# Tarea 6: Cliente eco UDP con Selective Repeat adaptativo para medir performance
# Redes
# Plazo de entrega: 30 de junio 2025

*José M. Piquer*

## 1. Descripción

Su misión, en esta tarea, es modificar el cliente UDP y Selective Repeat con 2 threads que hicieron en la T5, para que ahora sea capaz de adaptarse a redes dinámicas: le agregaremos timeouts adaptativos y ventana de congestión.

El servidor que usaremos para la medición es el mismo de la T5: `server_udp3.py` (se provee como material docente, no está en el código estándar del curso).

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

```
./client_sr2_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 T5, deben modificar el receptor, para que vaya modificando los timeouts según el RTT medido y el enviador para que maneje una ventana de congestión cuando enfrente pérdidas.

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. Inicializamos el valor con el parámetro de argumento, pero luego lo vamos calculando dinámicamente según el RTT medido.

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 el tamaño mínimo que tuvo la ventana de congestión y el RTT máximo encontrado.

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

```
% time ./client_sr2_bw.py 1400 1 400 /etc/services OUT anakena.dcc.uchile.cl 1818
sent 487 packets, retrans 87, tot packs 574, 17.864476386036962%
Max_win=400, min_cong=200, rtt_max=0.15378832817077637
0.65 real         0.24 user         0.17 sys
```

Un esquema del protocolo va en la página siguiente.

## 3. Timeouts dinámicos y Ventana Congestión

La parte nueva de esta tarea debe ser implementada como sigue (es una simplificación de los protocolos vistos en clases):

1. Timeouts: Se inicializa un timeout al valor recibido de parámetro. Cada vez que se recibe una respuesta, se calcula el RTT si no es una retransmisión. Nos iremos quedando siempre con el máximo valor de RTT medido durante esta conexión. El timeout utilizado para retransmitir los paquetes será el doble de ese valor. Pero, si es una retransmisión, duplicamos el valor del timeout por si el RTT estuviera creciendo.

2. Ventana Congestión: En el enviador, se inicializa una ventana de congestión del mismo tamaño de la ventana de envío. Cuando se retransmite un paquete, la ventana de congestión disminuye a la mitad. Pero, no debemos volver a disminuirla hasta que no pase el tiempo de un RTT máximo (calculado por el receptor). Cada vez que el enviador corre la ventana de envío por que esos paquetes ya fueron recibidos, la ventana de congestión se incrementa en una posición. Para hacer esto, no es necesario esperar un RTT (así se recupera más rápido).

## 4. 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 T5. Reporten sus resultados y las diferencias que obtengan. Este protocolo debiera ser más eficiente que Selective Repeat puro al adaptarse a las pérdidas y al RTT. Prueben con distintos valores de ventanas.

Responda las siguientes preguntas (se pueden basar en los valores medidos en la T1 y en la T5 como referencia):

1. Compare los resultados medidos con la T1 y la T5. Discuta las diferencias encontradas

2. ¿Hay algún escenario en que esta implementación pueda comportarse peor que la T5?

## 5. 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).

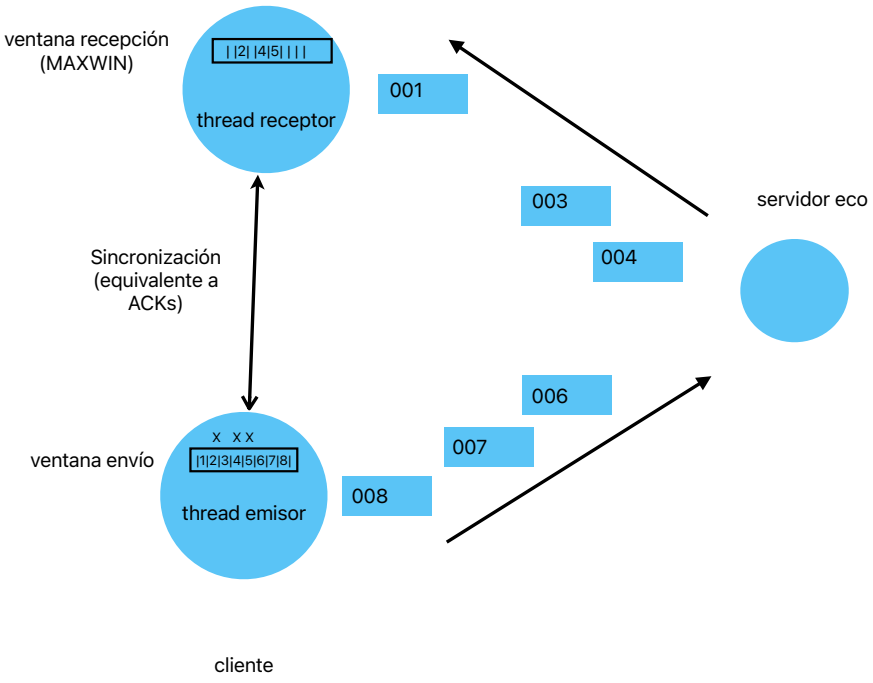
2. También necesito la hora de envío de cada paquete de la ventana y si ha sido retransmitido o no. Estos pueden ser arreglos paralelos de 1000 elementos, o usar una especie de estructura como elementos de la ventana.

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. 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.

## 6. 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 y/o el servidor en AWS, una comparación con la T1 y T5 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