práctica es saber los mecanismos que existen en las redes multimedia para proporcionar control de calidad, donde vamos a aprender diferentes métodos y donde llegaremos a ver en esta práctica lo siguiente:

- Ver el funcionamiento de un token bucket.
- Ver como afecta el token bucket a los flujos multimedia por la red.
- Comparar la técnica de conformado de tráfico con la política de tráfico.
- Aplicar los conocimientos que hemos aprendido en prácticas anteriores para poder realizar un buen estudio de esta técnica.

2 Realización de la práctica

1. Estudie el código proporcionado, emuladorTB.py. ¿Cuáles son los parámetros del algoritmo (b y r) que se manejan para los distintos escenarios propuestos en el código? Indique el CIR en bits/s y MBS en bytes que se deberían alcanzar para el escenario que coincide con su número de pareja. (0,5 puntos)

El código está dividido en dos hilos, que estos hilos se crean para que el funcionamiento del proceso principal no se sobrecarga y haga que esto influya a la hora del funcionamiento del Token Bucket. El primer hilo consiste en llenar el cubo → este hilo se encarga de introducir fichas en el cubo a la tasa que se corresponda con el escenario seleccionado.

```
# Esta clase de hilo se encarga de llenar el cubo del token bucket
class LlenarCubo(threading.Thread):
    def __init__(self):
        threading.Thread.__init__(self)
    def run(self):
        # Se usa la variable global cubo, que el otro hilo irá vaciando
        global cubo

        # Inicialmente se llena el cubo con todos los tokens
        cubo=tamano_cubo

        # Posteriormente se comprueba si hay que rellenarlo o sigue lleno
        while True:
            if cubo<tamano_cubo:
                if cubo+num_tokens>tamano_cubo : # El cubo no puede rebosar
                    cubo=tamano_cubo
                else:
                    cubo+=num_tokens
            #print(cubo)
            # Esperamos para seguir llenando el cubo
            time.sleep(1/tasa_token)
```

En el segundo hilo → se encarga de enviar el paquete al servidor, donde se cogen de una cola que hemos creado con anterioridad. Este hilo también se encarga de quitar del cubo los bytes que hemos utilizado.

```

# Esta clase de hilo se encarga de sacar los paquetes de la cola según la tasa fijada por el Token Bucket y enviarlos a la red
class Desencolar(threading.Thread):
    def __init__(self):
        threading.Thread.__init__(self)

    def run(self):
        # Se usan las variables globales cola, que se irá rellenando en el hilo principal y aquí vaciando
        # y cubo, que se irá vaciando aquí, y rellenando en otro hilo
        global cola, cubo

        while True:
            if not cola.empty(): # Si la cola no está vacía
                datos = cola.get()
                # Se obtiene el número de tokens a vaciar del cubo
                len_datos=len(datos)+CAB_ETH+CAB_IP+CAB_UDP
                # Se imprimen los datos y su longitud
                #print('datos:',len_datos)
                while len_datos>cubo: # No hay tokens suficientes
                    # Esperamos a que se llene el cubo, durmiendo mientras tanto.
                    #print(0)
                    time.sleep(0.001) # dormimos en cuantos de 1 ms
                # Quitamos los tokens necesarios del cubo
                cubo-=len_datos
                #print(cubo)
                # Enviamos finalmente el paquete
                sockC.sendto(datos, (ip_destino.exploded, puerto_destino))

```

Por parámetros introducimos la IP y puerto que recibimos paquetes, y también introducimos la IP y puerto que enviamos los paquetes que hemos recibido del cliente.

Como podemos observar tenemos diferentes escenarios con distintos llenados del cubo y distintos tamaños.

```

# Escenarios que hay en el programa
TAMANOS_CUBO = [5000, 10000, 20000, 25000, 5000, 10000, 20000, 25000, 5000] # Bytes
TASAS_TOKEN = [200, 100, 50, 40, 400, 200, 100, 80, 250, 125, 63, 50, 300] # Número de adiciones al cubo por segundo

```

El emulador está escuchando todo el rato por si recibe paquetes del cliente y los encola en una cola creada para que lo utilice el Token Bucket.

```

while True:
    # Se recibe un paquete
    data, addr = sockS.recvfrom(2048) # se pueden recibir segmentos de hasta 2048 bytes, lo normal es que no sean mayores de 1472

    # Se meten los datos recibidos en la cola, que irán siendo desencolados por el thread de desencolado
    cola.put(data)

```

El cálculo del CIR en bits/s es el siguiente:

$$\begin{aligned}
 \text{tokens/seg} &= 200 \text{ adiciones/seg} \times 1500 \text{ tokens} = 300000 \text{ tokens/seg} \\
 \text{CIR}(\text{bits/s}) &= 300000 \times 8 = 2400000 \text{ bits/s} \\
 \text{CIR}(\text{Kbits/seg}) &= 2400 \text{ Kbits/seg} = 2,4 \text{ Mb/s}
 \end{aligned}$$

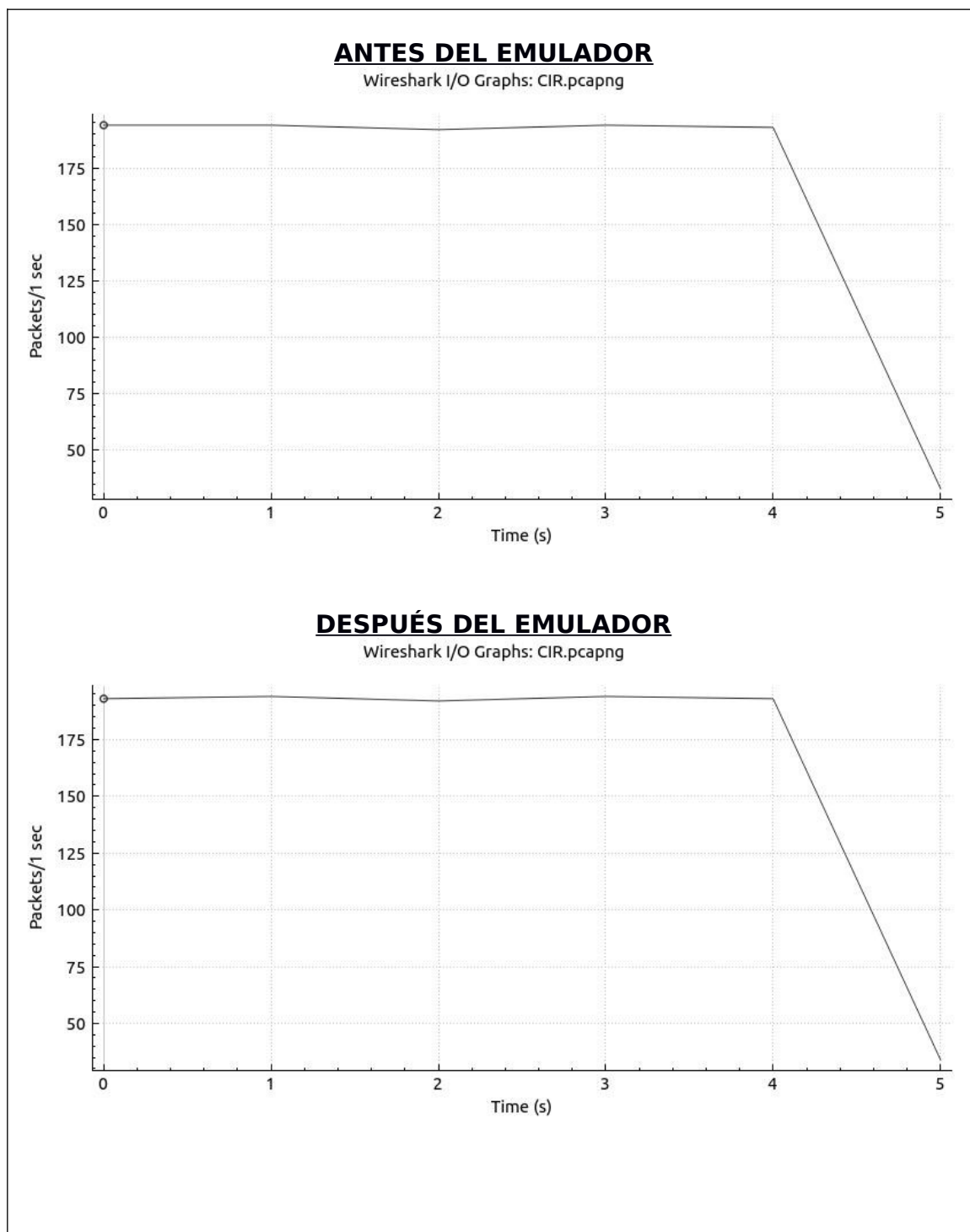
El MBS se coge de la lista de TAMANOS_CUBO y lo escogemos de la posición seis de esta lista que sería de 10000 bytes.

Escenario	CIR	MBS
6	2400000 bits/s	10000 bytes

- Utilice el clienteTren2.py que se facilita para esta práctica, que es equivalente al clienteTren2 solicitado en la práctica 2, en el que se puede limitar la tasa de transmisión en Kbit/s. Ejecute el emuladorTB.py utilizando el número de escenario que coincide con su número de pareja, transmita un tren de 1000 paquetes de 1460 bytes de datos de RTP, limitando la tasa al (1) CIR calculado previamente y (2) 10.000 Kbit/s.

- a. Capture el tráfico del emulador con Wireshark, cara a realizar medidas de ancho de banda y retardo, como ya realizó en la práctica 2. Represente con la función IOgraph de Wireshark el caudal **en bits/s** consumido por los 1000 paquetes a lo largo del tiempo, tanto a la entrada como a la salida del emulador (ambos flujos se pueden diferenciar fácilmente según su puerto de destino, excluyendo el tráfico ICMP). Explique los resultados obtenidos. (3 puntos).

Escenario 6, limitación al CIR

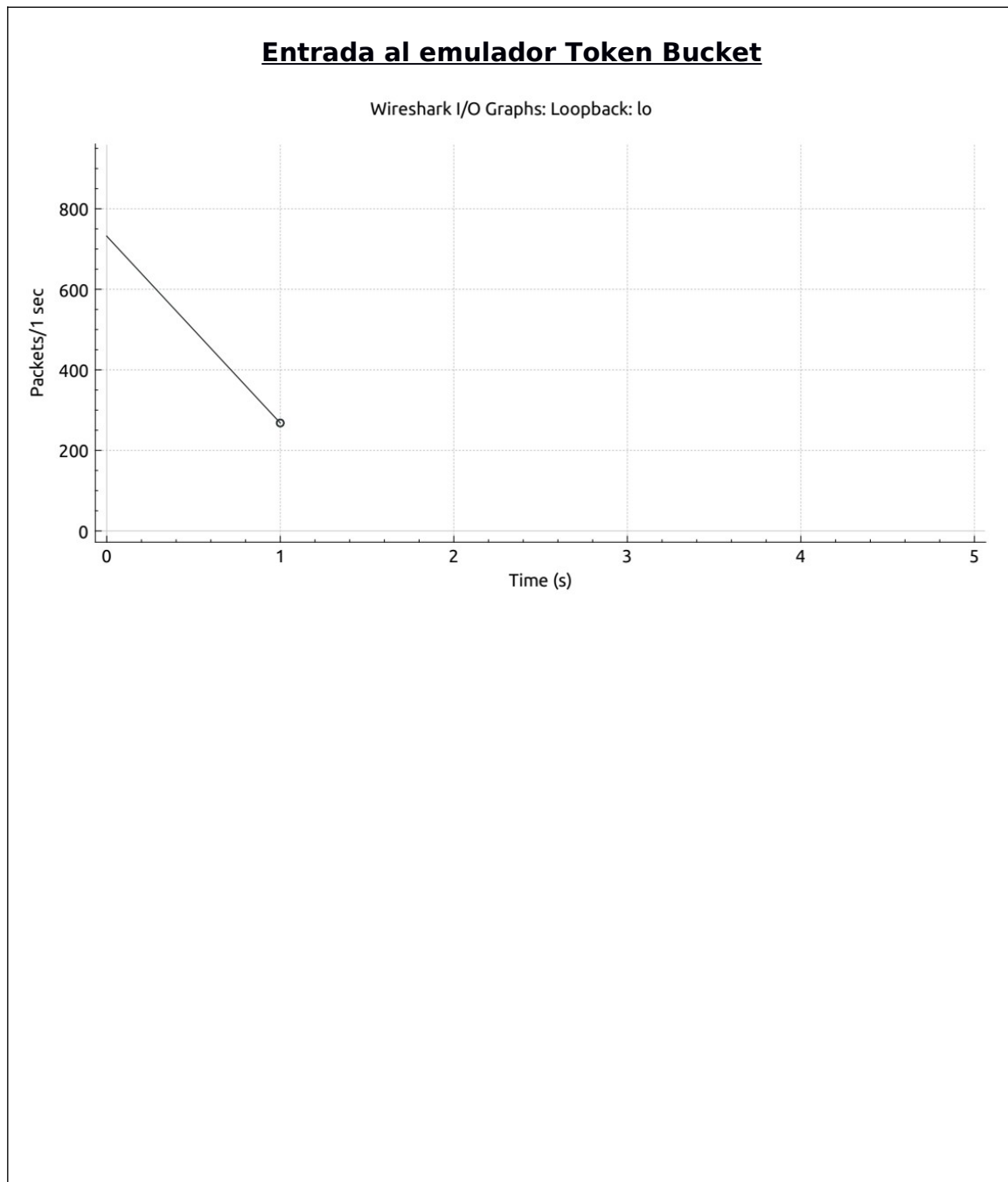


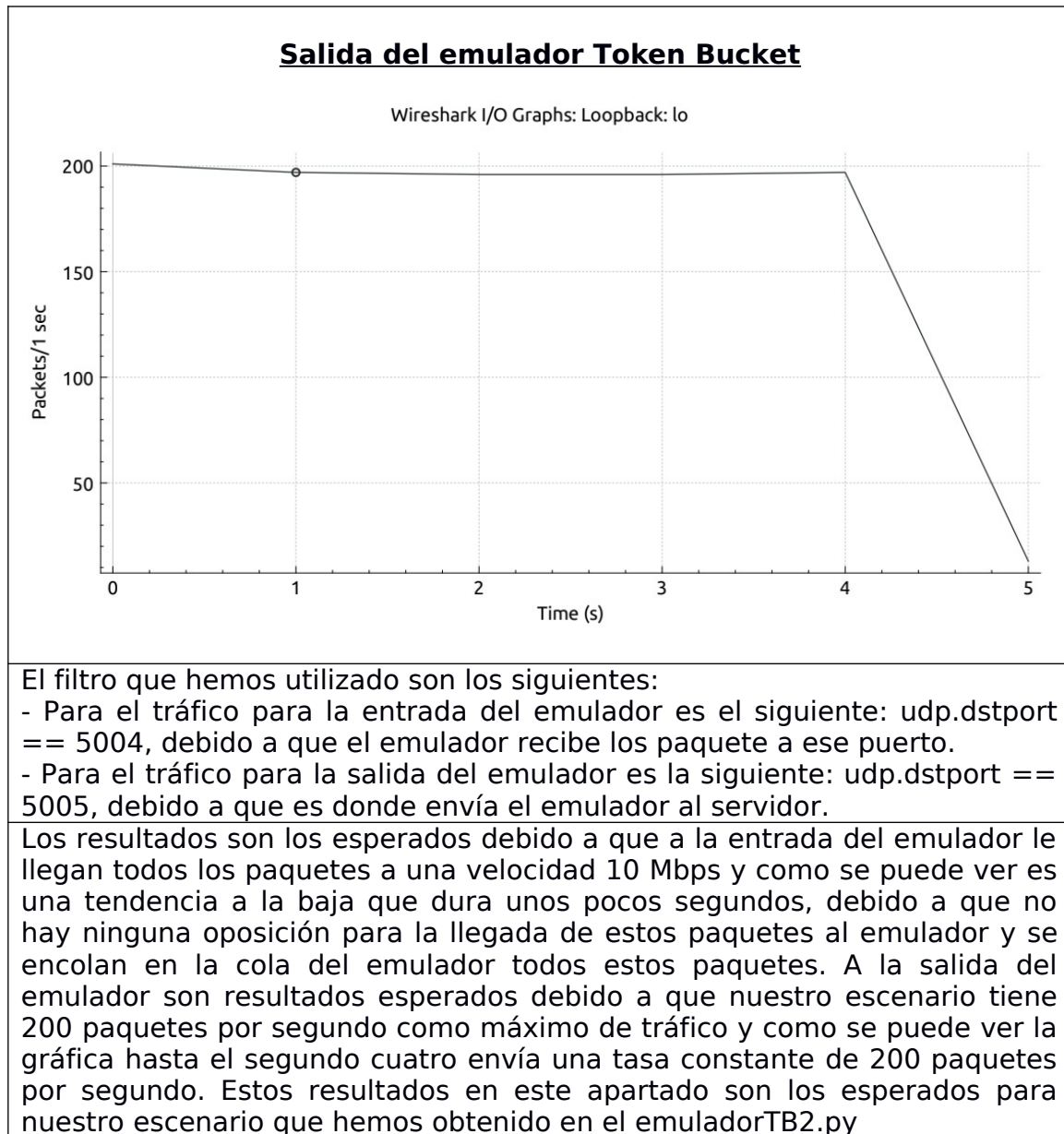
El filtro que hemos utilizado son los siguientes:

- Para el tráfico para la entrada del emulador es el siguiente: `udp.dstport == 5004`, debido a que el emulador recibe los paquete a ese puerto.
- Para el tráfico para la salida del emulador es la siguiente: `udp.dstport == 5005`, debido a que es donde envía el emulador al servidor.

Los resultados son los esperados, debido a que los paquetes no se encolan y los paquetes van llegando al emulador y van saliendo al mismo ritmo que van entrando en el Token Bucket, debido a que no excedemos la tasa máxima a la que puede enviar los paquetes.

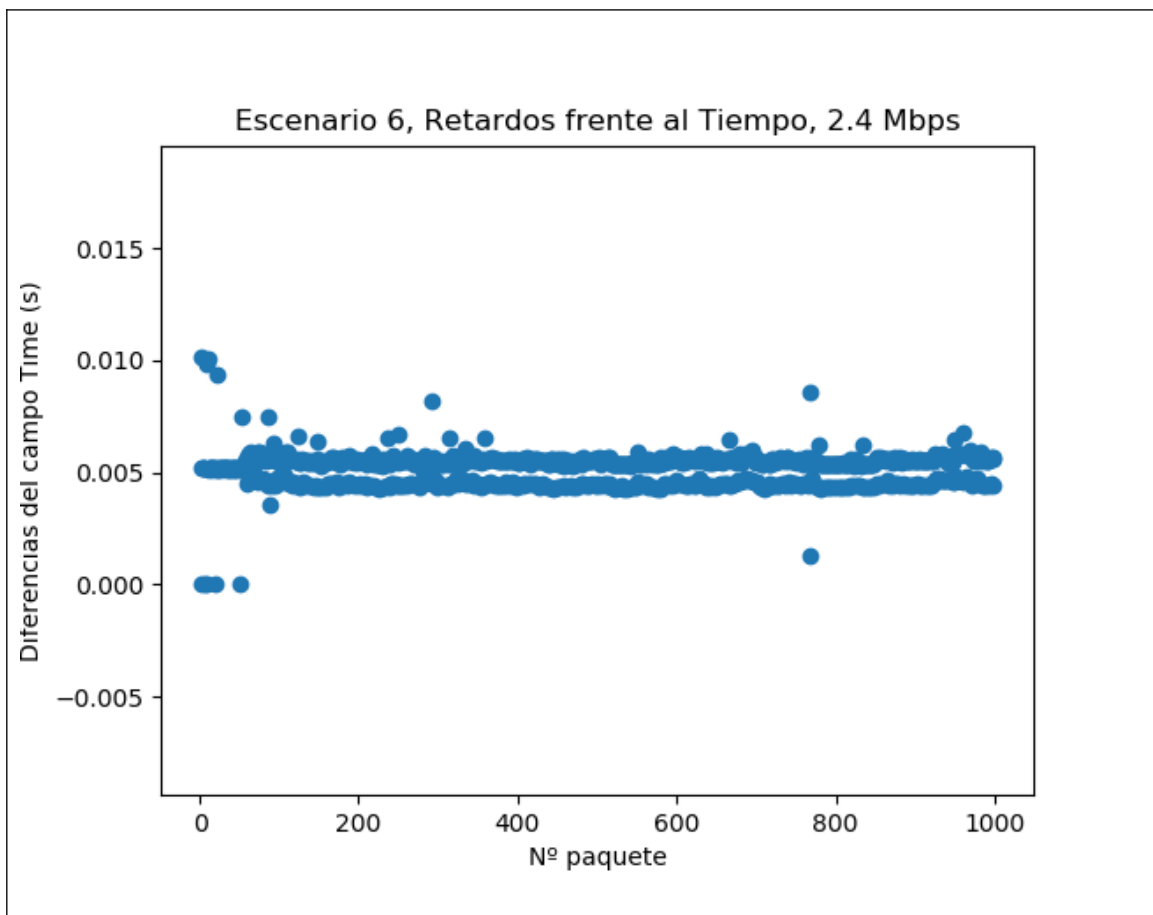
Escenario 6, limitación a 10 Mbps



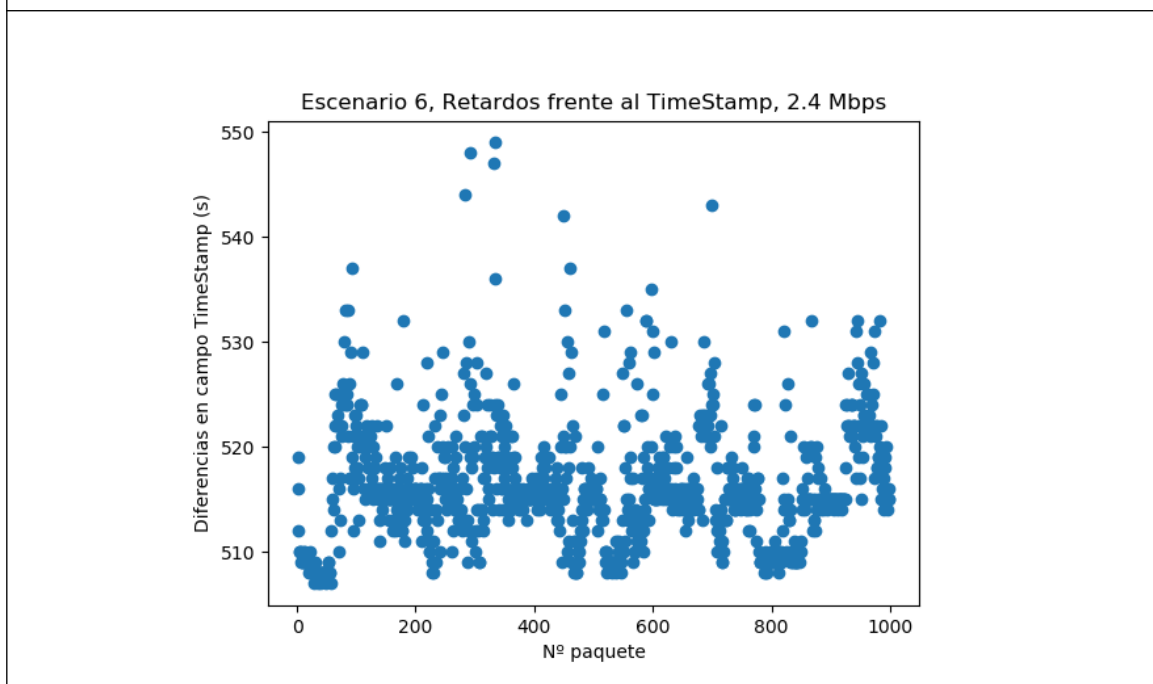


- b. Mida a partir de los datos capturados **a la salida del emulador** el retardo de los paquetes (entendido como la diferencia entre el tiempo de transmisión y el de recepción), extrayendo a una hoja de cálculo los parámetros necesarios. Represéntelos gráficamente frente al tiempo y explique a qué se debe la existencia o no de regiones bien diferenciadas. Realice una segunda gráfica relativa a los tiempos entre llegadas y explique igualmente los resultados. (2 puntos).

Escenario 6, limitación al CIR

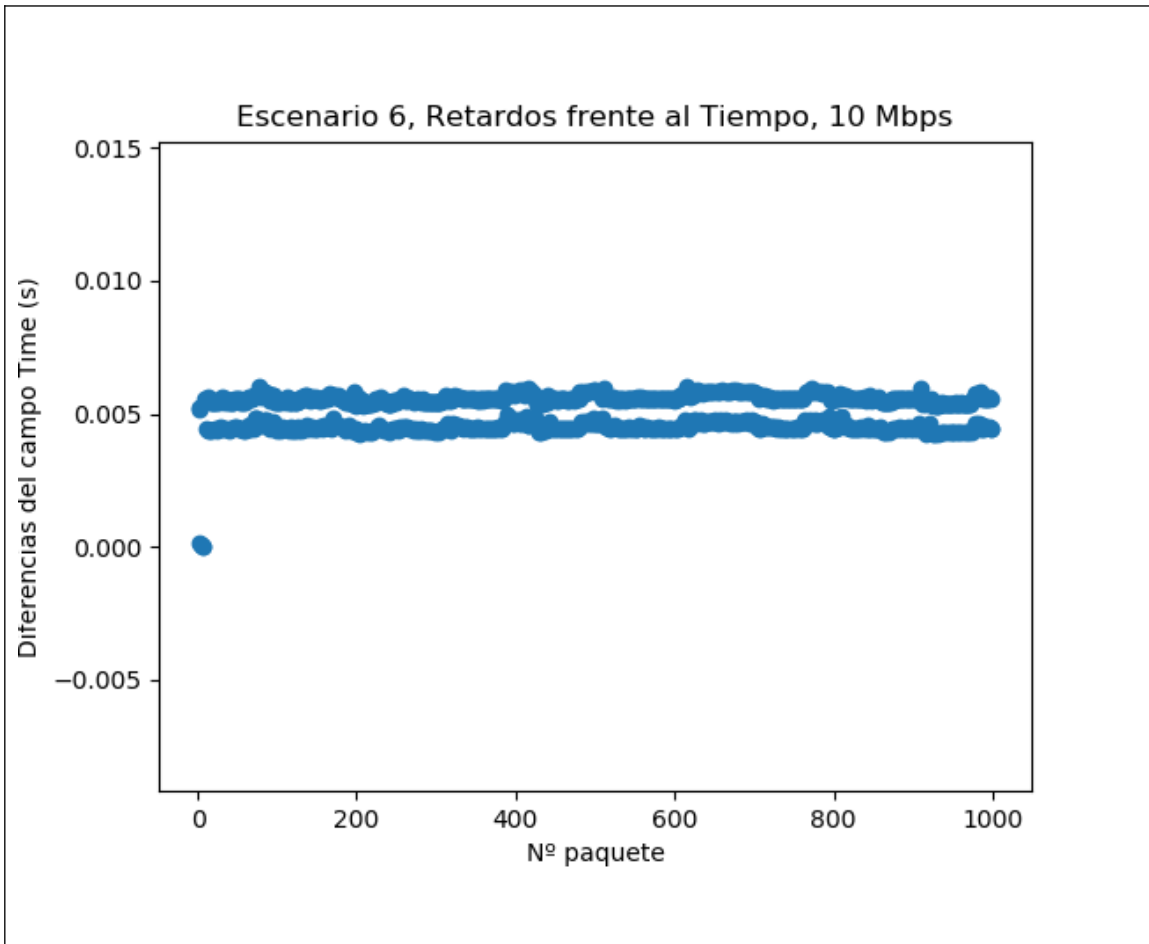


Como tenemos una velocidad que no satura el token bucket, podemos observar que los retardos son muy iguales para los 1000 paquetes que enviamos desde el emulador hasta el servidor. Es decir, observamos un retraso muy igual entre los distintos paquetes que enviamos

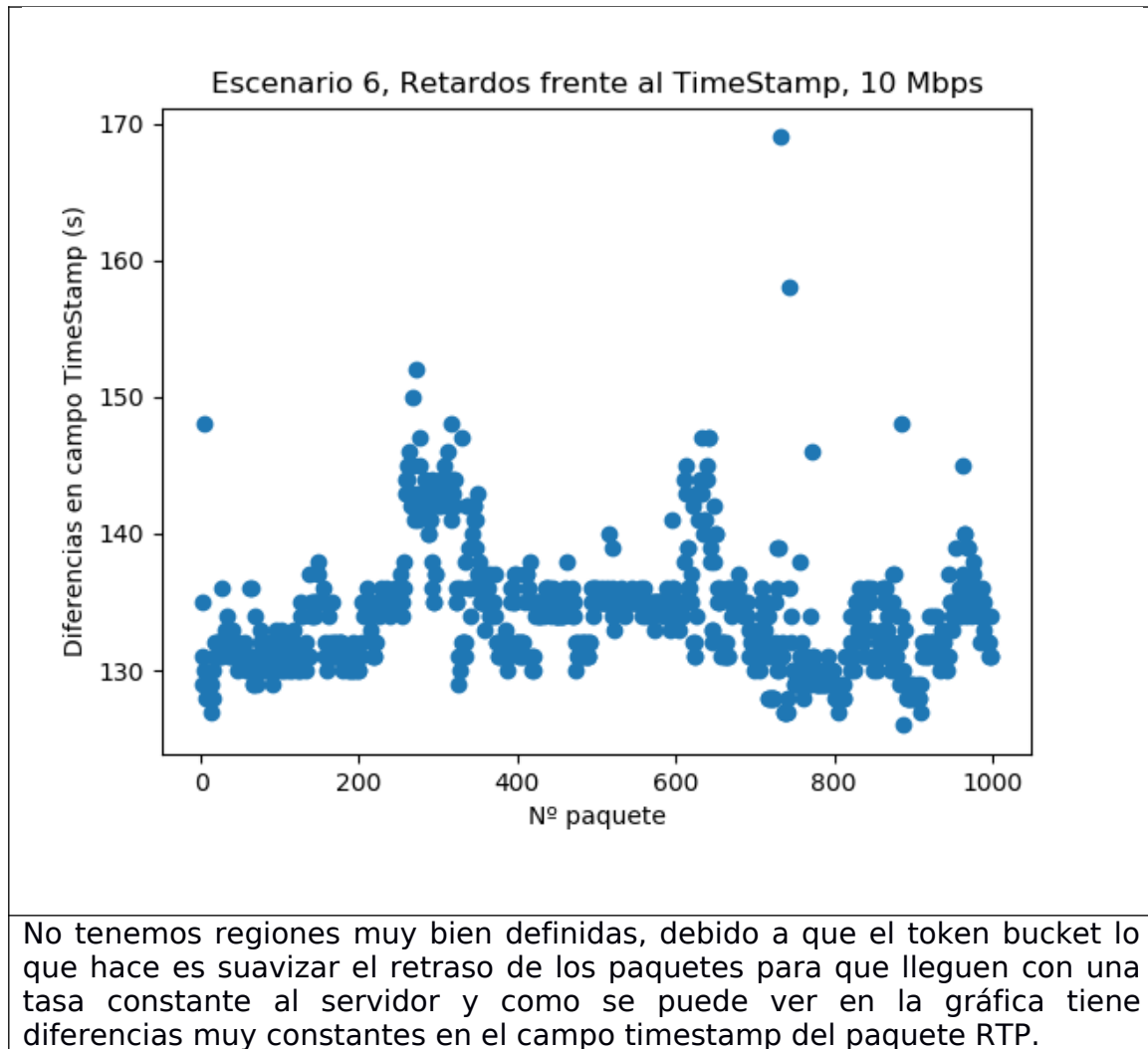


En esta gráfica no vemos unas zonas bien diferenciadas debido a que el token bucket no se satura. Lo único que podemos observar es cuando el token bucket llega a saturarse un poco en algunos paquetes.

Escenario 6, limitación a 10 Mbps



Como podemos observar si vemos dos regiones diferenciadas, la primera región se corresponde con los paquetes que no han llegado a tener saturación en el token bucket y luego vemos como ya el token bucket se satura y envía los paquetes con un retraso constante.



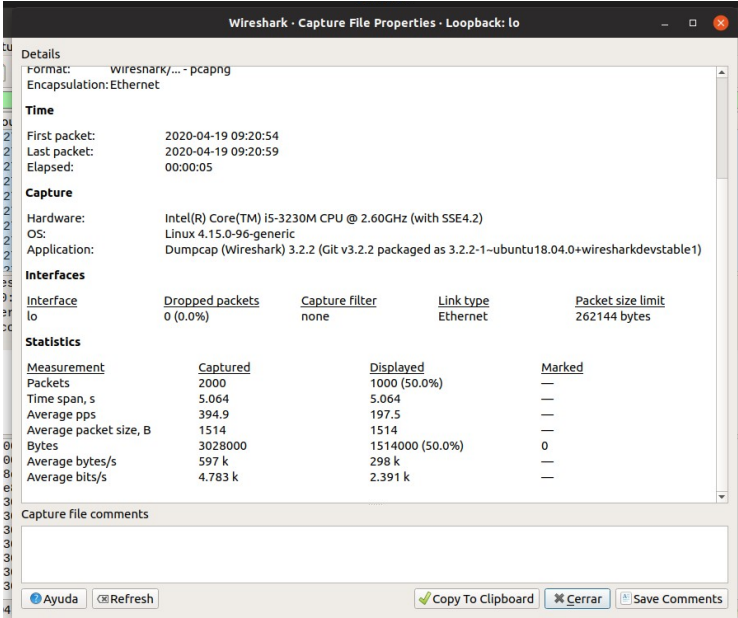
- c. Mida a partir de los datos capturados **a la salida del emulador**, el caudal efectivo en bits/s que finalmente se obtiene tras pasar por el *token bucket*. Utilice para ello la estimación de ancho de banda medio de la práctica 2. Compare el resultado obtenido con el previsto en el apartado 1. (2 puntos)

Escenario 6, limitación al CIR

Interface	Dropped packets	Capture filter	Link type	Packet size limit
lo	0 (0.0%)	none	Ethernet	262144 bytes
Statistics				
Measurement	Captured	Displayed	Marked	
Packets	2000	1000 (50.0%)	—	
Time span, s	5.173	5.168	—	
Average pps	386.6	193.5	—	
Average packet size, B	1514	1514	—	
Bytes	3028000	1514000 (50.0%)	0	
Average bytes/s	585 k	292 k	—	
Average bits/s	4.682 k	2.343 k	—	

Si es muy similar al previsto debido a que el emulador nos limita que la tasa máxima que enviemos los paquetes es de 2400 Kbits por segundo. Por lo tanto, los resultados son acordes con lo que hemos previsto con el Token Bucket.

Escenario 6, limitación a 10 Mbps



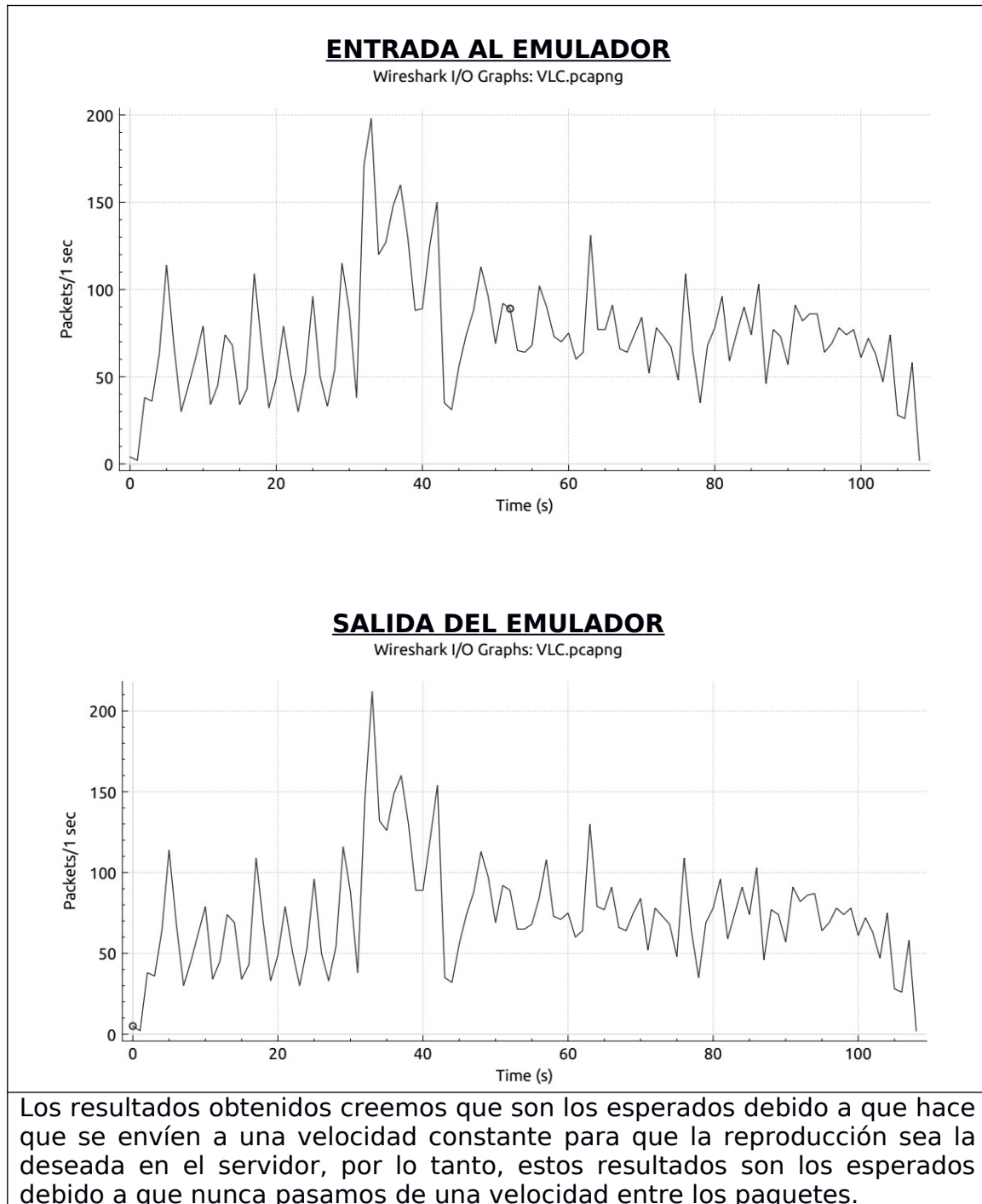
Interface	Dropped packets	Capture filter	Link type	Packet size limit
lo	0 (0.0%)	none	Ethernet	262144 bytes
Statistics				
Measurement	Captured	Displayed	Marked	
Packets	2000	1000 (50.0%)	—	
Time span, s	5.064	5.064	—	
Average pps	394.9	197.5	—	
Average packet size, B	1514	1514	—	
Bytes	3028000	1514000 (50.0%)	0	
Average bytes/s	597 k	298 k	—	
Average bits/s	4.783 k	2.391 k	—	

Si es muy similar al previsto debido a que el emulador nos limita que la tasa máxima que enviemos los paquetes es de 2400 Kbits por segundo. Por lo tanto, los resultados son acordes con lo que hemos previsto con el Token Bucket. Aquí podemos observar que da igual a la tasa que enviemos al emulador que siempre va a salir a la misma tasa del emulador a unos 2400 Kbits por segundo.

3. Utilice el mecanismo usado en la práctica 1 para estudiar la influencia del *token-bucket* sobre un flujo multimedia. Mediante dos VLC (emisor

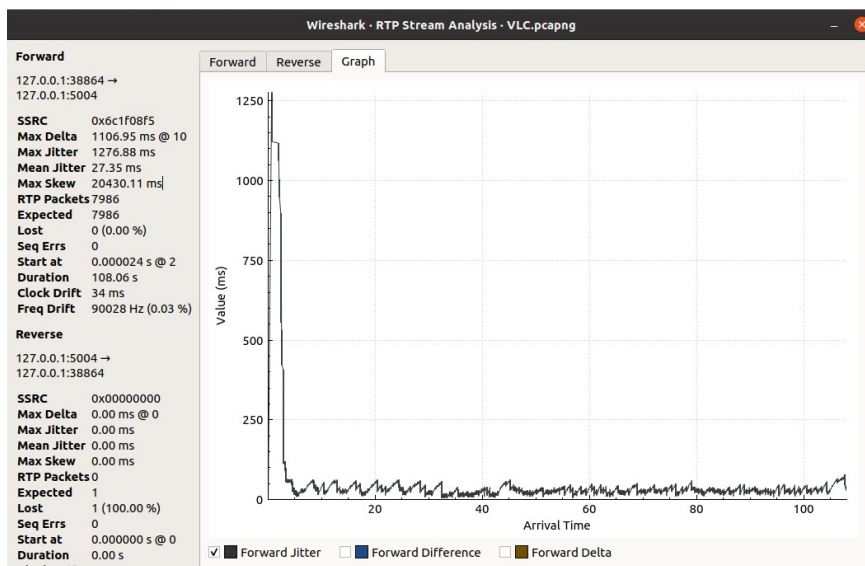
y receptor), envíe el vídeo proporcionado en dicha práctica a través del emuladorTB.py, para el escenario que coincide con su número de pareja. Recuerde que dicho vídeo es de tasa variable, con lo que será normal la presencia de picos en el mismo.

- a. Capture con el Wireshark el tráfico y represente con la función IOgraph el caudal consumido por el vídeo a lo largo del tiempo, tanto a la entrada como a la salida del emulador (ambos flujos se pueden diferenciar fácilmente según su puerto de destino). Explique la diferencia entre uno y otro flujo multimedia. (1 puntos)

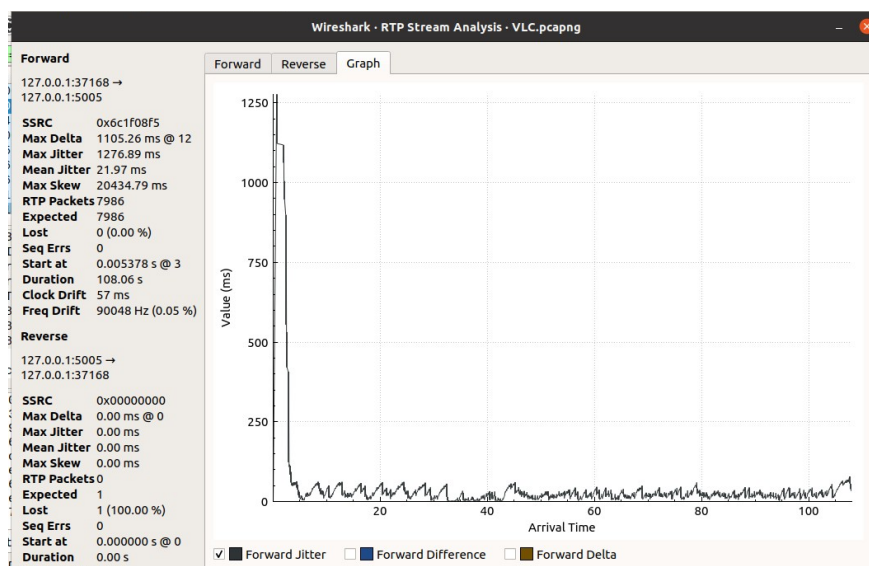


- b. Cuantifique experimentalmente qué retardo se añade en el caso peor a los paquetes en la traza capturada. Wireshark posee una funcionalidad de análisis de flujos RTP que puede serle de utilidad en este caso. (1 punto)

ENTRADA AL EMULADOR



SALIDA DEL EMULADOR



- c. Estudie desde un punto de vista subjetivo si aprecia alguna merma de calidad percibida (MOS), y si es posible y de qué manera corregir este problema en el receptor. (0,5 puntos)

¿Se aprecia alguna merma en la calidad percibida del vídeo en el receptor?

Solo se aprecia una leve perdida de paquetes al inicio de la reproducción del video, pero es tan leve que no afecta mucho a la reproducción del video que estamos emitiendo y por lo tanto, le pondríamos un 5 a la emisión del

VLC 1 al VLC 2 pasando por el emulador con el Token Bucket.

¿Se podría corregir este problema?

Se podría corregir este leve problema aumentando el buffer de recepción de paquetes del VLC 2.
--

3 Conclusiones

En esta práctica hemos aprendido a saber como funciona un Token Bucket y podamos ver como se utiliza en las redes multimedia, donde vemos todas las ventajas o inconvenientes que tiene esta técnica si la aplicamos a una red multimedia. En esta práctica la hemos utilizado para una red que envía trenes de paquetes, donde hemos observado que aunque saturemos al Token Bucket con los paquetes podemos llegar a tener alrededor de un 0% de perdidas de los paquetes que hemos enviado desde el cliente. También observamos como funciona entre dos VLC's que lo que hace es poder reproducir el video sin que haya ningún fallo a la hora de reproducirlo en el servidor y no saturar el enlace del servidor sin que el servidor llegue a desechar paquetes debido a que su cola de reproducción este llena.