Tarea 5: Cliente eco UDP con Selective Repeat para medir performance
Redes
Plazo de entrega: 13 de junio 2025

José M. Piquer

1. Descripción

Su misión, en esta tarea, es modificar el cliente UDP y Go-Back-N con 2 threads que hicieron en la T4, para que ejecute un protocolo Selective Repeat para corregir los errores en la conexión UDP en forma mucho más eficiente. El servidor que usaremos para la medición es el mismo de la T4: `server_udp3.py` (se provee como material docente, no está en el código estándar del curso).

Igual que en la T4, 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 T4, 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 y recepción, 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 Selective Repeat

Partiendo desde la T4, deben modificar profundamente el Receptor, que pasa de tener una ventana de tamaño 1 a tener una de tamaño `MAX_WIN`. El enviador queda bastante parecido, excepto en el manejo de las retransmisiones ya que ahora no se envía la ventana entera sino sólo los paquetes cuyo timeout ha expirado.

Al igual que en la T4, 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 Selective Repeat (número de secuencia, enviar ventana y esperar hasta que se reciba, retransmitir un paquete 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 algunas variables compartidas 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 ese paquete.

3. Cálculo de pérdidas: al retransmitir un paquete, incrementamos un contador de errores.

4. Al terminar el proceso, imprimimos un mensaje con el total de paquetes del archivo, los errores, el total de paquetes transmitidos en total (buenos más retransmisiones) y el porcentaje de error:

   ```
   sent 682 packets, retrans 16, tot packs 698, 2.346041055718475%
   ```

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 %.

6. BONUS: Ganan un punto en la tarea si implementan un timeout variable, que sea $1{,}5 * rtt$ usando la estimación de RTT implementada. El valor de timeout recibido de argumento sólo sirve para inicializar el timeout antes de lograr un cálculo de RTT.

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

```
% time ./client_sr_bw.py 1400 0.5 200 /etc/services OUT anakena.dcc.uchile.cl 1818
sent 682 packets, retrans 16, tot packs 698, 2.346041055718475%
Max_win: 200
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 T4. Reporten sus resultados y las diferencias que obtengan. Este protocolo debiera ser más eficiente que Go-Back-N cuando hay pérdidas. Prueben con distintos valores de ventanas, buscando un óptimo.

Responda las siguientes preguntas (se pueden basar en los valores medidos en la T4 como referencia):

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

2. ¿Cuál es el tamaño máximo de ventana teórico para Selective Repeat ahora? ¿De verdad falla si usamos un tamaño mayor?

4. Hints de implementación

1. Como los números de secuencia son solo 1000, lo más simple es implementar la ventana de envío y la de recepción como subíndices dentro de arreglos 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. Ahora sí es necesario programar un timeout por cada paquete de la ventana. Para evitar colas de timeouts o cosas complicadas, lo más simple es buscar el mínimo timeout que haya dentro de la ventana de envío (el paquete más antiguo, que ahora no necesariamente es el primero) y despertar para ese momento. 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 algún paquete.

5. Un pseudo-código del enviador propuesto es:

```
while not (eof and ventana envío vacía):
    with win_cond:
        correr la ventana de envío hasta el último eco en orden recibido bien

        while ventana llena:
            tout = proximo timeout
            if tout < 0:
                tout = 0
            if not win_cond.wait(tout):
                retransmitir paquetes expirados que no hayan sido recibidos

            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 en la ventana
    if proximo timeout < Now():
        retransmitir paquetes expirados que no hayan sido recibidos
```

6. Un pseudo-código del receptor propuesto (que ahora se complica mucho más) es:

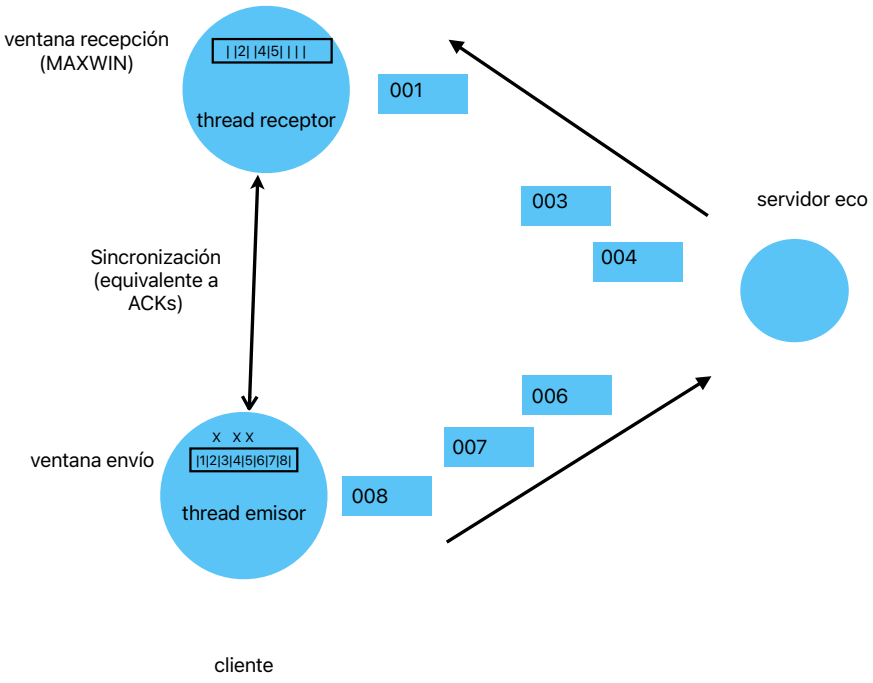
```
while not eof:
    seq, data = recv del socket

    with win_cond:
        if seq dentro de la ventana de recepción
            if seq == esperado:
                if len(data) == 0:
                    eof = True
                    break
                else
                    escribir data a archivo
                corro ventana de recepción revisando todos los
                paquetes ya recibidos y avisados al sender
                escribiéndolos al archivo (y revisando eof)
                win_cond.notify()
            else: # dentro de la ventana pero fuera de orden
                guardar en la ventana de recepción
                anotarlo como "recibido" para el sender
                (para que no lo retransmita)
        else: # lo ignoro
            continue
```

7. Para ver si un paquete está dentro de la ventana, esta función es útil:

```
# Revisar si min <= seq < max (dentro de la ventana circular)
def between(seq, min, max):
    if min <= max:
        return min <= x and x < max
    else:
        return min <= x or x < max
```

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 T4 y sus respuestas a las preguntas.


ventana recepción
(MAXWIN)
| |2| |4|5| | | |
thread receptor
001
003
004
servidor eco
Sincronización
(equivalente a
ACKs)
006
007
x x x
|1|2|3|4|5|6|7|8|
ventana envío
thread emisor
008
cliente