Tarea 1: medidor ancho de banda UDP
Redes
Plazo de entrega: 24 de abril 2023

José M. Piquer

1 DESCRIPCIÓN

Su misión, en esta tarea, es generar un cliente que use un socket UDP que permita medir ancho de banda total para enviar datos entre él y un servidor. Este cliente lee un archivo de entrada y lo envía completo al servidor (quien lo recibe). Luego, el cliente avisa que terminó (como un fin de archivo) y el servidor envía hacia el cliente la cantidad de bytes que recibió. Una vez terminado, el cliente toma los bytes recibidos por el servidor y los divide por el tiempo transcurrido desde el inicio hasta la recepción del resultado desde el servidor (usar la función `time.time()`), entregando un número de Megabytes por segundo (1 Megabyte = 1024*1024 bytes).

Siendo un socket UDP, habrá pérdida de paquetes, y se notará por que el servidor a veces informará menos bytes que los enviados. En algunos casos, puede fallar el protocolo también.

Un tema importante en los sockets UDP es qué tamaño de buffer usar cuando uno invoca `send()` y `recv()`. La documentación es un tanto vaga, hay un tamaño máximo absoluto de 65.535 bytes (un short integer si recuerdan su curso de Software de Sistemas), pero algunos sistemas limitan en el Sistema Operativo ese tamaño a números mucho más bajos.

En teoría es mejor usar paquetes grandes, envían más información por menos costo, y debieran ser más eficientes. Pero el tamaño puede afectar también la probabilidad de pérdida, y puede no ser tan bueno.

Esta tarea busca probar distintos valores para los paquetes enviados y recibidos, y entonces el cliente envía una propuesta de tamaño máximo y el servidor le responde con la suya. El valor a ser utilizado **siempre** debe ser la respuesta del servidor.

Para poder probar eso, el archivo de entrada debe leerse en bloques de bytes, del tamaño del paquete máximo, para ser enviados. En Python, pueden usar archivos binarios (de bytes) para eso y la función `read()`.

Hay un servidor corriendo en anakena.dcc.uchile.cl puerto 1818 UDP con este protocolo. Uds tienen que implementar el cliente.

El cliente puede invocarse como:

```
% ./bwc.py 5000 /etc/services anakena.dcc.uchile.cl 1818
propuse paquete: 5000
recibo paquete: 5000
bytes enviados=677977, bytes recibidos=670000, time=0.09839272499084473, bw=6.5713115687
```

los parámetros son: tamaño de paquete a proponer al servidor, archivo de entrada, nombre del servidor y número de puerto UDP a utilizar.

2 PROTOCOLO

1. Cliente: envía 'Cxxxx'
2. Servidor: responde 'Cyyyy'
3. Cliente: envía paquetes con todo el contenido de filein, siempre comienzan con 'D' y son de tamaño máximo yyyy bytes más la letra 'D' del inicio (o sea, resultan paquetes UDP de yyyy+1 máximo):

   'Dkkkkkkkkkkk'
   'Djjjjjjjjjjjj'
   ....

4. Cliente: al terminarse el archivo envía 'E' en un paquete sólo y con eso marca el fin de la transmisión
5. Servidor: envía 'Ezzz..zz' donde zzz..zz es un string (largo variable) que contiene el número de bytes recibidos por el servidor en total

En Python, estas secuencias son *bytearray*. Los números xxxx yyyy son los tamaños de datos máximo, codificados como *bytearray* de números como 4 caracteres. Por ejemplo, si el cliente quiere usar un tamaño de 900 bytes, enviará:

```
C0900
```

(pueden ver que el tamaño máximo de paquete que se puede usar es 9999)

Como son paquetes UDP en el socket, no necesitan fin de línea después.

El protocolo resiste archivos binarios (ejecutables, imágenes) como entrada y debiera funcionar bien. Esto quiere decir que lo que viene después de la 'D' inicial Uds no pueden suponer que es un string, sino que es una secuencia de bytes binarios (que es lo que en Python

es un bytearray). Esto debe ser así al enviarlo y al recibirlo, tanto del socket como del archivo.

Como es UDP, pueden perderse paquetes en la red. Por lo tanto, cuando Uds esperan recibir paquetes desde el servidor (respuesta a 'C' y a 'E'), siempre deben implementar un timeout de lectura desde el socket. Después de un timeout de 10 segundos, reintentan el envío 5 veces y si todas fallan, abortan con un mensaje de error. Esto es equivalente a decir que pasaron 10 segundos, 5 veces, sin recibir nada desde el socket. Los mensajes con el contenido del archivo ('D') no reciben respuesta, por lo que no requieren re-intentar.

Al abortar, no deben medir el ancho de banda, sólo reportar el error.

3 ENTREGABLES

Básicamente entregar el archivo con el cliente que implementa el protocolo.

En un archivo aparte responder las preguntas siguientes (digamos, unos 5.000 caracteres máximo por pregunta):

1. Genere algunos experimentos con diversos tamaños de paquete y haga una recomendación de tamaño de paquete a utilizar en base a sus resultados (entregar los experimentos realizados y sus resultados junto con las conclusiones). Incluya los valores 1 y 9999 en sus pruebas. Grafique sus resultados.

2. Discuta si la medición de ancho de banda obtenida es "válida": al haber pérdidas, ¿estamos midiendo mal? Los resultados obtenidos: ¿son coherentes? ¿se parecen a la realidad? Argumente con sus propias mediciones realizadas en el punto anterior. En particular, mire el caso de retransmisión de 'C' o 'E': ¿afecta la medición?

3. Si el tamaño del archivo de entrada es exactamente igual a los bytes reportados por el servidor ¿podemos ahí estar seguros que la medición es válida?

4. ¿El servidor no debería hacer lo mismo y repetir el 'E' hasta que el cliente lo reciba? ¿O esto generaría posibilidades de error? ¿podría afectar la medición?

5. Uno podría no esperar el reporte del servidor y simplemente enviar el archivo y medir cuánto demora en escribirse en el socket y usar ese tiempo para el cálculo del ancho de banda. Explique por qué eso está mal medido (les prometo que está mal medido).

6. Es una complicación agregarle una 'D' a cada paquete de datos enviado al servidor. ¿Por qué no enviamos todo el archivo no

más, y luego un paquete 'E' solo? Así, sería más fácil y el servidor detecta el fin de la transmisión cuando llega un paquete con una 'E'. ¿Funcionaría?

4 STRINGS Y BYTEARRAYS EN SOCKETS

Una confusión clásica en los sockets en Python es la diferencia entre enviar un string y/o un bytearray. Partamos por los strings, que Uds conocen mejor: los strings no son simplemente arreglos de bytes (alguna vez lo fueron, pero hoy pueden contener hasta alfabetos asiáticos y árabes), son codificaciones en un *encoding* particular. Los sockets no soportan strings, es decir, Uds no pueden llegar y enviar/recibir un string por el socket, deben convertirlo a un bytearray, que es una colección de bytes binarios primitivos, no se interpretan. La forma de convertir un string a un bytearray es aplicando la función *encode* y un bytearray a un string, con la función *decode*. Ojo que si se aplican a cualquier cosa, pueden fallar, particularmente decode falla si uno le pasa cualquier cosa en el bytearray. Entonces, si quiero enviar/recibir el string 'niño' hago:

```
enviador: s = 'niño'
    sock.send(s.encode('UTF-8'))    # UTF-8 es el encoding clásico hoy
receptor: s = sock.recv().decode()      # recibe s == 'niño'
        print(s)
```

En cambio, si leo un archivo cualquiera y quiero enviarlo, no sé si hay strings o no dentro, entonces mejor es no transformarlo y siempre enviar bytes puros:

```
data = fd.read(MAXBYTES)
sock.send(data)
```

En esta tarea hay una mezcla de ambas cosas: los 'comandos' ('C', 'E', etc) son letras, que deben ser transformadas a bytearrays antes de enviar al socket. En Python, ord('C') transforma la letra a bytearray, y chr(b) transforma un bytearray en letra. El string con el total de bytes recibidos desde el servidor es un string codificado a bytes que deben decodificar. Pero todo el resto (lo que viene después de las 'D' que uds envían en cada paquete de datos) es un bytearray puro que Uds obtuvieron desde el archivo, no deben tratar de codificarlo a string.