Tarea 1: Cliente eco TCP para medir performance
Redes
Plazo de entrega: 31 de marzo 2025

José M. Piquer

1. Descripción

Su misión, en esta tarea, es modificar el cliente TCP con 2 threads (`client_echo3.py`) para usarlo como un medidor de ancho de banda. El servidor que usaremos para la medición es: `server_echo4.py` (se provee como material docente).

Ahora el cliente usa un archivo de entrada y otro de salida como archivos binarios (así pueden probar con cualquier tipo de archivo), y recibe como argumento el tamaño de las lecturas y escrituras que se harán (tanto de/desde el socket como de/desde los archivos).

También deben modificar el código del cliente para detectar bien el término de la ejecución: el thread enviador debe contar los bytes enviados hasta el EOF, y el receptor debe esperar todos esos bytes hasta terminar, y ahí termina el programa.

El cliente que deben escribir recibe el tamaño de lectura/escritura, el archivo de entrada, el de salida, el servidor y el puerto TCP.

```
./client_bw.py size IN OUT host port
```

Para medir el tiempo de ejecución pueden usar el comando `time`.

El servidor para las pruebas pueden correrlo localmente en local (usar localhost o 127.0.0.1 como "host"). Dejaremos uno corriendo en anakena también, en el puerto 1818 TCP.

Un ejemplo de medición sería (suponiendo que tengo un servidor corriendo en anakena en el puerto 1818):

```
% time ./client_bw.py 10 anakena.dcc.uchile.cl 1818 < /etc/services > OUT
0.79 real         0.41 user         0.61 sys
```

Para enviar/leer paquetes binarios del socket usen send() y recv() directamente, sin pasar por encode()/decode(). El arreglo de bytes que usan es un bytearray en el concepto de Python (al ser binarios, no son strings).

2. Mediciones

El cliente sirve para medir eficiencia. Lo usaremos para ver cuánto ancho de banda logramos obtener y también probar cuánto afecta el largo de las lecturas/escrituras en la eficiencia.

Para probar en localhost necesitan archivos realmente grandes, tipo 0.5 o 1 Gbytes (un valor que demore tipo 5 segundos). Para probar con anakena, basta un archivo tipo 500 Kbytes.

Diseñen un experimento que pruebe con:

- 1 archivo grande y distintos tamaños de lectura/escritura
- 100 archivos chicos en paralelo (que sumen los mismos bytes que el archivo anterior), y distintos tamaños de lectura/escritura

3. Entregables

Básicamente entregar el archivo con el cliente que implementa el protocolo, los scripts shell que utilizó para los experimentos y un archivo con los resultados medidos, tanto para localhost como para anakena.

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 sólo usaremos bytearrays.