# Tarea 2: medidor ancho de banda Stop-and-Wait Redes Plazo de entrega: 9 de mayo 2022

*José M. Piquer*

## 1 DESCRIPCIÓN

Su misión, en esta tarea, es modificar el cliente de la Tarea 1, para que permita medir ancho de banda total entre él y un servidor usando Stop-and-wait para corregir los errores de transmisión. Este cliente lee un archivo de entrada y lo envía completo al servidor (quien lo almacena). Luego intercambian los roles y el servidor lo envía de vuelta hacia el cliente, quien lo escribe en un archivo de salida. Una vez terminado, el cliente suma los bytes enviados y recibidos y los divide por el tiempo transcurrido desde el inicio (usar la función `time.time()`), entregando un número de Megabytes por segundo (1 Megabyte = 1024*1024 bytes). Como el protocolo retransmite información, queremos medir el ancho de banda realmente logrado para el usuario, por lo que las retransmisiones no deben contarse en los bytes enviados o recibidos.

Siendo un socket UDP, habrá pérdida de paquetes, y ahora les pedimos que los corrijan, usando un protocolo Stop-and-Wait. Si todo funciona bien, será más lento que la Tarea 1, pero esta vez `filein` será idéntico a `fileout` si el programa termina correctamente.

En esta tarea se mantiene el problema de los tamaños de paquete y agregamos ahora el tamaño de los timeouts para retransmitir. Como en la tarea 1, el cliente le envía al servidor su propuesta de valores y el servidor responde con los que hay que usar.

Para poder probar eso, el archivo de entrada debe leerse en bloques de bytes, del tamaño del paquete máximo, para ser enviados. En Python, pueden usar archivos binarios (de bytes) para eso y la función `read()`.

Hay un servidor corriendo en anakena.dcc.uchile.cl puerto 1819 UDP. El servidor de la tarea 1 seguirá corriendo en el 1818.

El cliente debe invocarse como:

```
% ./bwc timeout size filein fileout host port
bytes=678397, time=0.06311702728271484, bw=10.250321079431274 MBytes/s
```

timeout es un entero con el tiempo (en milisegundos) que deben esperar un ACK a un paquete enviado antes de retransmitir.

size es un entero con el máximo de bytes de un paquete de datos que deben proponer como cliente. Este tamaño limita el máximo paquete que se escribe en el socket vía send(), por lo tanto, cuando le agregamos un *header* debemos reducir la cantidad de datos que le agregamos después. Si acordamos un paquete de tamaño máximo 1.000 bytes, por ejemplo, y el protocolo usa 2 bytes de header (como en esta tarea), significa que sólo podremos enviar 998 bytes de datos por paquete, para que su tamaño total sea justo 1.000 bytes.

Ejemplo:

```
% ./bwc 250 8000 filein fileout anakena.dcc.uchile.cl 1819
```

Le propone al servidor de anakena un timeout de 250 ms (0.25 segundos) y un tamaño de datos de 8.000 bytes.

## 2 PROTOCOLO

Implementamos un Stop-and-Wait con número de secuencia en los paquetes y en los ACKs.

1. Cliente: envía 'C0xxxxxyyyyy'
2. Servidor: responde 'A0zzzzzwwwww'
3. Cliente: envía paquetes con todo el contenido de filein, siempre comienzan con 'D' y el número de secuencia (0 o 1 solamente, en Stop and wait no necesito más):
4. Cliente: envía 'D1kkkkkkkkkk'
5. Servidor: responde 'A1'
6. Cliente: envía 'D0jjjjjjjjjjj'
7. Servidor: responde 'A0'
8. Cliente: envía 'D1sfsdfsdfsfsdf'
9. Servidor: responde 'A1'
10. Cliente: envía 'D0sddsdasd'
11. Servidor: responde 'A0'
12. ....

13. Cliente: envía 'En' (n es 0 o 1)
14. Servidor: envía 'An'
15. Servidor: envía paquetes con todo lo recibido antes, también todos comienzan con 'D' y número de secuencia
16. Cliente: responde 'An' por cada paquete recibido en orden
17. Servidor: envía 'Em' (m es 0 o 1)
18. Cliente: envía 'Am'

En Python, estas secuencias son "bytearray". Los números xxxxx zzzzz son los tamaños de paquete máximo, y yyyyy y wwwww son los timeouts, codificados como "bytearray"de números como caracteres. Por ejemplo, si el cliente quiere usar un tamaño de 10.000 bytes y 500 ms, enviaré:

```
C01000000500
```

Como son paquetes UDP en el socket, no necesitan fin de línea después. Ojo que ahora va un byte con el número de secuencia siempre, y la conexión y el EOF deben retransmitirse igual que un paquete de datos para asegurarnos que el protocolo funcione.

Los números de secuencia que usa el cliente no son los mismos que usa el servidor, cuando el cliente empieza a enviar los datos del archivo, debe comenzar con el número 1 (usó el 0 en la conexión). El EOF va con el número de secuencia que le tocó como si fuera un dato siguiente al último.

Cuando el servidor comienza a enviar el archivo de vuelta, él comienza con la secuencia 0 (él no ha usado ninguna).

Ojo al recibir los datos del servidor, deben estar preparados para recibir un ACK atrasado (el del EOF que es lo último que le enviaron). Si lo reciben mientras esperan más datos, deben ignorarlo no más.

Cada paquete enviado (salvo los ACKs) debe reintentarse 10 veces antes de abortar la conexión y terminar el programa con error.

Para implementar timeout, les recomiendo usar la opción timeout del socket mismo, por ejemplo:

```
s.settimeout(0.1)
```

Espera máximo 0.1s cada vez que invocan s.recv() (0.1 s == 100 ms). Pasado ese tiempo, si no han llegado datos, genera una excepción.

## 3 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 timeouts y haga una recomendación de combinación en base a sus resultados (entregar los experimentos realizados y sus resultados junto con las conclusiones)

2. Compare el ancho de banda medido con la Tarea1. ¿Cuál es más realista? ¿Por qué la diferencia?

3. Al terminar el cliente, envía un ACK del EOF. Pero ese ACK se puede perder y el servidor seguirá re-enviando el EOF inútilmente. ¿Se puede terminar bien este protocolo? ¿Tiene sentido tratar de arreglar esto?