Tarea 1: Cliente UDP de eco eficiente para archivos
Redes
Plazo de entrega: 22 de abril 2024

José M. Piquer

1. Descripción

Su misión, en esta tarea, es modificar el cliente con threads (`client_echo3_udp.py`) para tratar de que funcione bien al darle un archivo grande de entrada. Usaremos el mismo servidor de eco (sólo haremos que pueda recibir paquetes UDP más grandes) y modificaremos el cliente.

Ahora el cliente recibe un archivo de entrada y uno de salida como argumentos, y los dos threads del cliente se sincronizarán para asegurarse que el emisor no vaya mucho más rápido que el receptor.

La idea es que la mayoría de las pérdidas de datos se producen cuando el emisor va demasiado rápido y no da tiempo a que el paquete de respuesta llegue. Como sabemos que cada paquete emitido debe volver (salvo que se pierda), podemos contar cuánto hemos transmitido y restarle cuánto hemos recibido y así saber qué tan lejos va el emisor del receptor. Definiremos la distancia entre emisor y receptor como la diferencia entre paquetes emitidos y recibidos.

Usaremos entonces dos argumentos más: el tamaño de paquete que se usará y la distancia máxima que se permite entre emisor y receptor (medida en paquetes).

Para mantener los dos threads sincronizados, se requiere una variable compartida que mida la distancia y cuando llega al valor máximo, el emisor se bloquea esperando que disminuya. El receptor debe avisarle al emisor cuando eso ocurra. Como puede haber pérdidas de paquetes (y también duplicados, aunque es menos frecuente) el emisor necesita un timeout en la espera, de modo que si no llegan nunca esos paquetes, igual continúe transmitiendo. Se les pide implementar un timeout de 0.8s.

Al terminar el envío, deben esperar a que el receptor reciba todos los

paquetes pendientes (u ocurra un timeout), lo que es equivalente a esperar a que la distancia sea cero. Si ocurre un timeout, deben terminar no más, aunque la distancia sea positiva (ya no llegarán más paquetes).

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.

El cliente que deben escribir recibe el tamaño de paquete a proponer, la distancia máxima, archivo de entrada, archivo de salida, servidor, puerto. Al terminar escribe en la salida estándar un resumen de lo medido:

```
./copy_client.py pack_sz dist filein fileout host port
```

Hay un servidor de eco corriendo en anakena.dcc.uchile.cl puerto 1818 UDP. Pero, en el foro publicaremos el código del servidor para que puedan correrlo localmente para pruebas con localhost también.

Un ejemplo de ejecución sería:

```
% ./copy_client.py 1000 10 /etc/services OUT anakena.dcc.uchile.cl 1818
Usando: pack: 1000, maxdist: 10
Tiempo total: 5.32519793510437s
```

Escriban ese mismo tipo de salida en sus clientes, para que podamos corregirlos correctamente.

Para que el archivo de entrada pueda ser binario (no sólo texto), usen la opción 'b' para open() en python y luego lean/escriban con read()/write().

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.

2. 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 distancias (con anakena). Mire el tiempo total que toma la transferencia y revise qué tan distinto es el archivo de entrada al de salida. Recomiende los mejores valores para su caso según sus resultados.

2. Si hay paquetes duplicados, ¿qué pasa con la distancia?

3. Si hay pérdidas, ¿qué pasa con la distancia?

4. Su programa, ¿mantiene bien medida la distancia? ¿puede perderse entre pérdidas y duplicaciones?

5. ¿Su programa funciona si lo invocamos con una distancia cero? ¿Por qué? ¿Debería funcionar o no?

3. 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.