Tarea 1: medidor ancho de banda UDP confiable usando Stop-and-Wait
Redes
Plazo de entrega: 25 de septiembre 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 recibir datos desde un servidor, aplicando un protocolo de corrección de errores. Este cliente le pide al servidor que le envía una cantidad de bytes usando un protocolo corrección de errores tipo Stop-and-Wait. Luego, el servidor avisa que terminó (con un fin de archivo) y el cliente debe reportar la cantidad de bytes que recibió y el ancho de banda logrado. Para esto, toma los bytes recibidos desde el servidor y los divide por el tiempo transcurrido desde el inicio hasta la recepción del fin de archivo 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, por lo que su cliente debe estar preparado para fallas en la secuencia y corregir.

Un tema importante en los sockets UDP es qué tamaño de buffer usar cuando uno invoca `send()` y `recv()`. 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.

De la misma forma, también importan otros parámetros: el procentaje de pérdida y el valor del timeout en el servidor (que debe aproximar el máximo RTT: tiempo que toma ir y volver en la red entre cliente y servidor). El cliente que deben escribir recibe el tamaño de paquete a proponer, la cantidad de bytes a pedir al servidor, el valor del timeout (en ms) a pedir al servidor, la pérdida (en porcentaje), archivo a generar como salida, servidor, puerto:

```
./bwc-sw.py pack_sz nbytes timeout loss fileout host port
```

Para implementar la probabilidad de pérdida, usen estas funciones:

```
# Envía un paquete con loss_rate porcentaje de pérdida
# si loss_rate = 5, implica un 5% de pérdida
def send_loss(s, data):
    global loss_rate

    if random.random() * 100 > loss_rate:
        s.send(data)
    else:
        print("[send_loss]")

# Recibe un paquete con loss_rate porcentaje de pérdida
# Si decide perderlo, vuelve al recv y no retorna aun
# Retorna None si hay timeout o error
def recv_loss(s, size):
    global loss_rate

    try:
        while True:
            data = s.recv(size)
            if random.random() * 100 <= loss_rate:
                print("[recv_loss]")
            else:
                break
    except socket.timeout:
        print('timeout', file=sys.stderr)
        data = None
    except socket.error:
        print('recv err', file=sys.stderr)
        data = None

    return data
```

En pruebas reales, con anakena, habrán además pérdidas reales, por lo que siempre el porcentaje real será un poco mayor.

Hay un servidor corriendo en anakena.dcc.uchile.cl puerto 1819 UDP con este protocolo.

Pueden usar `sock.settimeout(val)` para programar el timeout del próximo `sock.recv()` que hagan. Si ponen `val=0` el socket se vuelve no-bloqueante y pueden preguntar si hay datos sin bloquearse. En este protocolo, lo usual es que siempre quieran bloquearse un rato esperando datos desde el servidor. Es él que tiene que implementar los timeouts inteligentes de retransmisión, Uds sólo deben generar los ACKs correspondientes. Si su lectura desde el servidor les da timeout, quiere decir que todo falló y deben terminar con un error, no tienen como corregir eso. Implementen un timeout para morir de 5 segundos.

El cliente debe invocarse como:

```
% ./bwc-sw.py 5000 100000000 20 5 out anakena.dcc.uchile.cl 1819
propuse paquete: 5000
recibo paquete: 5000
recibiendo 100000000 nbytes
bytes recibidos=100000000, time=1246.566123008728, bw=0.07650410987460893 MBytes/s
errores=51
```

Los parámetros son: paquetes de 5.000 bytes (más el header), 100 millones de bytes, 20 ms de timeout, 5 % de pérdida y archivo de salida out. El servidor genera siempre bytes con el caracter 'a', por lo que el archivo out debe quedar de tamaño 100 millones y lleno de 'a'.

2. Protocolo

1. Cliente: envía 'Cxxxxtttt'
2. Servidor: responde 'Cyyyy'
3. Cliente: envía 'Nnbytes'
4. Servidor: envía paquetes hasta completar 'nbytes' bytes en total. Siempre comienzan con 'D' y dos bytes que codifican el número de secuencia (entre 00 y 99) y son de tamaño máximo yyyy bytes más el header utilizar (o sea, resultan paquetes UDP de yyyy+3 como máximo tamaño): 'D00aaaaaaaaaaaaaaaaaaaaaaaa'
5. Cliente: envía ACK (acknowledgements) con el número de secuencia del último paquete recibido bien: 'A00'
6. Servidor: 'D01aaaaaaaaaaaaaaaaaaaaaaaa'
7. Cliente: 'A01'

8. ...

9. Servidor: al completarse el envío, termina con un paquete 'E' con su número de secuencia en un paquete sólo y con eso marca el fin de la transmisión: 'E81'

10. Cliente envía último ACK: 'A81'

En Python, estas secuencias son *bytearray*. Los números xxxx yyyy son los tamaños de datos máximo, codificados como *bytearray* de números como 4 caracteres. Igual el timeout tttt medido en milisegundos. Los números de secuencia son números de dos caracteres. En cambio nbytes (del paquete N) es un string de tamaño variable del número de bytes solicitados:

```
N93000000
```

Estoy pidiendo 93 millones de bytes.

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

Como es UDP, pueden perderse paquetes en la red. Para corregir las pérdidas, el servidor espera siempre el ACK del último número de secuencia enviado antes de seguir. Si no lo recibe en el tiempo que Uds le enviaron al comienzo del protocolo, retransmite el número de secuencia pendiente. Este protocolo se conoce como Stop-and-Wait y lo estudiaremos en detalle en el capítulo de protocolos de transporte.

Los números de secuencia van de 00 a 99, y luego se reciclan, después de 99 le sigue el 00 otra vez. El paquete 'E' de final siempre lleva el número de secuencia siguiente al último dato enviado.

Uds siempre deben esperar el número de secuencia que les toca, y cualquier paquete con otro número de secuencia deben descartarlo, pero siempre retransmitir el último ACK que habían enviado, por si acaso ese ACK se hubiese perdido. Ese caso se considera un error, y se les pide que cuenten cuántos de estos errores ocurren en la transmisión.

Para los paquetes iniciales, usen un timeout de envío de 3s y retransmitan 100 veces antes de morir con error. En el resto del protocolo, cuando están recibiendo los datos, mantengan un timeout de 3s y aborten el programa si no reciben nada en ese tiempo. Para el timeout de recepción usen la función: *sock.settimeout()* en el socket UDP.

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

Al recibir el paquete 'E', envían su ACK, y terminan reportando el ancho de banda logrado y la cantidad de errores detectados.

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, timeout y pérdidas y haga una recomendación de valores a utilizar para las distintas pérdidas. Grafique sus resultados.

2. Discuta si la medición de ancho de banda obtenida es "válida": al haber retransmisiones, ¿estamos midiendo mal?

3. Revise si ocurre desorden de paquetes: ¿recibe siempre los ACKs en orden? Si no es así, ¿el protocolo debiera funcionar igual? ¿En algún caso extremo de desorden podría fallar?

4. En este caso usamos números de secuencia de 0 a 99, pero el protocolo stop-and-wait normalmente se implementa con sólo 0 y 1. ¿Aporta algo el tener más números de secuencia para defendernos de un eventual desorden en los paquetes?

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 recibo bytes y quiero escribirlos en un archivo cualquiera, no sé si hay strings o no dentro, entonces mejor es no transformarlo y siempre usar bytes puros:

```
enviador:
    data = fdin.read(MAXDATA)
    sock-send(data)

receptor:
    data = sock.recv()
    fdout.write(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 pedidos al servidor es un string codificado a bytes que deben decodificar. Pero todo el resto (lo que viene después de las 'D' que uds reciben en cada paquete de datos) es un bytearray puro que Uds recibieron desde el servidor, no deben tratar de codificarlo a string. Deben escribirlo en el archivo de salida directamente (sin el header) con la función *fdout.write()*