Tarea 2: Cliente UDP con corrección de errores de eco eficiente para archivos
Redes
Plazo de entrega: 27 de mayo 2024

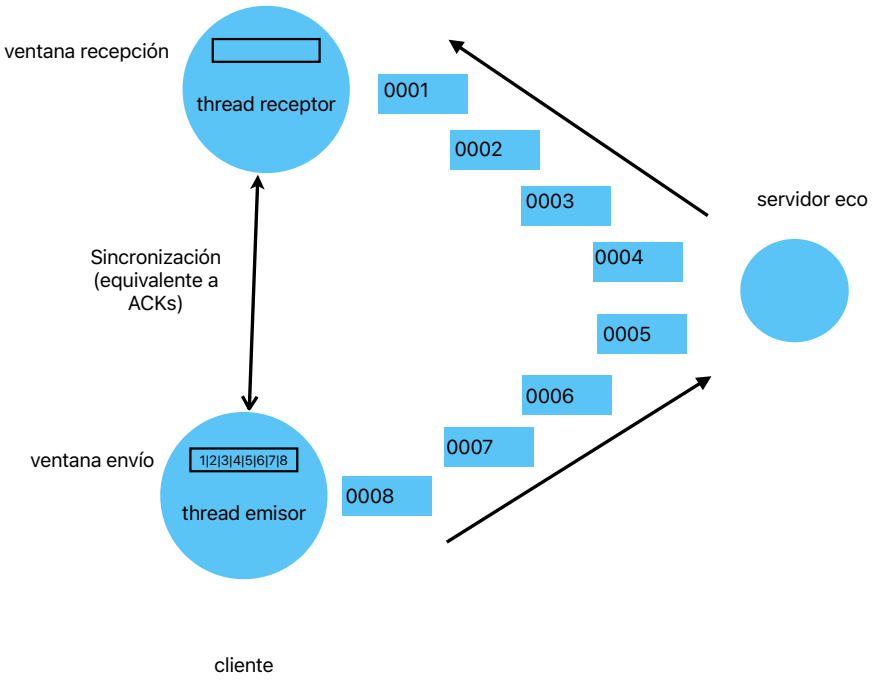
José M. Piquer

1. Descripción

Su misión, en esta tarea, es modificar el cliente con threads de la tarea 1 para hacer que funcione corrigiendo los errores de transmisión al darle un archivo grande de entrada. Usaremos el mismo servidor de eco de la T1 y modificaremos el cliente.

Ahora los dos threads del cliente se sincronizarán como si fueran un emisor y un receptor en un protocolo clásico de ventana corredera. La idea es que implementen el mejor protocolo posible: si hacen un Stop-and-Wait y no se pierde nada tienen un 4.0, si hacen un Go-Back-N bueno tienen un 5.5, si hacen un Selective-Repeat bueno tienen un 7.0.

La estructura de la solución se parece a la de la Tarea 1, y la distancia máxima que mantenían en esa tarea es el tamaño de la ventana. Deben mantener una ventana de envío en el thread emisor y una de recepción en el thread receptor. La única diferencia importante con un protocolo normal, es que no usarán ACKs, ya que pueden compartir memoria entre emisor y receptor, por lo que el receptor le puede avisar al emisor qué paquetes ha recibido e incluso los threads pueden mirar la ventana del otro (con mutex y esas cosas eso sí). Ver diagrama.


ventana recepción
thread receptor
0001
0002
0003
0004
0005
0006
0007
0008
servidor eco
Sincronización
(equivalente a
ACKs)
ventana envío
1|2|3|4|5|6|7|8
thread emisor
cliente

Como siempre: Stop-and-Wait usa ventanas de tamaño 1, Go-Back-N usa una ventana de envío de tamaño N y una de recepción de tamaño 1, Selective-Repeat usa ambas ventanas de tamaño N.

El tamaño de la ventana (N) es un parámetro que deben recibir (salvo en el caso de Stop-and-Wait) y va entre 1 y 4999.

Para poder implementar eso, se les pide usar números de secuencia en los paquetes, con 4 caracteres que representan el número de secuencia (0000-9999). Todos los paquetes que envíen deben venir con esos cuatro caracteres, y así los reciben igual. Cuando definimos el tamaño máximo de paquete a usar, ahora debe incluir esos 4 caracteres, por lo que deben leer paquetes de datos desde el archivo de entrada que sean (tamaño máximo de paquete)-4.

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. Por eso, es importante en esta tarea permitir que los dos threads ejecuten en paralelo y el receptor pueda ir recibiendo paquetes lo más rápido posibe, mientras el emisor debe detenerse si la ventana de envío se le llena con paquetes que no han sido recibidos aún. Para eso, manejen los mutex de forma que los bloques protegidos sean lo más pequeños posibles y asegúrense de soltarlos de vez en cuando para que el otro thread pueda entrar.

Mantendremos entonces los argumentos: el tamaño de paquete que se usará y el tamaño de la ventana (N) cuando no usemos Stop-and-Wait.

Para retransmitir, se les pide implementar un timeout de 0.5s.

Al terminar el envío, deben enviar un paquete de tamaño cero (es decir, los 4 números de secuencia correspondientes y nada más). Al recibir ese paquete, saben que es el último de la secuencia.

Si están en Stop-and-Wait o Go-Back-N, recibir el último paquete les permite terminar todo. Si están en selective-repeat, no es tan simple, por que pueden haber paquetes pendientes de recepción aun. En ese caso, sólo pueden terminar cuando sacan el paquete vacío de la ventana de recepción.

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).

El cliente que deben escribir recibe el tamaño de paquete a proponer, la ventana, archivo de entrada, archivo de salida, servidor, puerto. Al terminar escribe en la salida estándar un resumen de lo medido, como en la Tarea 1, pero ahora agregamos los errores que son: retransmisiones y paquetes recibidos fuera de orden.

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

Hay un servidor de eco corriendo en anakena.dcc.uchile.cl puerto 1818 UDP. Pero, 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, maxwin: 10
Tiempo total: 5.32519793510437s
Errores: 20
```

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.

Para el desarrollo del programa, les recomiendo ir en orden: implementar primero un stop-and-wait, luego hacer un Go-Back-N y luego un Selective-Repeat. De esa forma, van asegurando nota en la tarea si se demoran más de lo esperado (estas tareas son más complicadas en la práctica de lo que parecen en la teoría).

De la misma forma, para cada protocolo, les recomiendo probar primero en localhost hasta que todo funcione, y ahí pueden ir a probar a anakena, donde todo será más difícil.

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 ventanas (con anakena). Mire el tiempo total que toma la transferencia y recomiende los mejores valores para su caso según sus resultados.

2. En la misma conexión que la pregunta anterior, pruebe con el cliente/servidor de eco TCP (`client_echo3.py` y `server_echo2.5.py`) y tome el tiempo de la transferencia del mismo archivo anterior. Hay un servidor TCP de eco corriendo en anakena en el puerto 1818. Comente los resultados.

3. Si implementó más de un protocolo, compare Stop-and-Wait, Go-Back-N y Selective-Repeat. ¿se parecen sus resultados a los del simulador?

4. Modifique los valores del timeout, por ejemplo a 0.01s y 1.0s. ¿Cuánto cambian los resultados?

5. De toda esta experiencia, ¿qué cree Ud que es más importante para ser eficientes?: ¿tamaño paquete, tamaño ventana, protocolo, timeout?

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.