Tarea 4: Cliente eco UDP con Go-Back-N para medir performance
Redes
Plazo de entrega: 30 de mayo 2025

José M. Piquer

1. Descripción

Su misión, en esta tarea, es modificar el cliente UDP y Stop-and-Wait con 2 threads que hicieron en la T3, para que ejecute un protocolo Go-Back-N para corregir los errores en la conexión UDP en forma eficiente. El servidor que usaremos para la medición es el mismo de la T3: `server_udp3.py` (se provee como material docente, no está en el código estándar del curso).

Igual que en la T3, 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).

Igual que en la T3, se debe terminar el envío con un paquete UDP vacío (cero bytes) que hace de EOF. Cuando el receptor detecta este paquete, debe terminar la ejecución.

Deben definir un timeout máximo de espera de 15 segundos en el socket para el receptor, con `settimeout()`. Cuando ocurre este timeout deben terminar con un error, pero esto no debiera ocurrir nunca si el protocolo está bien implementado. Este valor no es lo mismo que el timeout de retransmisión, que se recibirá de parámetro (y obviamente debe ser inferior a 15s).

Pueden medir el tiempo de ejecución y usar el tamaño del archivo de salida como la cantidad de bytes transmitidos.

El cliente que deben escribir ahora recibe el tamaño de lectura/escritura, el timeout de retransmisión, el tamaño de la ventana de envío, el archivo de entrada, el de salida, el servidor y el puerto UDP.

```
./client_gbn_bw.py size timeout win 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 UDP.

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. Protocolo Go-Back-N

Este caso tiene algo muy distinto y raro con respecto a las implementaciones clásicas: Uds están haciendo un protocolo que conversa entre el thread enviador y el thread receptor que corren en el *mismo cliente*. El servidor no participa del protocolo, simplemente hace eco de los paquetes que Uds le envían. El envío y recepción usarán la misma lógica habitual de Go-Back-N (número de secuencia, enviar ventana y esperar hasta que se reciba, retransmitir toda la ventana en caso de timeout). Pero, no usaremos ACKs como mensajes, ya que no son necesarios si estamos en el mismo proceso: basta con que el receptor le informe al enviador que recibió bien un paquete.

La forma más sencilla en Python es usar `Condition` y una variable compartida para esto.

Se les pide implementar el protocolo según las especificaciones siguientes (se revisará la implementación para que lo cumpla):

1. Números de secuencia: 000-999 como caracteres de largo fijo (3) y se reciclan cuando se acaban (el siguiente a 999 es 000). Todo paquete enviado/recibido va con estos tres caracteres de prefijo, incluso el paquete vacío que detecta el EOF (que consiste en sólo el número de secuencia, un paquete de largo 3 bytes).

2. Timeout de emisión: si pasa más de ese timeout sin haber recibido el paquete de vuelta, retransmitimos toda la ventana.

3. Cálculo de pérdidas: al retransmitir una ventana, incrementamos un contador de errores y un contador de paquetes retransmitidos por cada paquete de la ventana.

4. Al terminar el proceso, imprimimos un mensaje con el total de paquetes del archivo, los errores, el porcentaje de error, la cantidad de paquetes transmitidos en realidad (con las retransmisiones) y el porcentaje de paquetes extras transmitidos.

5. Además, se les pide escribir el tamaño máximo que tuvo la ventana de envío y una estimación del tiempo de ida y vuelta hacia el servidor (rtt estimado). Esta estimación se hace anotando la hora de envío del paquete y restándola de la hora de recepción de su eco. Los paquetes retransmitidos deben ser ignorados para este cálculo. Se les pide ir haciendo un promedio ponderado en el tiempo, donde la última medición vale un 50 % y el promedio acumulado un 50 %.

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

```
% time ./client_gbn_bw.py 1400 0.1 100 /etc/services OUT anakena.dcc.uchile.cl 1818
sent 487 packets, retrans 11, 2.2587268993839835%, tot packs 1471, 202.05338809034907
Max_win: 100
rtt est = 0.07572424230420438
4.38 real         0.17 user         0.14 sys
```

Un esquema del protocolo va en la página siguiente.

3. 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 y el timeout 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.

Midan y prueben en las mismas condiciones que probaron la T3. Reporten sus resultados y las diferencias que obtengan. Este protocolo debiera ser mucho más eficiente que Stop-and-Wait. Prueben con distintos valores de ventanas, buscando un óptimo.

Responda las siguientes preguntas (se pueden basar en los valores medidos en la T1 y en la T3 como referencia de los que son valores "correctos", la T2 no sirve porque se perdían muchos datos):

1. Verifique que una ventana de tamaño 1 funcione y demore lo mismo que Stop-and-Wait

2. ¿El protocolo funciona con una ventana de tamaño 1000? ¿Debería fallar?

3. Una gracia de Go-Back-N es que un ACK para n implica un ACK para todo $i <= N$ en la ventana de envío. En esta tarea, que no hay ACKs, ¿esta propiedad sirve de algo? ¿O simplemente es inútil?

4. Hints de implementación

1. Como los números de secuencia son solo 1000, lo más simple es implementar la ventana de envío como subíndices dentro de un arreglo de tamaño 1000, y así podemos usar de subíndice el número de secuencia de cada paquete. El tamaño máximo de la ventana se implementa simplemente como la distancia entre ambos subíndices. Ojo que la ventana es un arreglo circular dentro del arreglo grande, entonces, si estoy en la posición 999 y necesito ir a la siguiente, debo pasar al 0 (y así "entro" por el otro lado del arreglo).

2. También necesito la hora de envío de cada paquete de la ventana y si ha sido retransmitido o no. Estos pueden ser arreglos paralelos de 1000 elementos, o usar una especie de estructura como elementos de la ventana.

3. Proteger el send() con un try/except es buena idea por que cuando la ventana es grande podemos saturar el socket y nos da un error que podemos ignorar, será simplemente como una pérdida de paquete.

4. No es necesario programar un timeout por cada paquete de la ventana. Como se retransmite toda junta, en realidad sólo necesito programar un timeout para el primer paquete (que siempre es el más antiguo). Sí necesito conocer la hora de envío para todo paquete de la ventana (y modificarla cuando se retransmiten). Este timeout programado, usualmente es el que se usa de parámetro en el wait() que hace el enviador cuando la ventana está llena. Pero también debemos revisarlo si la ventana no está llena e igual toca retransmitir.

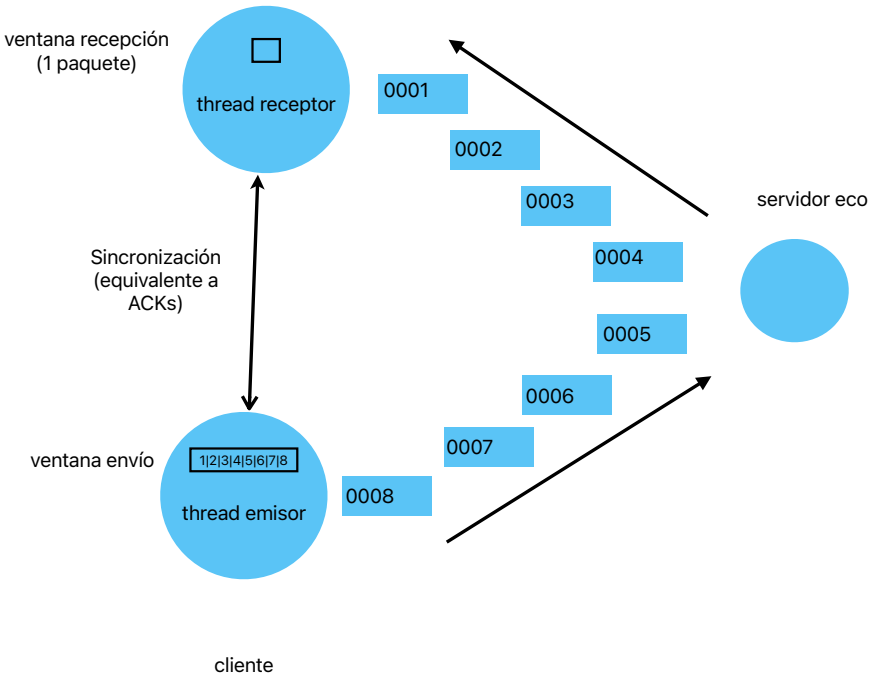
5. Un pseudo-código del enviador propuesto es:

```
while not (eof and ventana vacía):
    correr la ventana hasta el último eco recibido bien

    while ventana llena:
        tout = proximo timeout
        if tout < 0:
```

```
            tout = 0
        if not win_cond.wait(tout):
            retransmitir ventana

        correr la ventana hasta el último eco recibido bien

# Aquí sé que hay un espacio en la ventana
    if not eof:
        data = leer archivo
        if fin de archivo:
            eof = True

        agregar data a la ventana con su hora de envío
        enviar data

# Reviso si hay que retransmitir la ventana
    if proximo timeout < Now():
        retransmitir ventana
```

5. Entregables

Básicamente entregar el archivo con el cliente que implementa el protocolo, una descripción de los experimentos y un archivo con los resultados medidos, tanto para localhost como para anakena, una comparación con la T1, la T2, la T3 y sus respuestas a las preguntas.


ventana recepción
(1 paquete)
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